Can High-Level Synthesis Compete Against a Hand-Written Code in the Cryptographic Domain? A Case Study*

Ekawat Homsirikamol and Kris Gaj
Volgenau School of Engineering
George Mason University
Fairfax, Virginia 22030
email: {ehomsiri, kgaj}@gmu.edu

***Abstract*—This paper investigates the state of the current high-level synthesis (HLS) tools by using Xilinx Vivado HLS for designing a cryptographic module based on Advanced Encryption Standard. The obtained results are compared with the results for the hand-written Register-Transfer Level (RTL) VHDL code to determine the suitability of the HLS-based approach for implementing cryptographic algorithms in hardware. Our study has shown that the RTL-based approach still outperforms the HLS-based approach due to the flexibility in designing a control unit, which affects the throughput of the circuit. Nevertheless, the HLS-based approach can successfully compete with the RTL-based approach in terms of area and maximum clock frequency.**

***Index Terms*—FPGA, High-level synthesis, Advanced Encryption Standard, AES, Cryptography**

I. Introduction & Motivation

Hardware Description Language (HDL) source codes, while easier to generate than gate-level descriptions, remain difficult to create and verify, as compared to programs written in traditional high-level programming languages, such as C. As a result, high-level synthesis (HLS) has emerged as a next step in the development of modern hardware design methodologies. At the same time, in spite of HLS tools being developed since 1980s [1], HDLs, such as VHDL and Verilog, have remained the primary languages of choice for hardware designers. Simultaneously, the latest generation of HLS tools have been gaining wider acceptance in specific domains, particularly in digital signal processing (DSP), as over time the HLS tools have become significantly more powerful and easier to use [1].

Even though HLS has been slowly gaining traction in the hardware designers community in the past decade, the entry cost for industrial-quality tools was prohibitively high, especially for small and medium-size companies, as well as academia. Integration with the logic synthesis and implementation (mapping, placing, and routing) tools was also non-existent, which further exacerbated the adoption of HLS-based approach. This situation has changed when Xilinx released Vivado Design Suite in the second half of 2012 [2]. The integration of HLS tool, Vivado HLS, at no additional cost, in the widely adopted industrial-level hardware development platform has provided unprecedented access to this design methodology. The question remains, however, whether this approach can compete against the tried-and-true method of hand-writing the RTL code using either VHDL or Verilog.

Earlier studies have demonstrated that HLS tools can, undoubtedly, generate FPGA designs that outperform the corresponding software implementations [3], [4]. Nevertheless, outside of the DSP domain, with the exception of [5], little is known about the performance of FPGA designs generated using the HLS-based approach in comparison to those obtained using the RTL-based methodology [6], [7]. This question remains in particular open in the field of cryptography and network security. The specific challenges of this domain include: iterative algorithms, not suitable for parallel processing, large look-up tables, and domain-specific operations, such as permutations, variable shifts and rotations, etc. These specific features can be more difficult for an HLS tool to infer from a high-level language description, as compared to the standard DSP operations, such as add, multiply-and-accumulate, etc.

Previous studies [8]–[12] in which HLS-based approach has been used to implement cryptographic algorithms have not answered this question in a satisfactory way. They were more concerned with determining whether HLS can be possibly used to develop complex cryptographic systems. As a result, this paper aims to determine whether the current generation of HLS tools can produce HDL code for modern cryptographic algorithms that can successfully compete against traditional hand-written RTL designs. And if not, what can be possibly done to improve the performance of the digital systems generated using HLS-based approach.

II. Methodology

In this study, we focus on the most popular modern secret-key cipher, Advanced Encryption Standard (AES) [13]. For a mode of operation, we choose the Counter (CTR) mode [14], used in multiple internet protocols [15], [16] due to its simplicity, speed, and suitability for parallel processing. Although in general, AES supports three different key sizes, corresponding to three different security levels, in this study,

*This work is supported by NSF Grant #1314540

we limit our implementations, for simplicity, to AES with a 128-bit key size.

The design process follows the same intermediate steps, in both RTL and HLS-based approaches. First, AES in the Electronic Codebook mode (AES-ECB) is implemented, as a basic building block. As our target is AES-CTR, only the encryption mode is supported in the AES-ECB. This module is our core design, and is denoted as AES-ECB-ENC. In both design approaches, the AES-ECB-ENC modules were developed to produce the same basic iterative architecture. Using the same architecture allows us to focus exclusively on the differences in the quality of the HDL codes: hand-written RTL code and the code generated automatically by Vivado HLS. In order to infer basic iterative architecture, the reference software implementation in C has been extended with several compiler directives, specified as pragmas. At the same time, no timing constraints were specified, to prevent the tool from inferring any unintended architectures. Afterwards, the Counter mode has been added on top of the AES-ECB-ENC unit. Finally, the input and output processing units have been added to adhere to the interface and protocol specifications described in Section III-A.

The initial C source code used as a starting point in the HLS-based approach is the original reference software implementation of Rjindael [17]. While there exist numerous other software implementations of AES, including the implementation included in the open-source benchmarking suite targeting HLS, CHStone [18], the reference implementation was selected to demonstrate the entire process of porting a typical software implementation to hardware. The initial C source code used for HLS, denoted in the rest of the paper as HLSv0, was a stripped down version of the original code with the removed support for additional block and key sizes, other than 128.

As demonstrated in the rest of the paper, an outcome of the HLS-based approach is highly dependent on the quality of the source code and the synthesis tool directives added to the C code as pragmas. As a result, multiple steps are required in order to generate the desired hardware architecture. At each step, functional verification is performed in both software and hardware to ensure the correctness of the design. The synthesis of HDL code in the HLS-based approach is performed using Vivado HLS v2014.1.

In order to make our comparison fair, a further reduction of discrepancies between the RTL and HLS-based approaches is achieved by using exactly the same HDL language, FPGA tools, and FPGA tool option optimization techniques. For HDL language, VHDL is used as a language of choice for the handwritten RTL code, and as a target for HLS. For logic synthesis and implementation (mapping, placing, and routing), ISE Design Suite v14.7 and Quartus II v13.0sp1 are used, for Xilinx and Altera FPGAs, respectively. Finally, the uniform optimization of FPGA tool options was facilitated by ATHENa [19].

The results of each optimization applied to the HLS source code were analyzed against the results obtained using the hand-written RTL VHDL code. Results were generated for two FPGA families from two primary FPGA vendors, Xilinx and Altera, using one high-performance and one low-cost family from each vendor. For the ease of comparison of the obtained results (especially area and throughput to area ratio), the tools were configured to avoid using any dedicated FPGA resources, such as block memories and DSP units.

III. Reference Design

A. Interface and Protocol

The interface and the communication protocol used in this study are the extended versions of the solutions proposed for hash functions in [20]. The modified interface is shown in Figure 1. The input ports are separated into those used for public and secret data. Public data ports, pdi, are used for non-sensitive inputs, such as initialization vectors (IVs), messages, and ciphertexts. Secret data ports, sdi, are used for sensitive inputs, such as keys. Output data ports, do, are used for all outputs from the cryptographic core.

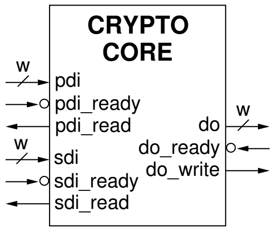

Fig. 1. Input/Output interface of a cryptographic core.

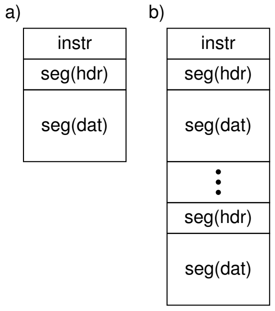

Fig. 2. Formats of inputs, using a) one segment, b) using multiple segments.

The format of inputs is shown in Figure 2. Each input contains an instruction word, instr, notifying the core whether the next operation to be performed is encryption or decryption. This word is then followed by a word of a segment header, sec(hdr). The header contains information such as the segment type (IV, Plaintext, Ciphertext, or Key), the last-segment flag, and segment size. Finally, the header is followed by multiple data words, sec(dat). Multiple segments are used if inputs of different types are required (e.g., an IV and a message in the counter mode), or when the data is too big to fit in a single segment. Figure 2b presents an example of the input format, when message is split into multiple segments. It must be noted that input messages are assumed to be padded with zeroes to a multiple of the I/O data bus width, w.

B. Top-level Block Diagram

The top-level block diagram of the full cryptographic module implementing AES in the counter mode is shown in Figure 3. It consists of four modules, *input processor*, *bypass FIFO*, *cryptographic core*, and *output processor*, which operate independently and in parallel to each other.

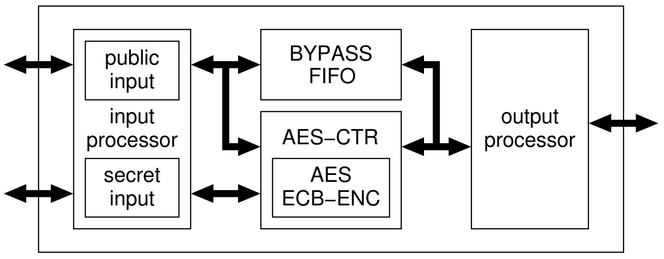

Fig. 3. Top-level block diagram of the full cryptographic module implementing AES in the counter mode.

The *input processor* includes two sub-modules, *public input* and *secret input*, which can operate concurrently, if needed. The *public input* unit analyzes public input data, based on our communication protocol, and provides the IV and message/ciphertext to the cryptographic core. It also passes information that does not need to be processed to the *bypass FIFO*. The *secret input* unit performs a similar task for the secret key. Together the two units form the *input processor* that provides necessary data and control signals to the cryptographic module (*AES-CTR*). The *bypass FIFO* forwards any public data that does not need to be processed, but needs to be included in the output from the top-level unit, to the *output processor*.

AES-CTR is the main cryptographic module used in this study, implementing AES-128 in the counter mode. This unit combines the 96-bit IV with the 32-bit counter used in the counter mode, and provides data and the associated key to the AES-ECB-ENC submodule. The output of the submodule is XORed with the input message to form the ciphertext or plaintext (for encryption and decryption, respectively), which is then written to the *output processor*. The *output processor* converts outputs from the *AES-CTR* module to the format specified by our protocol. The type and size of data segments is determined based on the formatting information passed from the public data input by the *bypass FIFO*.

Since all the aforementioned units must operate independently, in the HLS-based approach, they are synthesized separately, independently of each other. They are then combined into a single top-level unit using component instantiation in VHDL. The input and output ports of each module generated using HLS are controlled via **INTERFACE** pragmas. These pragmas allow a wide range of options. In our design, we have mainly used a standard handshake (**ap_hs**) and FIFO (**ap_fifo**) interfaces for communication among submodules.

C. AES-ECB-ENC (Core Unit)

The core unit developed using our RTL-based approach is shown in Figure 4. This unit calculates all round keys on the fly. *BDI* and *BDO* in the diagram refer to block data input and

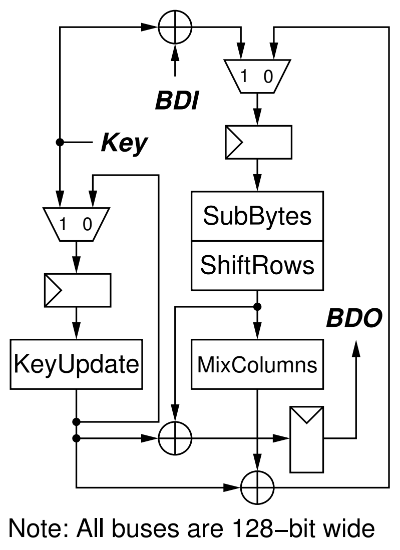

Fig. 4. AES-ECB-ENC module

block data output, respectively. In the first clock cycle, *Key* is *XORed* with BDI and stored in the state register. The *Key* is also used to initialize the round-key register in the left side of the diagram. In each subsequent clock cycles, a new round key is calculated together with an output from the next round. In the final clock cycle, MixCol is bypassed, and the result of encryption is stored in the output register.

IV. Results and Discussions

A. Basic Iterative Architecture of AES-ECB

TABLE I
Optimization Results of AES-ECB-ENC

Imp.	Area (LEs/ ALMs/SLICEs)	Memory/ BRAMs	Latency	Freq. (MHz)	TP (Mbits/s)	TP/Area
Altera Cyclone IV						
RTL	4925	0	11	143	1833	0.37
HLSv0	N/A	N/A	7367	N/A	N/A	N/A
HLSv1	N/A	N/A	3224	N/A	N/A	N/A
HLSv2	4934	0	11	144	1540	0.31
Altera Stratix V						
RTL	1134	0	11	413	5291	4.67
HLSv0	N/A	N/A	7367	N/A	N/A	N/A
HLSv1	N/A	N/A	3224	N/A	N/A	N/A
HLSv2	1169	0	11	424	4520	3.87
Xilinx Spartan 6						
RTL	354	0	11	230	2943	8.31
HLSv0	374	0	7367	183	3	0.01
HLSv1	211	0	3224	193	8	0.04
HLSv2	343	0	11	231	2467	7.19
Xilinx Virtex 7						
RTL	317	0	11	336	4298	13.56
HLSv0	318	1	7367	265	5	0.01
HLSv1	236	1	3224	334	13	0.06
HLSv2	344	0	11	352	3760	10.93

The results for the RTL and HLS-based designs of the core unit (AES-ECB-ENC) are shown in Table I. The results for hand-written VHDL code are denoted as RTL. The remaining results, corresponding to three different versions of the C code, post-processed using our HLS-based approach, are denoted as HLSv0, HLSv1, and HLSv2, respectively.

The main differences between our first C source code, HLSv0, and the reference software implementation of Rjindael [17] are as follows: 1) Any code supporting ECB decryption is removed. 2) Any code supporting block and key sizes other than 128 bits are removed. 3) *KeyScheduling* is called at the beginning of the AES encryption function. 4) An additional output port is inserted, and written to only once at the end of the encryption function. While this operation is redundant from the software point of view, it is required for HLS designs. This is because every change to the function arguments is interpreted by HLS tools as a write operation. Since we want only a single write of the encrypted data to be performed during the entire encryption, an output argument should be written to only once during the entire function. 5) Temporary variables are used instead of the input arguments for intermediate results. This is to prevent the tool from creating an output port corresponding to an input argument, in case the input argument is modified.

The results for our initial C code, HLSv0, include the latency of 7367 clock cycles, as compared to 11 clock cycles used by the RTL design. Clearly, we are still very far from our target basic iterative architecture. Additionally, the Altera tool, Quartus, was not able to process the VHDL code generated by Vivado HLS. On the closer inspection of the generated code, we found that Quartus was not able to synthesize the dual-port read/write RAM generated by Vivado HLS. Furthermore, even though we specifically directed the logic synthesis tools not to use any dedicated FPGA resources, one BRAM is still inferred by Xilinx ISE for the Virtex 7 FPGA family. In terms of area, excluding BRAMs, the HLS-based design consumes approximately the same amount of resources (CLB slices) than the RTL-based implementation. The frequency is lower by about 20% for both Spartan-6 and Virtex-7 FPGA families. However, due to a very large latency of this initial HLS-based design, the throughput and throughput to area ratio are far below that of the RTL-based implementation.

In the next step, the *KeyScheduling* function, which is called once per encryption, is replaced by the *KeyUpdate* function, which is called once per each round. This change infers an architecture with round keys calculated on the fly, the same as in the RTL design. The *KeyUpdate* function is shown in Listing 1(a). Furthermore, in the reference software implementation, the Galois field multiplication, used in *MixColumns* function, is based on look-up tables. While this approach is efficient in software, it requires a large amount of resources in hardware. The use of these resources is not well justified, as the only Galois field multiplications required by AES encryption are multiplications by two small constants, $0x02$ and $0x03$. For the purpose of the hardware implementation, a much better solution is the implementation of the multiplication by $0x02$ using the *xtime()* operation defined in [13]. The multiplication by $0x03$, is accomplished by computing $0x03 \cdot A = 0x02 \cdot A \oplus A$. The original and modified implementations of the *MixColumns* function are shown in Listing 1(b). We call the code obtained by performing both modification summarized in Listing 1 HLSv1. The latency of the inferred circuit (3224 clock cycles) is more than two times smaller than the latency of the design inferred by HLSv0 (7367 clock cycles).

Listing 1. Fragments of the code modified between HLSv0 and HLSv1.

```
// (a) The new KeyUpdate function

void KeyUpdate (word8 k[4][4], word8 round) {
    int i, j;
    word8 tk[4][4];
    static const word32 rcon[10] = {
      0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0
          x80, 0x1b, 0x36};

    for(j = 0; j < 4; j++)
      for(i = 0; i < 4; i++)
        tk[i][j] = k[i][j];

    for(i = 0; i < 4; i++)
      tk[i][0] ^= S[tk[(i+1)%4][3]];
    tk[0][0] ^= rcon[round];

    for(j = 1; j < 4; j++)
        for(i = 0; i < 4; i++)
          tk[i][j] ^= tk[i][j-1];

    for(j = 0; j < 4; j++)
      for(i = 0; i < 4; i++)
        k[i][j] = tk[i][j];
}


// (b) Modification of the MixColumns function

/* original code */

b[i][j] = mul(2,a[i][j])
                ^ mul(3,a[(i + 1) % 4][j])
                ^ a[(i + 2) % 4][j]
                ^ a[(i + 3) % 4][j];

word8 mul(word8 a, word8 b) {
        if (a && b) return Alogtable[(Logtable[a]
           + Logtable[b])%255];
              else return 0;
}

/* modified code */

b[i][j] = mul2(a[i][j])
                ^ mul3(a[(i + 1) % 4][j])
                ^ a[(i + 2) % 4][j]
                ^ a[(i + 3) % 4][j];

word8 mul2(word8 a)
{
 if (a >> 7) == 1
   return (a << 1) ^ 0x1b;
 else
   return a << 1;
}

word8 mul3(word8 a)
{
 return mul2(a) ^ a;
}
```

The VHDL code generated by Vivado HLS from the modified C code, HLSv1, cannot be still synthesized by Altera tools. Therefore, only the results for Xilinx FPGA families are reported in Table I. As expected, the replacement of

KeyScheduling by *KeyUpdate* and an alternative implementation of the Galois field multiplication reduced the resource utilization by a significant factor, 44% for Spartan-6 and 26% for Virtex-7. The frequency increased by 5% for Spartan-6 and 26% for Virtex-7. At the same time, the biggest improvement came from the reduction of the circuit latency by 56%, independently of the FPGA family used. As a result, the throughput to area ratio, increased by a factor of 4 for Spartan-6, and by a factor of 6 for Virtx-7. Still, the absolute values of these ratios remained very small compared to the corresponding ratio for the RTL implementation.

The primary reasons for the large latency and small throughput of the HLSv1 design are as follows: First, the unit performs each operation a byte at a time. This is because all major function arguments and state variables are arrays of components of the type word8 (representing 8-bit unsigned integers). The components of these arrays are not yet combined into full-size 128-bit blocks. Secondly, majority of loops have not been unrolled, implying sequential processing of individual bytes. Due to the aforementioned reasons, the second step of optimization involves the insertion of synthesis tool directives aimed at increasing the datapath width from 8 to 128 bits, and as a result reducing the latency of the entire circuit. The following directives, specified using pragmas, were used to infer our target architecture:

- **ARRAY_RESHAPE** pragma, shown in Listing 2(a), is used to reshape an array to be synthesized as a different shape and size array. In order to match our target architecture, our goal is to unroll all 4x4 arrays of 8-bit words into a single word of the size of 128 bits. Without this pragma, all functions that use 4x4 arrays of 8-bit words process only 8-bits per clock cycle, instead of 128-bits per clock cycle, as intended. It must be noted that since we use two-dimensional arrays, sub-arrays must be reshaped as well.
- **UNROLL** pragma, shown in Listing 2(b), is used to unroll loops, so that operations belonging to different iterations of the loop are all performed in parallel. This modification was applied to all major loops used in our C code.
- **INLINE** pragma, shown in Listing 2(c), reduces the latency by combining multiple sub-function calls into a single higher level function. This directive allows multiple functions calls (otherwise taking one clock cycle each) to be executed within one clock cycle.
- **RESOURCE** pragma, shown in Listing 2(d), is used to specify the target resource, such as Block RAM or distributed memory. In our implementation, this directive is specifically used to make sure that all ROMs, used to implement SubBytes and the Rcon constant look-up, are implemented using distributed memory located in multipurpose LUTs.
- **INTERFACE** pragma, shown in Listing 2(e), is used for adding the output register resulting in a design that matches our target architecture. This also used to specify the type of handshake the input/output ports are using. For the core unit, the default handshake is used.

Listing 2. Fragments of the code HLSv2 produced in the second optimization step

```
// (a) Usage of ARRAY_RESHAPE pragma

void AES_encrypt (word8 a[4][4], word8 k[4][4],
    word8 b[4][4])
{
#pragma HLS ARRAY_RESHAPE variable=a[0] complete
    dim=1 reshape
#pragma HLS ARRAY_RESHAPE variable=a[1] complete
    dim=1 reshape
#pragma HLS ARRAY_RESHAPE variable=a[2] complete
    dim=1 reshape
#pragma HLS ARRAY_RESHAPE variable=a[3] complete
    dim=1 reshape
#pragma HLS ARRAY_RESHAPE variable=a complete dim
    =1 reshape

// (b) Usage of UNROLL pragma

    OutputLoop: for (i = 0; i < 4; i ++)
#pragma HLS UNROLL
      for (j = 0; j < 4; j ++)
#pragma HLS UNROLL
         b[i][j]   = s[i][j];

// (c) Usage of INLINE pragma

void KeyUpdate (word8 k[4][4], word8 round) {
#pragma HLS INLINE
...
}

// (d) Usage of RESOURCE pragma

word32 rcon[10] = {
          0x01, 0x02, 0x04, 0x08, 0x10,
          0x20, 0x40, 0x80, 0x1b, 0x36};
#pragma HLS RESOURCE variable=Rcon0 core=ROM_1P_1S

// (e) Usage of INTERFACE pragma

void AES_encrypt (word8 a[4][4], word8 k[4][4],
    word8 b[4][4])
{
#pragma HLS INTERFACE register port=b
```

The addition of design directives has produced the design HLSv2, with 11 clock cycles of latency. A huge reduction in latency compared to HLSv1 was accomplished mostly due to the **ARRAY_RESHAPE** and **UNROLL** pragmas, allowing the design to operate on all bits of a 128-bit internal state in parallel. Additionally, the **RESOURCE** pragma eliminated the usage of BRAMs in Xilinx Virtex-7. On top of that, since the design does not require a true dual-port memory (with two write ports) any longer, the Altera tools have completed their operation without any errors, and produced meaningful results.

Compared to HLSv1, the following changes in resource utilization and performance measures occurred for HLSv2 (see Table I). Area increased by 63% for Spartan-6 and 46% for Virtex-7. At the same time, the clock frequency increased for both Xilinx families, and slightly exceeded the frequency of RTL-based designs for all four investigated Altera and Xilinx FPGAs. Since the latency was dramatically reduced compared to HLSv0 and HLSv1, the throughput and the throughput to area ratio followed, reaching values only slightly smaller than those obtained for the RTL implementation.

The further improvement of the HLS circuit throughput appeared to be very difficult to accomplish, due to the operation of the control unit inferred by Vivado HLS from the C code. Although the two investigated approaches produce designs with the same latency, the intervals between two consecutive data inputs are not the same. By default, the control unit of the HLS-based design always requires one additional clock cycle per block of data for initialization, as it cannot start a new operation in the same clock cycle in which the previous output becomes available. This feature adds one additional clock cycle in the formula for the throughput, which becomes $Throughput = 128 \cdot f_{CLK}/(Latency + 1)$, where f_{CLK} is the maximum clock frequency. On the other hand, the RTL design, permits inputting a new block of data every 10 clock cycles (in general $Latency - 1$ clock cycles), as the precomputations involving the round key(0) for block i+1 can be overlapped with the calculation of the 10th AES round for block i. Thus, the formula for the throughput becomes $Throughput = 128 \cdot f_{CLK}/(Latency - 1)$. The same mode of operation is not supported by the control unit generated by the HLS tool. As a result, the throughput and the throughput to area ratio of the HLS-generated AES Core Design are 17% smaller, even if the frequency and area remain the same as in the corresponding RTL implementation.

TABLE II
COMPARISON OF RESULTS BETWEEN HLS AND RTL OF AES-ECB-ENC WITH ITERATIVE ARCHITECTURE

Imp.	Area (LEs/ ALMs/SLICEs)	Frequency (MHz)	TP (Mbits/s)	TP/Area
Altera Cyclone IV				
RTL	4925	143	1833	0.37
HLS	4934	144	1540	0.31
Δ	+0.2%	+0.8	-16.0%	-16.1%
Altera Stratix V				
RTL	1134	413	5291	4.67
HLS	1169	424	4520	3.87
Δ	+3.1%	+2.5	-14.6%	-17.1%
Xilinx Spartan 6				
RTL	354	230	2943	8.31
HLS	343	231	2467	7.19
Δ	-3.1%	+0.6	-16.2%	-13.5%
Xilinx Virtex 7				
RTL	317	336	4298	13.56
HLS	344	352	3760	10.93
Δ	+8.5%	+5.0	-12.5%	-19.4%

A full comparison between the final HLS and RTL-based designs for AES-ECB-ENC is shown in Table II. The difference in resource utilization is minimal for all investigated FPGA families, except Virtex-7, where the HLS-generated design requires 8.5% more CLB slices. Similarly, the differences in frequencies are smaller than 3% for all investigated families, except Virtex-7, where HLS-generated design outperforms the RTL design by 5%. As a result, a minor difference in the number of clock cycles between two consecutive input blocks, 12 vs. 10, appears to be the only reason for a slight performance disadvantage of the HLS-based design in terms of throughput and throughput to area ratio. This disadvantage does not exceed 20% for any of the four investigated FPGA families. Overall, if not for two additional clock cycles required by the control unit, the HLS-based approach would produce almost exactly the same results as the RTL-based methodology.

B. *Unrolled Architecture of AES-ECB*

A two-times unrolled architecture of the Core Unit, AES-ECB-ENCx2, is explored in this section. For the RTL-based design, the round and key update operations are unrolled, so two round keys and two AES rounds can be calculated in each clock cycle. As a result, the latency of the circuit becomes 6, and the time between two consecutive inputs becomes 5.

In order to infer the two-times unrolled architecture in C, KeyUpdate as well as all other component operations of AES (SubBytes, ShiftRows, MixColumns, and AddRoundKey) are called twice per each iteration of the main encryption loop in C. It must be noted that the HLS directive **UNROLL** could not be used to achieve this goal because both the SubBytes and KeyUpdate operations contain array look ups. Using the same array twice infers resource sharing, thereby increasing the circuit latency. In order to eliminate this undesirable behavior, functions that use lookup tables are manually duplicated, with each of the two versions of a given function assigned a different name. The latency of the generated design and the time interval between two consecutive inputs are equal to 6 and 7 clock cycles, respectively.

TABLE III
COMPARISON OF RESULTS BETWEEN HLS AND RTL OF AES-ECB-ENC WITH UNROLL X2 ARCHITECTURE

Imp.	Area (LEs/ ALMs/SLICEs)	Frequency (MHz)	TP (Mbits/s)	TP/Area
Altera Cyclone IV				
RTL	9478	76	1941	0.20
HLS	9493	76	1391	0.15
Δ	+0.2%	+0.4	-28.3%	-28.4%
Altera Stratix V				
RTL	2087	218	5569	2.67
HLS	2099	218	3986	1.90
Δ	+0.6%	+0.2	-28.4%	-28.8%
Xilinx Spartan 6				
RTL	581	123	3152	5.43
HLS	624	113	2070	3.32
Δ	+7.4%	-8.1	-34.3%	-38.8%
Xilinx Virtex 7				
RTL	633	215	5515	8.71
HLS	560	206	3775	6.74
Δ	-11.5%	-4.2	-31.5%	-22.6%

The results of implementing the two times unrolled architecture of AES using both the RTL and HLS-based approaches are summarized in Table III. For both Altera FPGA families, the area and frequency are almost identical for both approaches. For the HLS design on Virtex-7, both area and frequency decrease compared to the RTL implementation.

These changes have the opposite effect on the throughput to area ratio. For Spartan-6, the area increases and frequency decreases. Thus, both changes affect the throughput to area ratio in the same (negative) direction. However, in both cases, the most substantial influence on the throughput and the throughput to area ratio comes from the interval between two consecutive inputs, which is 5 clock cycles in the RTL design and 7 clock cycles in the HLS-generated design. This difference itself contributes to about 28% of reduction in the values of both performance measures.

C. AES-CTR

While using the Core Unit, AES-ECB-ENC, as a base, a wrapper is added over this design to create a higher-level module implementing AES in the counter mode (AES-CTR). As AES-CTR requires more I/O pins for an initialization vector (IV), no results are provided for the low-cost FPGA devices due to the lack of sufficient I/O resources (please note that multiplexing of inputs is not used at this stage). The results for the extended design are summarized in Table IV.

TABLE IV
COMPARISON OF RESULTS BETWEEN HLS AND RTL OF AES-CTR

Imp.	Area (LEs/ ALMs/SLICEs)	Frequency (MHz)	TP (Mbits/s)	TP/Area
Altera Stratix V				
RTL	1236	412	5274	4.27
HLS	1165	411	4379	3.76
Δ	-5.7%	-0.4	-17.0%	-11.9%
Xilinx Virtex 7				
RTL	461	309	3960	8.59
HLS	557	390	4155	7.46
Δ	+20.8%	+25.9	+4.9%	-13.2%

Compared to the RTL implementation, for Altera Stratix V FPGAs, the area of the HLS-based design improves by about 6% and the frequency remains the same. For Xilinx Virtex-7, the area increases by about 21%, and the frequency by about 26%. These effects almost cancel each other in terms of their influence on the throughput to area ratio. Once again, the main difference in terms of the throughput to area ratio comes from the different number of clock cycles between two consecutive input blocks, 10 for the RTL implementation vs. 12 for the HLS-based design. As the only major surprise, the throughput of the HLS-based design is about 5% higher than the throughput of the RTL implementation for Virtex 7 due to the substantial gain (26%) in terms of the clock frequency. Interestingly, the clock frequency of HLS-based approach for Altera Stratix V remains closely matched to RTL-based approach.

Overall, the decrease in the throughput to area ratio is about 12%-13% for both investigated FPGA families. These results, relatively better than those for the core designs, indicate that the HLS-based approach does well for simple extensions of basic cryptographic cores into those implementing simple modes of operation, such as the counter mode. In the RTL-based design, the wrapper, including the control logic, is added on top of the Core Unit, while, the HLS-based approach can merge the two parts together more efficiently.

D. AES-CTR with I/O

In this section, the feasibility of utilizing I/O modules to handle a complex communication protocol is investigated. In the RTL approach, the input and output modules are created independently, and combined at the top level. The HLS-based design was created in a similar manner. The input and output modules were synthesized separately. The top-level of the HLS-based approach is created using component instantiation, based on the diagram shown in Figure 3.

TABLE V
COMPARISON OF RESULTS BETWEEN HLS AND RTL OF FULL AES ENCRYPTION/DECRYPTION

Imp.	Area (LEs/ ALMs/SLICEs)	Frequency (MHz)	TP (Mbits/s)	TP/Area
Altera Cyclone IV				
RTL	6397	143	1827	0.29
HLS	6877	131	1398	0.20
Δ	+7.5%	-8.1	-23.5%	-28.8%
Altera Stratix V				
RTL	1666	418	5345	3.21
HLS	1835	318	3392	1.85
Δ	+10.1%	-23.8	-36.5%	-42.4%
Xilinx Spartan 6				
RTL	651	217	2782	4.27
HLS	581	149	1586	2.73
Δ	-10.8%	-31.6	-43.0%	-36.1%
Xilinx Virtex 7				
RTL	695	334	4281	6.16
HLS	541	212	2263	4.18
Δ	-22.2%	-36.6	-47.1%	-32.1%

The results of the full AES-CTR encryption/decryption unit are summarized in Table V. The HLS-based approach requires up to 10% more area than the RTL-based approach in Altera devices, and noticeably less area (11% and 22%, respectively) in two Xilinx devices. These results indicate that for Xilinx devices, the HLS-based approach can produce more area-efficient top-level units with complex I/O interface.

On the other hand, the frequencies of the HLS-based designs are smaller than those for the RTL-based approach ranging from 8% for Cyclone IV to 37% for Virtex-7. The reason for the reduction in the frequencies of the HLS-based approach lies in the handshake signals. It seems that the handshake signals in the HLS-generated designs are not registered. After taking into account the differences in the formulas for throughput, the throughput of the cryptographic module using the HLS-based approach is less than the throughput of the RTL-based implementations by the amount ranging from 23.5% for Cyclone IV to 47% for Virtex-7. Similarly, the throughput to area ratio is between 28% and 42% lower for Cyclone IV and Stratix V.

V. CONCLUSION

This paper compares the results of designing cryptographic modules based on AES using HLS and RTL-based approaches in order to determine whether the HLS-based approach can produce results comparable to those obtained using handwritten RTL VHDL code. Our study has demonstrated that both approaches lead consistently to comparable results in terms of area and clock frequency. However, the small increase in the number of clock cycles between two consecutive inputs hinders the HLS-based approach in terms of throughput. It should be noted that this affect will be smaller for cryptographic algorithms with the larger number of rounds, and for low-area architectures (such as various horizontally and vertically folded architectures).

From our study, as long as the designer has a target architecture in mind, the HLS-based approach seems to be able to efficiently flatten and improve the designs in terms of area and frequency. The HLS-based approach can also produce I/O units capable of supporting complex communication protocols with reasonable performance. Overall, while the performance of HLS-generated designs tend to be still somewhat lower than the performance of handwritten RTL implementations, this approach can already lead to industrial-quality cryptographic modules. The differences in performance, while relatively high due to our choice of the algorithm and architecture, can be much lower in other scenarios. As the amount of time required to create a single cryptographic module using the HLS-based approach is generally significantly lower compared to the RTL-based approach, hardware designers can spend their time hand-optimizing other modules that may be more critical for the overall performance of the entire system.

References

[1] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 18–25, July 2009.

[2] C. Maxfield. (2012, Jul) First public access release to Xilinx Vivado design suite. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1317376

[3] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*, Oct 2011, pp. 1102–1105.

[4] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level Synthesis: Productivity, Performance, and Software Constraints," *Journal of Electrical and Computer Engineering*, vol. 2012, Article ID 649057, 14 pages, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1155/2012/649057

[5] F. Gruian and M. Westmijze, "VHDL vs. Bluespec System Verilog: A Case Study on a Java Embedded Architecture," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 1492–1497. [Online]. Available: http://doi.acm.org/10.1145/1363686.1364037

[6] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level Synthesis for FPGAs: From Prototyping to Deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, April 2011.

[7] Berkeley Design Technology, Inc. (2010) High-Level Synthesis Tools for Xilinx FPGAs. [Online]. Available: http://www.bdti.com/MyBDTI/pubs/Xilinx_hlstcp.pdf

[8] F. Burns, J. Murphy, D. Shang, A. Koelmans, and A. Yakorlev, "Dynamic global security-aware synthesis using SystemC," *Computers Digital Techniques, IET*, vol. 1, no. 4, pp. 405–413, July 2007.

[9] M. Ernst, S. Klupsch, O. Hauck, and S. Huss, "Rapid prototyping for hardware accelerated elliptic curve public-key cryptosystems," in *Rapid System Prototyping, 12th International Workshop on, 2001.*, 2001, pp. 24–29.

[10] S. Morioka, T. Isshiki, S. Obana, Y. Nakamura, and K. Sako, "Flexible architecture optimization and ASIC implementation of group signature algorithm using a customized HLS methodology," in *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, June 2011, pp. 57–62.

[11] S. Ahuja, S. Gurumani, C. Spackman, and S. Shukla, "Hardware Coprocessor Synthesis from an ANSI C Specification," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 58–67, July 2009.

[12] J. Davis, D. Buell, S. Devarkal, and G. Quan, "High-level synthesis for large bit-width multipliers on FPGAs: a case study," in *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on*, Sept 2005, pp. 213–218.

[13] National Institute of Standards and Technology, *FIPS PUB 197: Advanced Encryption Standard (AES)*, Nov 2001.

[14] ——, *SP PUB 800-38A: Recommendation for Block Cipher Modes of Operation*, Dec 2001.

[15] R. Housley, "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)," RFC 3686 (Proposed Standard), Internet Engineering Task Force, Jan. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3686.txt

[16] S. Shen, Y. Mao, and N. Murthy, "Using Advanced Encryption Standard Counter Mode (AES-CTR) with the Internet Key Exchange version 02 (IKEv2) Protocol," RFC 5930 (Informational), Internet Engineering Task Force, Jul. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5930.txt

[17] P. Barreto and V. Rijmen, "Reference code in ANSI C v2.2," Online, Mar. 2002. [Online]. Available: https://web.archive.org/web/20100329233031/http://www.ktana.eu/html/theRijndaelPage.htm

[18] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical C-based high-level synthesis." in *ISCAS*. IEEE, 2008, pp. 1192–1195. [Online]. Available: http://dblp.uni-trier.de/db/conf/iscas/iscas2008.html#HaraTHTI08

[19] K. Gaj, J. Kaps, V. Amirineni, M. Rogawski, E. Homsirikamol, and B. Brewster, "ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, Aug 2010, pp. 414–421.

[20] K. Gaj, E. Homsirikamol, and M. Rogawski, "Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, ser. Lecture Notes in Computer Science, S. Mangard and F.-X. Standaert, Eds. Springer Berlin Heidelberg, 2010, vol. 6225, pp. 264–278. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15031-9_18