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

We find ourselves caught in the snare of ever-increasing computational complexity, where the demands of modern computation can 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 synthesize optimized 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, HLS must create optimized hardware 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. Instead, 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]. 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 first explain the methodology used to conduct the case study and present its findings. Next, an evaluation of their conclusions will be presented alongside alternative methods of comparison. This report concludes with an analysis of what to expect from HLS in the future, ultimately presenting a more complete picture of whether HLS is, or ever will be, a dream-come-true.

2 BACKGROUND

2.1 HLS

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 to a functionally equivalent representation in an RTL language like VHDL or Verilog. These algorithmic level descriptions are often written in a relatively high-level language, in this study that language is C,

2.2 The Advanced Encryption Standard (AES) in Counter Mode

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. 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.

Recommendations for AES operating modes are described in [10]. AES in Counter Mode (AES-CTR) is defined by the following:

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

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 [11] [12]. The most important attribute of AES-CTR in the context of this study is that an implementation of the inverse cipher, $CIPH_k^{-1}(T_j)$, is not required.

3 Summary of Selected Paper

3.1 Problem

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 [13] [14] [15] [16] [17]. Other studies [18] [19] 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 quality metrics (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 synthesize. The selected paper chooses to make an HLS-to-RTL comparison in the cryptographic domain, specifically comparing implementations of the Advanced Encryption Standard.

3.2 Proposal

Ultimately, the authors wish to demonstrate the quality of HLS by comparing the performance of circuits synthesized using a software description to circuits synthesized using an RTL description. Metrics used in the study to define performance included: area, frequency, throughput and efficiency (throughput per unit area). In order to make a fair comparison, the authors need to make a case that the synthesis process alone was responsible for performance changes. This was accomplished by conducting their evaluations across multiple chip manufacturers and device families, thus removing the target technology as a possible source of performance variability. The RTL language, FPGA tools and synthesis options were all held constant, leaving HLS as the claimed independent variable in the case study.

Using the above methodology, the authors used the AES reference design in ANSI C [20] as a starting point for creating the study's HLS specifications. They then created three different implementations of AES, each optimized in a different fashion. The first, referred to as HLSv0, was streamlined by eliminating unnecessary code. The HLSv0 code was then optimized by modifying the mathematical methods used to implement the Key Expansion and MixColumns transformations; this design is labeled HLSv1. Finally, HLS synthesizer directives, called *pragmas*, were added to the HLSv1 design, thus creating the last design referred to as HLSv2. The performance of each HLS design was measured against the baseline RTL design.

Three wrappers were then placed around the core AES implementations, thereby creating three distinct cryptographic systems; the performance of these systems underwent the performance analysis defined above. The three cryptographic systems included a twice-unrolled architecture of AES, whereby the AES core was essentially duplicated, an implementation of AES in counter mode (AES-CTR) and finally, an implementation of AES-CTR using an I/O-multiplexing communication protocol modified from a previous study [21].

3.3 Evaluation

Four performance analyses were presented. These compared RTL-to-HLS implementations for each of the AES core modules (RTL, HLSv0, HLSv1, HLSv2), the twice unrolled architecture, the AES-CTR system and the AES-CTR system using an I/O-multiplexed communication protocol.

The report found that, while HLS generated comparable results to RTL based on area and frequency, RTL designs consistently outperformed their HLS counterparts in throughput and efficiency across all design architectures, chip manufacturers and device families by an average of 24.8%; the sole exception being the AES-CTR system implemented on the high-end Xilinx platform. The authors call the AES-CTR anomaly the "only major surprise" from their study but note that, while the frequency of the HLS design surpassed that of RTL by 25.9%, the throughput of the HLS design was only 4.9% better than the RTL implementation. Furthermore, these gains were made at the cost of a significant increase in area: 20.8% larger than the RTL design. The magnitude of this trade-off becomes apparent when the efficiency of this design is considered, which shows that the RTL design is superior in throughput per unit area by 13.2% for that particular design.

Homsirikamol and Gaj attribute the difference in throughput and efficiency results to the inability of HLS to infer an optimal data controller. While the RTL design used in the study overlaps the calculation of the final AES round with the computation of the first round key for the next block of data, the controller inferred by HLS actually requires an extra clock cycle to register the ciphertext to the output port. The equations for RTL throughput (Equation 1) and HLS throughput (Equation 2) show a difference of two clock cycles between the two implementations. Since the cryptographic domain involves highly iterative calculations, it is shown that these two clock cycles are responsible for the significant performance drop in throughput and efficiency seen by HLS designs. The authors then concede that attempts to optimize the controller inferred by HLS would be "very difficult to accomplish."

$$\text{RTL Throughput} = \frac{128 * f_{CLK}}{(\text{Latency} - 1)} \tag{1}$$

$$\text{HLS Throughput} = \frac{128 * f_{CLK}}{(\text{Latency} + 1)} \tag{2}$$

3.4 Conclusion

The authors conclude that HLS can produce circuits occupying a similar amount of resources to those generated by RTL synthesis and can also achieve comparable clock frequencies. They acknowledge the inability of HLS to fully optimize the throughput of its designs, but that the difference is only a "small increase in the number of clock cycles." Furthermore, they claim that this small increase can be easily mitigated by using ciphers that maximize the amount of time between inputs.

The study concludes with the claim that HLS could be used for designs that have a specific architecture (for defining pragmas) and that the designer wants to "flatten and improve the designs in terms of area and clock frequency." They cite the ability to quickly scale their cryptographic cores into cryptographic systems using HLS while

achieving similar results to that of RTL synthesis and that the time saved in development can ultimately be used to optimize critical modules.

4 Analysis of Selected Paper

4.1 Paper Strengths

4.1.1 Carefully Crafted Caveats

The authors were careful to avoid broad claims regarding the state of HLS. They are mindful to articulate that their findings are limited to designing "cryptographic modules based on AES." Their conclusions contain similar caveats: that HLS is comparable to RTL synthesis specifically targeting "low-area architectures", cryptographic algorithms with a larger number of rounds, and that HLS may ultimately be an efficient design solution if "the designer has a target architecture in mind" and wishes to "improve the designs in terms of area and frequency."

4.1.2 High Bar for Comparison

HLS is described by the authors as "a next step in the development of modern hardware design methodologies." This implies that the ultimate goal of HLS is to replace RTL synthesis as the primary means of designing hardware, which seems to be a widely-held opinion in literature [3] [18] [19]. For HLS to replace RTL synthesis, it must perform at a comparable level, yet the focus of many studies investigating the state of HLS shy away from this comparison and directs their efforts, instead, on comparing the performance of hardware generated by HLS to that of the code's corresponding software implementation [18] [19] [2]. By comparing the performance of HLS to that of RTL synthesis, Homsirikamol and Gaj elected to boldly investigate the state of HLS as it applies to its ultimate goal.

4.1.3 Controlling External Influences

Studies claiming to investigate the state of HLS must present convincing evidence that performance improvements or degradations were caused by HLS and not another external influence. Since synthesis tools vary greatly across manufacturers and device families, 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.

Common experimental oversights include failing to approach the target technology as a variable that needs to be controlled, as seen in [2]. Also, some studies do not control proprietary FPGA resources as in [15]. Homsirikamol and Gaj were careful to design controls for the former of the above two experimental oversights by synthesizing their designs using multiple chip manufacturers (Xilinx and Altera) and across multiple chip families (high-end and low-end family from each manufacturer). They addressed the latter oversight by instituting strict directives to the synthesis tools forbidding the use of dedicated FPGA resources, although these directives were ignored by the synthesizer on two occasions (HLS inferred a BRAM when synthesizing the HLSv0 and HLSv1 designs).

4.2 Paper Weaknesses

4.2.1 Clock Frequency as a Metric

Maximizing clock frequency is not a useful metric in isolation. Rather it is useful as a component of calculating throughput. One may argue that the block size multiplier in Equations 1 and 2 dictates that increasing the clock frequency is the most lucrative venture in terms of increasing a design's throughput. While the math does suggest this to be the case, the results of the study indicate that increasing the clock frequency using HLS comes at such a severe cost in latency that it overwhelms the coefficient in the numerator.

4.2.2 RTL Optimization

While the authors set a high bar for comparison, as articulated in Section 4.1.2, they appear to limit their efforts to optimize the study's RTL designs. The authors create three different designs for HLS, each optimizing a different aspect of the C code. However, the RTL code was never specified beyond naming the utilized HDL and presenting a generalized block diagram outlining the design's architecture. Important design decisions and modifications to the RTL descriptions, if any, were conspicuously missing from the document. For example, did the RTL design use an S-Box (look-up table) to accomplish SubBytes or did it calculate the affine transform on the fly? Did the RTL design use the *xtime()* function, a log/antilog table or something else to calculate MixColumns?

In the presence of this ambiguity, it can be assumed that no serious efforts were made to optimize the RTL designs. This is troubling for the HLS cause as the performance of the (assumed) unoptimized RTL designs still outperforms that of the optimized algorithmic level descriptions in all metrics, save the AES-CTR implementation described in Section 3.3.

4.2.3 Parallelization

The most glaring shortcoming of this study is eluded to in Section 4.2.2 and that is the failure to address the issue of HLS and inferred parallelization, which will be elaborated upon in detail in Section 4.4. The National Security Agency (NSA) published official RTL implementations of AES, which includes both an iterative and pipelined version [22]. The throughput of the pipelined version was superior to the iterative version by a staggering margin (605.77 Mbps compared to 5745.06 Mbps, or +848%) [23].

Equation 1 infers that the RTL design used in Homsirikamol and Gaj's study already parallelizes the computation of the subsequent data block's first round key (*round_key(0)* of *block(i+1)*) with the computation of the last AES round for the current block(*block(i)*). Given the performance benefits of parallelization demonstrated in the NSA study, one is left to wonder why the authors ceased their parallelization efforts at this point. One possible answer to the question is that given the difficulty of HLS to infer parallelism, as outlined in Section 4.4, a performance analysis pitting a pipelined RTL design with an optimized HLS description would not be fair in terms of throughput. This is based on the fact that a pipelined RTL design implemented on 2000-era technology using 2000-era tools still managed to outperform the best HLS-based solution in this study (4520 Mbps compared to 5745.06 Mbps [23], or +27.1%).

While it may be true that a fair comparison cannot be made between HLS designs and those of parallelized RTL in terms of throughput it would still be interesting to investigate how the two compare in terms of efficiency, as the parallelized RTL design in [23] is over 568% larger in transistor count than the iterative design.

4.3 Critique

The working level of abstraction is defined as the lowest level presented to the designer [8], therefore a design process should not claim to operate at a level of abstraction higher than the lowest layer presented to the designer. HLS is defined as hardware synthesis at the algorithmic level of abstraction [18] [19]. It can be argued that Homsirikamol and Gaj achieved their largest performance gains through the use of pragmas. Does the use of pragmas present the designer with a level of abstraction lower than the algorithmic, thereby contradicting the definition of HLS? In order to find out, a working definition of the RTL and algorithmic abstraction level must be presented.

As an abstraction level, RTL differs from that of the algorithmic level in that RTL, as the name implies, specifies the movement of bits 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]. Furthermore, **architecture is implicit for HLS designs** [24], which stands in stark contrast with the authors' conclusion that HLS is efficient in some aspects "so long as the designer has a target architecture in mind."

HLS claims to abstract away architecture by registering data in a similar fashion to that of software [25]. 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 [26]. Optimizing these sequential memory interactions present significant challenges to the performance of HLS circuits, which will be elaborated upon in Section 4.4.

The use of pragmas that focus on reducing latency by explicitly defining operations executed per clock cycle, such as **INLINE** and **ARRAY_RESHAPE**, by definition, lower the level of abstraction from algorithmic to RTL. The use of pragmas were effective in reducing latency but they also lowered the working level of abstraction from the algorithmic level to RTL. Therefore, it could be argued the study did not, in fact, perform an HLS-to-RTL comparison as it claims; rather, the study actually compares two circuits synthesized using RTL descriptions.

4.4 Future Work

Since the HLS dream has yet to be fully realized, at least in the cryptographic domain, 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.

The missing performance component holding back the dream of creating optimized hardware from a software language intended for sequential execution is automatic

parallelization, which has been described as the "holy grail" of parallel computing [27]. As shown in Section 4.2.3, 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-supported C-based languages were not designed with parallelization in mind [28]. Plus, the addition of architectural detail in a high-level language can negatively impact throughput (i.e., inferring more registers increases latency) and mitigating this effect increases development time [24], all of which are the metrics HLS is supposed to improve.

It would be useful to develop an automated process that teases out implicit parallelization from this sequential software description. Unfortunately, since C uses pointers to reference memory [26], the complexity involved with automatically extracting parallelism 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 [29]. Even if pointers were carefully managed, or somehow avoided completely, the use of dynamic storage and recursive data structures in software presents the same memory-management dilemma that has been shown to be at least undecidable and at wost uncomputable [30]. 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, as Homsirikamol and Gaj elude to in their conclusions, 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 original promise of HLS, i.e., creating optimized hardware from a software description [18], 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 [31] and C++ (targeting compilation) supports object-oriented capabilities such as inheritance [32], but does that mean a hardware designer (targeting HLS) can use all properties of inheritance like virtual subclass destructors? The answer is no [33]. 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 that are suitable for RTL synthesis [34], thus separating the synthesizable subset from the rest of the language. This solution is valid provided that 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 and serves to increase development time rather than reduce it.

5 Conclusion

Homsirikamol and Gaj ultimately conclude that HLS can generate results comparable to RTL while significantly lowering development time. They further claim that the time saved using HLS can be used by the designer to hand-optimize problematic or critical modules. For the authors' claim maintain consistency, HLS should be able to achieve

good results while being utilized in the manner for which it was intended: designing hardware at a higher level of abstraction. Their claim hinges so precariously on higher abstraction because this is the only stated means of reducing development time. As shown in Section 4.2, the authors seem to contradict this intent through the use of pragmas that effectively lower the level of abstraction.

Based on the bleak future for HLS presented in Section 4.4, it can be argued that HLS is, at best, a zero-sum game and, at worst, a costly distraction.

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] 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.

[11] 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

[12] 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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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

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

[21] 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
[22] 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.
[23] B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke, "Hardware performance simulations of round 2 advanced encryption standard algorithms." National Security Agency, Nov 2000.
[24] R. S. Nikhil and K. R. Czeck, "Bsv by example," 2010.
[25] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 33–36.
[26] R. Reese, *Understanding and Using C pointers*. " O'Reilly Media, Inc.", 2013.
[27] J. P. Shen and M. H. Lipasti, *Modern processor design : fundamentals of superscalar processors*. Boston: McGraw-Hill Higher Education, 2005, index. [Online]. Available: http://opac.inria.fr/record=b1129703
[28] 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
[29] 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
[30] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 323–337, 1992.
[31] T. Feist, "Vivado design suite," *White Paper*, 2012.
[32] 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.
[33] 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.
[34] "IEEE standard for verilog register transfer level synthesis," *IEEE Std 1364.1-2002*, 2002.

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.