PhD. Qualification Report

Can High-Level Synthesis Compete Against a Hand-Written Code in the Cryptographic Domain? A Case Study [1]

Authors:
Ekawat Homsirikamol and Kris Gaj

Ryan Silva
Boston University
Department of Electrical and Computer Engineering
rjsilva@bu.edu

1 Introduction

EVER-INCREASING computational complexity creates situations in which the demands of pure software solutions bog down a general-purpose CPU to the point of impracticality [2]. Hardware accelerators are mechanisms used to mitigate such situations by offloading specific computational tasks to hardware. Developing optimized hardware solutions to replace software comes with the trade-off of increased development time due to the introduction of hardware's inherent lower abstraction levels. This increased overhead can often be prohibitive, thus it has been a dream of hardware designers since the 1970s to create tools that reliably synthesize hardware using traditional software development techniques [3]. This dream tool would complete a process known as High Level Synthesis (HLS).

In order to consider the dream realized, logic synthesis must operate reliably using an abstract, algorithmic level description of a circuit as the primary input specification [4]. Unfortunately, due to reasons such as standardization [5], reliability [6] and portability [7], the two most ubiquitous languages for constructing hardware descriptions (VHDL and Verilog) do not operate natively at the algorithmic level [8]. This presents a problem: the most oft-used, best supported, languages for specifying synthesizable hardware do not behave like that of traditional compiled software. These two languages operate at a level of abstraction known as the Register-Transfer Level (RTL), which is one level of abstraction below that of the algorithmic level [9], and HLS seeks to bridge that gap.

The development of HLS tools is not the focus of Homsirikamol and Gaj's study, rather they seek to validate the state of the HLS dream by comparing the performance of hardware generated using an RTL circuit description to that of an algorithmic, HLS, specification. This report will explain the methodology used to conduct the case study and evaluate its conclusions based on alternative methods of comparison, ultimately presenting a more complete picture of whether HLS is a dream-come-true.

2 Background

HLS is the process of converting an algorithmic level description to RTL. Therefore, in this context the term synthesis can be defined as a method for traversing abstraction levels in hardware design. This is carried out by implementing a description found at higher levels of abstraction using only methods found in the lower [7]. Specifically, HLS translates descriptions found at the algorithmic level, often written in a relatively high-level language like C, to a functionally equivalent representation in an RTL language like VHDL or Verilog.

The Register Transfer Level of abstraction differs from that of the algorithmic in that RTL, as the name implies, specifies the movement of data between registers and the combinational processes that occur between. Unlike RTL, the algorithmic level of abstraction has no concept of a clock or specific delays, rather this abstraction level executes a sequence of instructions in order to accomplish computational tasks [7]. Software operating at the algorithmic level treats memory as a either a single, addressable repository of instructions and data (Von Neumann Architecture) or as two separate banks of information, one for storing instructions and one for data (Harvard Architecture). Some algorithmic-level software languages like C and C++ use pointers or address labels to interact with memory elements. Optimizing these sequential memory interactions presents significant challenges to the performance of HLS circuits, which will be elaborated upon in Section 7.1.

Measuring the quality of HLS tools is a problem that requires a high degree of specificity in order to be useful. Quality has often been measured by simply investigating whether a complex circuit synthesized through HLS can achieve a desired functionality [10] [11] [12] [13] [14]. Other studies [15] [16] seek to measure the quality of HLS by comparing the performance of an HLS circuit to its software implementation. The authors contend that neither of the above metrics for quality (functionality or software comparison) paint a sufficient picture of the state of HLS, rather they claim that the quality of HLS is better assessed when compared to a circuit synthesized through traditional RTL synthesis. Additionally, since computational challenges vary greatly based on functional domain, Homsirikamol and Gaj claim that comparisons between HLS tools and RTL synthesis tools must be domain-specific, i.e., it is not useful to make blanket statements of a tool's ability without, first, specifying the types of computational tasks each method is expected to complete. The selected paper chooses to make an HLS-to-RTL comparison in the cryptographic domain, specifically comparing implementations of the Advanced Encryption Standard (AES) in Counter Mode (AES-CTR).

3 The Advanced Encryption Standard in Counter Mode

This section defines the cryptographic transformations involved in AES and comments on challenges presented to the synthesizer during each stage of the encryption process. These challenges are framed in the context of the definition of synthesis from Section 2, which implies that the least difficult HLS-to-RTL transformations are those in which operations are identical in both the algorithmic and RTL levels of abstraction. Difficulty of synthesis increases as divergence of arithmetic methods between the algorithmic and RTL levels increase. Additionally, this section briefly explains what it means to operate AES in Counter Mode.

3.1 AES

AES is a symmetric key block cipher encryption algorithm that operates on 128-bit blocks of data using keys of length 128, 192 or 256 bits; however, the authors chose to restrict their test designs to implementations of the algorithm using only 128-bit keys [1]. The algorithm uses four main transformation functions on plaintext to provide cryptographic diffusion: SubBytes, ShiftRows, MixColumns and AddRoundKey. AES also transforms the key according to a function called Key Expansion.

3.2 Counter Mode

[17] describes recommendations for AES operating modes. AES in Counter Mode (AES-CTR) is defined by the following:

$$
\begin{array}{lll}
\text{CTR Encryption:} & O_j = CIPH_k(T_j) & \text{for } j = 1, 2 \ldots n; \\
& C_j = P_j \oplus O_j & \text{for } j = 1, 2 \ldots n-1; \\
& C_n^* = P_n^* \oplus MSB_u(O_n) & \\
\\
\text{CTR Decryption:} & O_j = CIPH_k(T_j) & \text{for } j = 1, 2 \ldots n; \\
& P_j = C_j \oplus O_j & \text{for } j = 1, 2 \ldots n-1; \\
& P_n^* = C_n^* \oplus MSB_u(O_n) &
\end{array}
$$

In the above definition, O_j is the output of the forward AES cipher using distinct counter vectors, T_j, as the plaintext on which the cipher operates. P_j is the plaintext and C_j is the ciphertext. P_n^* and C_n^* are the plaintext and ciphertext for the final block. The final block need not be a complete 128-bit block for the cipher to work properly in counter mode. This property is useful for processing streaming data where the input text size is not guaranteed to be a multiple of 128 [18] [19].

The most important attribute of AES-CTR in the context of hardware synthesis is that an implementation of the inverse cipher, $CIPH_k^{-1}(T_j)$, is not required.

3.2.1 ShiftRows

The ShiftRows transformation is defined by the following operation [20]:

$$s_{r,c}^{'} = s_{r,(c+shift(r,4)) \bmod 4} \text{ for } 0 < r < 4 \text{ and } 0 \leq c < 4 \tag{1}$$

$$\text{Where } shift(1,4) = 1;\ shift(2,4) = 2;\ shift(3,4) = 3 \tag{2}$$

The variable s in equation 1 is a 4x4 array of bytes called the *state array*. This equation effectively describes a byte-wise rotation operation performed on each row of the array. The bytes are rotated left according to the row number (e.g., 0 rotations for row 0, 1 rotation for row 1, etc.).

The ShiftRows transformation is a function that is trivial to synthesize as reordering bytes in an array is an operation that is identical in both the algorithmic and RTL levels of abstraction. [21].

3.2.2 SubBytes

SubBytes is more complex to synthesize than ShiftRows, as it requires Euclid's Extended Algorithm to calculate multiplicative inversions in a finite field, which is an operation not natively found in RTL descriptions. The algorithm is used to compute the polynomial $b^{-1}(x)$ such that Equation 3 holds true:

$$a(x) \cdot b(x) \bmod m(x) = 1 \tag{3}$$

Once the multiplicative inverse, labeled x below, is found the value undergoes an affine transformation according to the operation defined in equation 4.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{4}$$

While these calculations would be computationally demanding to execute on the fly [18], it is important to note that SubBytes is performed on individual bytes; therefore, these calculations can be carried out by pre-computing all possible values and then referencing a 256-byte lookup table called a Rijndael S-box [20]. The authors chose the S-Box method of implementing SubBytes in their HLS descriptions.

3.2.3 AddRoundKey

The AddRoundKey transformation is accomplished through a bitwise XOR operation between the state matrix and the expanded key from a Key Expansion [22]. Since bitwise XOR operations are identical in both the algorithmic and RTL levels of abstraction, HLS for the AddRoundKey transformation is a trivial task.

3.2.4 Key Expansion

Unlike the other four AES transformations, Key Expansion is performed on the cipher key rather than plaintext. Key Expansion references the Rijndael S-box from Section 3.2.2, but also requires iterative byte-wise rotations and XOR operations with a table of constants. Since the operations for byte-wise substitutions, XORs and rotations are identical in the algorithmic and RTL domains, HLS for Key Expansion is trivial.

3.2.5 MixColumns

MixColumns is accomplished by multiplying each four-byte column in the state array by a fixed polynomial, $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$, modulo $x^4 + 1$. This entire operation occurs within a Galois Field, $GF(2^8)$, which can be written as a matrix multiplication [20]:

$$s'(x) = a(x) \otimes s(x) : \begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \text{ for } 0 \leq c < 4 \quad (5)$$

Equation 5 finally describes a function whereby the methods of implementation could differ from an HLS description to that of an RTL.

4 Methodology

Synthesis tools vary greatly across technology and corporate barriers, therefore comparing implementations of synthesized designs becomes a delicate exercise. The authors present a case that their methodology for comparing synthesis performance is a fair one and this section explores that claim. Homsirikamol and Gaj limit their analysis to AES using a 128-bit key (AES-128). This restriction allowed them to remove from the reference ANSI C design all code supporting other key lengths [22]. This streamlined AES code is used as the baseline input to HLS and is referred to by the authors as HLSv0.

5 Results

5.1 HLSv0

5.2 HLSv1

Modified MixColumns: because only the encryption algorithm is used for AES-CTR, there are only two constants that need to be multiplied, 0x02 and 0x03. The authors are able to eliminate an entire 256-byte lookup table by implementing these multiplications using xtime.

5.3 HLSv2

6 Discussion

6.1 Abstraction

A design process cannot claim to operate at a level of abstraction higher than the lowest layer presented to the designer. The use of pragmas that describe reduced latency in terms of operations executed per clock cycle, by definition, lower the level of abstraction from algorithmic to RTL. Plus, the addition of architectural detail in a high-level language results in decreased speed (inferring more registers increases latency) and increased development time [23], all of which are the metrics HLS is supposed to dramatically improve.

6.2 Prior Findings

It takes guts to show up with results that contradict a study published in the same conference one year prior [2]..

6.3 RTL Code

The control for this case study, i.e. the RTL code in VHDL, was never specified beyond naming the utilized HDL. Important design decisions and modifications to the RTL descriptions, if any, were conspicuously missing from the document. For example, the authors never mention refining the VHDL as they refine the HLS code. They also never mention what implementation design decisions they made in implementing the algorithm in VHDL. Did they use an S-Box (look-up table) to accomplish SubBytes? Or did they calculate the affine transform? Parallelization? Timing constraints? For example, the official NSA RTL implementation of AES uses both an iterative and pipelined implementations [24]. The performance of the pipelined version of AES was significantly better than the iterative [25].

6.4 Parallelization

For example, the official NSA RTL implementation of AES uses both an iterative and pipelined implementations [24]. The performance of the pipelined version of AES was significantly better than the iterative [25], yet only the iterative RTL approach was considered in this paper. This is because automatic parallelization is a task proven to be NP-Hard due to pointer aliasing [26].

7 The State of HLS

Possibly lower development time, similar performance to **iterative** RTL designs (pipelined RTL with no optimization other than the addition of a pipeline AND implemented on 2000-era technology designs crush the highest HLS-based solution found in this study (MAX throughput of HLS = 4520 Mbps vs MAX throughput of pipelined RTL=5745Mbps [25]).

7.1 Is HLS the Future?

Since the HLS dream has yet to be realized, at least in the cryptographic [1] or matrix-multiplication [2] domains, what should hardware engineers expect from HLS in the future? The answer appears to be not much. Fundamental flaws in the HLS dream seem to erode expectations of HLS ever operating in the manner in which it was intended. Two main issues confronting HLS are performance and the ability of software descriptions to be synthesized.

The missing performance component holding back the dream of creating optimized hardware from a software language intended for sequential execution is parallelization. As shown in Section 6.3, parallelization can significantly improve the performance of a hardware design. The issue HLS is up against is the fact that software languages were originally designed to be performed sequentially on a general-purpose CPU, and thus the HLS C-based languages were not designed with parallelization in mind [27]. It would be nice if parallelization could be inferred automatically from software. Since C uses pointers to reference memory [28], the process to automatically extract parallel computations while guaranteeing that a memory location will not be asked to store two different values at the same time has been shown to be NP-Hard [26]. Even if pointers were carefully managed, or somehow avoided completely, the use of if statements and loops in software presents the same memory-management dilemma that has been shown to be at least undecidable and at wost uncomputable [29]. If parallelization cannot be inferred automatically, then it must be specified explicitly, thereby lowering the level of abstraction from the algorithmic level back to RTL. If not for raising levels of abstraction what, then, is the case for using HLS as opposed to RTL?

One might contend that raising design abstraction under some conditions is better than not raising it at all. In other words, why not use HLS to reduce development time and sacrifice the performance boosts associated with pipelining? This trade-off appears reasonable until it is put into practice. The promise of HLS, creating optimized hardware from a software description [15], seems to imply that one can write code targeting HLS the same way one can write code targeting traditional compilation. Unfortunately, not all language constructs are synthesizable. For example, C++ is supported by the Vivado HLS tool [30] and C++ (targeting compilation) supports object-oriented capabilities such as inheritance [31], but does that mean a hardware designer (targeting HLS) can use all properties of inheritance like virtual subclass

destructors? The answer is no [32]. This, then, creates subsets of software languages that are appropriate for synthesis. The issue then becomes knowing the difference between language constructs that are appropriate for synthesis versus those that are not. Currently, the solution is to define a standard for language constructs suitable for RTL synthesis [33]. This solution works provided the hardware engineer has deep knowledge of the standard as well as an understanding of the synthesizer's behavior. These requirements seem to defeat the purpose of the original HLS dream, i.e., creating optimized hardware using traditional software techniques [15] and serves to increase development time rather than reduce it.

7.2 Alternatives

Bluespec SystemVerilog [23]. Domain Specific Languages as an alternative to RTL

8 Conclusion

is difficult to imagine a computer engineer having to create a VLSI layout and fabricate an ASIC every time they wanted to run a C program. While there will always be a place for custom circuit design in the world of digital electronics, the basic tenants of digital design require that it remain very much the exception rather than the rule. Unfortunately for the experimentalist, this is not the case in the field of microfluidics. The creation of custom, "one-off", designs for individual microfluidic experiments, no matter how user-friendly the corresponding CAD software is, could be is what is keeping mLSI from more closely resembling that of its silicon counterpart in terms of productivity. Since Thorsen *et al.* successfully integrated thousands of micromechanical valves in 2002 [34], academic researchers have attempted to manage exponentially greater complexity in microfluidic design via the introduction of new design methodologies that attempt to introduce "top-down" specificity and move away from a "bottom-up" design philosophy [35] [36] [37] yet microfluidic experimentalists still find themselves in front of an oven baking a photoresist until it ceases to be sticky. If the goal is truly experimental automation the costly, in units time and overhead cost, fabrication step must be removed from the work flow for the majority case as it is for electronic computation. Often, academic papers delving into the relm of mLSI begin by presenting an analogy between microfluidic LSI and LSI found in digital electronics. This analogy seems strange as computer engineers are not required to know how to wash chemical from printed circuit boards (PCBs), use CAD tools to layout application specific integrated circuits (ASICs), nor enlist the aid of experts in fabrication who can in order to be productive. The *in silico* analogy does not hold when applied to the common-use case: that of the individual scientist or engineer. Why is that?

should be well versed in the art of fabricating devices would they ever hope to conduct a microfluidic experiment has served as a catalyst for academic research since . This paper will provide a historical overview of the emergence of digital design and highlight instances where microfluidics has deviated, resulting in the current design topology seen today. It will also analyse various attempts to rectify microfluidic design challenges using various computer-aided tools and the state of microfluidic design and computation alongside a historical analysis of the emergence of digital computation *in silico*. Important deviations will be highlighted and a new approach that better fits the *in silico* analogy is presented. digital electronics and highlight important deviations from . A natural first step in the production of viable microfluidic design rules that draw from those found *in silico* is the definition of specific layers of abstraction. These layers allow the engineer to properly design, build and test complex microfluidic designs at the least complex, or "highest", level of detail possible [38]. Analogies are often made between computation *in silico* versus via a microfluidic platform. At first glance, the analogy is sound in that computation *in silico* provides automation in computation by managing complexity through the introduction of abstraction layers. Abstraction works by placing the user at only the highest level relevant to the computation being performed and masking all underlying details. It can, therefore, be contended that functionally complete automation of microfluidic experiments implies placing the scientist at only the highest levels of abstraction and masking all underlying details. Currently, even the best efforts in microfluidic tools only remove intermediate levels of abstraction, while exposing the scientist to the highest **and lowest** levels. Imagine if the only output of a C program were a circuit schematic that must first be built in order to obtain the result of the program. presented its output in the form of a circuit

schematic that must top-down [35] [36]
abstraction layers [38] Furthermore, the successful execution of this research will present emergent capabilities as applied to the field of microfluidics.

8.1 Managing Complexity

Managing complexity is a necessary craft in that it allows the engineer to design complicated systems without becoming overwhelmed by details. The art of managing complexity in digital electronics design is a mature process relative to that found in microfluidics. This is evident by the existence of larger scales of integration *in silico*, such as VLSI, and by efforts to create tools that mirror the design-to-execution workflow found in electronic computing. Examples of such tools are Micado, for automation of control layer routing [39], and BioStream [38], which could serve as the cornerstone of true experimental automation and will be described in the subsequent section. It could serve as a useful exercise to review the methodology computer engineers use to manage complexity and then to contrast these principles with the current state of microfluidic design. There are few better place to find fundamental digital design practices than in an introductory textbook, which lists five key components involved with properly managing complexity: abstraction, discipline, hierarchy, modularity and regularity [8].

8.1.1 Abstraction

Abstraction can be defined as a method for hiding details when they are not important, often through the creation of different working levels of minutia. [8]. Figure 1 is an example of how electronic computing can be broken down into separate working levels of complexity.

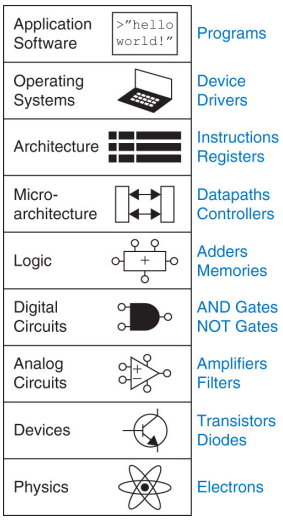

Fig. 1. Abstraction Layers for Electronic Computing (I'll have to make my own. . .)

The use of the term "working levels" of complexity implies that the person operating within that layer need not concern themselves with the details of a lower layer, as such requirements would ultimately defeat the purpose of abstraction. Lower levels of detail are said to be "abstracted" away when their use is considered automatic. However, designers of systems residing in one particular abstraction layer should have an understanding of how their design decisions affect the layers immediately above and below the working layer, such as a C programmer understanding the nature of an address space [8]. Theis *et al* advocate for the creation of abstraction layers in microfluidics similar to that found in electronic computing [38]. These layers achieve success by focusing on three basic fluidic operations: mixing, transport and storage. Their BioStream protocol is a brilliant step in decoupling microfluidic architecture from biological computation by providing a common language for describing an experimental protocol. BioStream served initially as a standard language for reporting biological protocols but expanded to an end-to-end system that effectively describes biological protocols within the BioStream Fluidic ISA and

executes them at the hardware level independent of microfluidic chip microarchitecture [38]. BioStream, however, does not fully address a functional purpose of abstraction, which is to provide automation, but it does accomplish a very important first step the authors describe as a "division of labor" between the biology and microfluidic experts.

One incredible benefit of BioStream is its ability to operate independently of a standard microfluidic architecture. Unfortunately, this independence could be holding back the biological experimentalist from true top-to-bottom automation. Why would the experimentalist purposely restrict their design decisions by adopting a standard architecture? The answer to this question is found within the second

8.2 The Productivity Gap

There is, and probably always will be, a place in digital electronics for PCB design and ASIC fabrication. However, before an engineer decides to begin the process of building a custom PCB or layout a new ASIC they should first consider how their deisgn decision addresses the productivity gap. In order to proceed, a working definition of the term "productivity" must be presented. Process and requirements engineers [40] have defined productivity strictly in terms of hours saved [41], as a function of on-time delivery [42] or as some measure of quality [43]. This paper will define the productivity of a particular method as the number of hours saved through the implementation of a particular process.

Device fabrication is not a task oft performed by a computer scientist. Rather, a computer scientist spends many hours debugging a program such that it runs reliably and correctly within the confines of a particular ISA. This exemplifies the nature of design discipline. It is well-within the realm of possibility for a computer engineer to give up debugging a program and reach for a CAD tool, which which to build a custom chip designed for their particular purposes. That scenario does not make sense because it creates an extremely large gap in productivity. The amount of lead-time required to design and build a PCB could significantly outweigh the benefits of having a single custom-chip to use only in very specific circumstances and only within that one engineer's lab. Why then is this practice deemed acceptable in microfluidics?

Even attempts to create some framework for flow-based microfluidic design, such as a common microfluidic ISA [39] or predefined software modules [44] are still, fundamentally, design methods for chip fabrication. Microfluidic chip fabrication is a highly unproductive in that it requires many hours to design and build a device incapable of performing diverse and repeatable experiments. Fortunately for the computer engineer there exists other prototyping options besides PCBs and ASICs, such as the use of a field programmable gate array (FPGA) or microcontroller. The microfluidic experimentalist is left with only one prototyping option that almost always requires some level of device fabrication.

REFERENCES

[1] E. Homsirikamol and K. Gaj, "Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, Dec 2014, pp. 1–8.

[2] S. Skalicky, C. Wood, M. Lukowiak, and M. Ryan, "High level synthesis: Where are we? a case study on matrix multiplication," in *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, Dec 2013, pp. 1–7.

[3] 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.

[4] M. C. McFarland, A. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301–318, Feb 1990.

[5] "IEEE standard for verilog register transfer level synthesis," *IEEE Std 1364.1-2002*, 2002.

[6] S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, and Y. Xie, "Reliability-centric high-level synthesis," in *Design, Automation and Test in Europe, 2005. Proceedings*, March 2005, pp. 1258–1263 Vol. 2.

[7] P. P. Chu, *RTL Hardware Design Using VHDL: Coding For Efficiency, Portability, and Scalability.* John Wiley & Sons, Inc., 2006, pp. 1–22. [Online]. Available: http://dx.doi.org/10.1002/0471786411.fmatter

[8] D. M. Harris and S. L. Harris, *Digital design and computer architecture*, second edition ed. Amsterdam, Boston: Morgan Kaufmann Publishers, Cop., 2013.

[9] F. Vahid, *Digital Design with RTL Design, Verilog and VHDL.* John Wiley & Sons, Inc., 2010, p. 247.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level Synthesis: Productivity, Performance, and Software Constraints," *JECE*, vol. 2012, pp. 1:1–1:1, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1155/2012/649057

[17] National Institute of Standards and Technology, "SP PUB 800-38A: Recommendation for Block Cipher Modes of Operation," *Special Publications*, vol. 800, no. 38A, Dec 2001.

[18] R. Housley, "Using Advanced Encryption Standard (AES) Counter Mode with IPsec Encapsulating Security Payload (ESP)," *Internet Engineering Task Force*, no. RFC3636 (Proposed Standard), Jan 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3686.txt

[19] S. Shen, Y. Mao, and N. Murthy, "Using Advanced Encryption Standard Counter Mode (AES-CTR) with the Internet Key Exchange version 02 (IKEv2) Protocol," *Internet Engineering Task Force*, no. RFC5930 (Informational), Jul 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5930.txt

[20] National Institute of Standards and Technology, "FIPS PUB 197: Advanced Encryption Standard (AES)," *Federal Information Processing Standards Publications*, vol. 197, Nov 2001.

[21] R. J. Silva, *Implementation and Optimization of the Advanced Encryption Standard Algorithm on an 8-Bit Field Programmable Gate Array Hardware Platform*. BiblioScholar, 2012.

[22] P. Barreto and V. Rijmen, "Reference code in ANSI C v2.2," http://www.ktana.eu/html/theRijndaeIPage.htm, Mar 2002.

[23] R. S. Nikhil and K. R. Czeck, "Bsv by example," 2010.

[24] N. S. Agency, "NSA's VHDL Implementations of the Five Advanced Encryption Standard (AES) Candidate Finalists," http://csrc.nist.gov/archive/aes/round2/r2anlsys.htm#NSA, Dec 1999.

[25] B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke, "Hardware performance simulations of round 2 advanced encryption standard algorithms." National Security Agency, Nov 2000.

[26] S. Horwitz, "Precise flow-insensitive may-alias analysis is np-hard," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 1, pp. 1–6, Jan. 1997. [Online]. Available: http://doi.acm.org/10.1145/239912.239913

[27] D. M. Ritchie, "The development of the c language," in *The Second ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL-II. New York, NY, USA: ACM, 1993, pp. 201–208. [Online]. Available: http://doi.acm.org/10.1145/154766.155580

[28] R. Reese, *Understanding and Using C pointers*. " O'Reilly Media, Inc.", 2013.

[29] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 323–337, 1992.

[30] T. Feist, "Vivado design suite," *White Paper*, 2012.

[31] Y.-F. R. Chen, E. R. Gansner, and E. Koutsofios, "A c++ data model supporting reachability analysis and dead code detection," in *Software EngineeringESEC/FSE'97*. Springer, 1997, pp. 414–431.

[32] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan, "Xilinx vivado high level synthesis: Case studies," in *Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014). 25th IET*. IET, 2013, pp. 352–356.

[33] "IEEE standard for verilog register transfer level synthesis," *IEEE Std 1364.1-2002*, 2002.

[34] T. Thorsen, S. J. Maerkl, and S. R. Quake, "Microfluidic large-scale integration," *Science*, vol. 298, no. 5593, pp. 580–584, 2002.

[35] W. H. Minhass, P. Pop, J. Madsen, and T.-Y. Ho, "Control synthesis for the flow-based microfluidic large-scale integration biochips," in *18th Asia and South Pacific Design Automation Conference (ASP-DAC 2013)*, 2013, pp. 205–212.

[36] J. Melin and S. R. Quake, "Microfluidic large-scale integration: the evolution of design rules for biological automation," *Annu. Rev. Biophys. Biomol. Struct.*, vol. 36, pp. 213–231, 2007.

[37] W. H. Minhass, P. Pop, J. Madsen, and F. S. Blaga, "Architectural synthesis of flow-based microfluidic large-scale integration biochips," in *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2012, pp. 181–190.

[38] W. Thies, J. P. Urbanski, T. Thorsen, and S. Amarasinghe, "Abstraction layers for scalable microfluidic biocomputing," *Natural Computing*, vol. 7, no. 2, pp. 255–275, 2008.

[39] N. Amin, W. Thies, and S. Amarasinghe, "Computer-aided design for microfluidic chips based on multilayer soft lithography," in *IEEE International Conference on Computer Design, 2009. ICCD 2009.* IEEE, 2009, pp. 2–9.

[40] D. Damian and J. Chisan, "An empirical study of the complex relationships between requirements engineering processes and other processes that lead to payoffs in productivity, quality, and risk management," *Software Engineering, IEEE Transactions on*, vol. 32, no. 7, pp. 433–453, July 2006.

[41] S. Lauesen and O. Vinter, "Preventing requirement defects: An experiment in process improvement." *Requir. Eng.*, vol. 6, no. 1, pp. 37–50, 2001. [Online]. Available: http://dblp.uni-trier.de/db/journals/re/re6.html#LauesenV01

[42] H. Wohlwend and S. Rosenbaum, "Software improvements in an international company," in *Proceedings of the 15th International Conference on Software Engineering*, ser. ICSE '93. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993, pp. 212–220. [Online]. Available: http://dl.acm.org/citation.cfm?id=257572.257621

[43] J. D. Herbsleb and D. R. Goldenson, "A systematic survey of cmm experience and results," in *Proceedings of the 18th International Conference on Software Engineering*, ser. ICSE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 323–330. [Online]. Available: http://dl.acm.org/citation.cfm?id=227726.227791

[44] A. K. Soe, M. Fielding, and S. Nahavandi, "Lab-on-a-chip turns soft: Computer-aided, software-enabled microfluidics design," in *Advances in Social Networks Analysis and Mining (ASONAM), 2013 IEEE/ACM International Conference on*. IEEE, 2013, pp. 968–971.

Ryan Silva Ryan Silva is a Captain on active-duty in the United States Air Force. He is currently assigned to Boston University, where is pursuing his PhD in Computer Engineering. After graduating with his degree, Ryan will be assigned to an Air Force unit before returning to teach at the United States Air Force Academy, where he graduated with a degree in Electrical Engineering in 2005.