Efficient Verification of an Elaboration-Time, Key Size Configurable, Pipelined AES Encoder and Decoder using a Mentor Veloce Emulator

Daniel E. Collins, *Student, PSU* and Alex J. Pearson, *Student, PSU*

***Abstract* — This paper presents follow-on work done to efficiently verify an AES Encoder and Decoder using Mentor Veloce emulation technology. We summarize the previous work done as part of the PSU class ECE 571. We show the major contributions implemented in the design and top-level testbench in order to increase the emulation performance. We present our results and demonstrate the speedup achieved by using the Veloce emulator. Finally, we explore our solutions to challenges encountered and discuss their impact on the design and emulation performance.**

***Index Terms*—Mentor Graphics, Veloce, Emulation, Questa, Simulation, Cryptography, AES, SystemVerilog**

I. Introduction

OUR starting point for the competition was a compile-time, key size configurable, pipelined AES encoder and decoder developed by Dan Collins, Alex Pearson, and Scott Lawson (unable to participate in competition) for the Portland State University class ECE 571: Intro to SystemVerilog.

The Advanced Encryption Standard (AES) is a symmetric-key block cipher specification established by the United States National Institute of Standards (NIST) and detailed in the FIPS 197 standard [2]. It is widely adopted throughout the security industry.

AES operates on a 128-bit block of input data and combines it with an input key. The standard supports three different key sizes: 128, 192, and 256 bits. Each larger key size corresponds to a higher level of cryptographic strength by increasing the rounds of state transformation applied to the input clear text: 10, 12, and 14 rounds respectively. Our AES engine supports all three key sizes.

The key is expanded into per-round keys that are combined with the current state (the data being operated on that round) in an exclusive-or operation. In addition to the process of adding the round key to the state there are three additional steps in each round: substitute-bytes is a non-linear substitution (commonly implemented with a lookup table), shift-rows cyclically shifts rows in the state, and mix-columns where the state is multiplied by a fixed, linear transformation [1].

Submitted November 18th, 2016. This work was done as an entry for the Mentor Graphics Need for Speed competition.

Daniel E. Collins is with Intel Corporation, Hillsboro, OR 97124 USA (email: Daniel.E.Collins@intel.com).

Alex J. Pearson is with Mentor Graphics, Wilsonville, OR 97070 USA (email: Alex_Pearson@mentor.com).

We implemented each of the different steps in each round as a module. We then implemented a round module and a pipeline buffered version of the round. Our top level cipher instantiates the appropriate number of buffered rounds and the key expansion module. The key expansion operation responsible for generating the round keys is implemented as a separate module. In this manner the three different key size ciphers have a pipeline depth equal to the number of rounds. We wrote testbenches that use known answer test vectors to verify the operation of each of the sub-modules prior to our randomized top-level module testing. Our initial project and verification solution suffered from a couple of issues:

1) Our design was compilation-time configurable; we could only instantiate an encoder and decoder for 1 of the 3 key sizes at a time, meaning we also could only test one key size at a time.
2) Our top-level verification testbench (consisting of an HVL testbench and HDL transactor) simulated using Questa and emulated successfully on the Veloce emulator, but our Veloce throughput was <1% and our simulation and emulation times were similar. HDL time advance is a measure of how efficiently the emulator is being used. It is a ratio of the total amount of emulator clock cycles that were spent moving the design's clock forward over the total emulator clock cycles run during the emulation time. A low HDL time advance percentage indicates that we are not effectively utilizing the emulator. For our initial implementation, simulating or emulating 10 million test vectors through each of the different key size compiled encoders or decoders required approximately 2 hours of run time.

II. Major Project Contributions

For the competition, our primary focus was on removing performance bottlenecks in our emulation strategy and design with the goal of realizing the gains that emulation is capable of delivering.

The performance issues of the original effort fell broadly into three categories: inefficient communication of test data with the emulator, serialized testing of modules, and suboptimal implementation of the portions of the base design. More simply put, 1) transferring large data sets to and from the emulator 2) how the data was used once it got there and 3) emulator resource constraints due to design choices.

We were able to determine that there was communication bottleneck between the HVL testbench and the Veloce emulator by analyzing the Veloce runtime report and the Questa sample-based call tree report to figure out where the majority of the runtime was being spent. We found that our initial low throughput was caused by frequently pausing the advance of the design in order to transfer additional data. With the assistance of Jeff Evans, Sr. Technical Staff Member at Mentor Graphics, we were able to identify a read port bottleneck that was making our design not fit on the emulator by evaluating the compile_velsyn_0.log produced during the compile process. To overcome this limitation we added the ability to compile our project with and without inferred RAMs. In our design, inferring RAMs instead of synthesizing logic elements for our lookup tables causes a large number of read ports to be needed which can't fit on the available emulator. We will explore the effect of inferring RAMs on emulator performance in our Results section.

We chose to focus on the first two categories identified above, as successfully emulating a design that is beyond your control is a situation real emulation engineers find themselves in. Our major contributions were the following:

1) Modify the design to be elaboration-time configurable so that we could instantiate multiple encoders/decoders of different key sizes in parallel.
2) Modify our verification strategy to utilize assertions for checking the validity of our output instead of transferring data from the emulator to the HVL testbench for verification. This eliminated all data transfers out of the emulator.
3) Modify our test vectors, HVL testbench, and HDL transactor to test all 3 key size encoders and decoders in parallel.
4) Create a 2-phase testing strategy:
    a) In the first phase we directly validate the encrypted text output of the encoders and the plain text output of the decoders.
    b) In the second phase we only validate that the output of the encoder can be successfully decoded by the decoder without directly validating the encrypted output of the encoder. We also instantiate an additional set of encoders and decoders where we check that the decoded output can be successfully encoded without directly checking the plain text output of the decoder.
5) Generate a large number of test variations in the HDL transactor by sending a "seed" value and then permuting that value in the transactor to create many unique test cases to allow for multiple clock cycles of execution without needing to transfer a new set of values from the HVL testbench.
6) Improve the efficiency of communication of test vectors by compacting the format of data delivered from the HVL testbench to the HDL transactor.

As part of our design improvement effort we also investigated pipelining the key expansion module. We were able to determine that this module represented the logical critical path for the design by analyzing reports generated by the Veloce compilation process. Pipelining the key expansion module is practical in the encoder due to the order that the expanded round keys are consumed but impractical in the decoder due to the round keys having dependencies on the previous keys and needed in reverse order. We successfully implemented a pipelined version of the key expansion module but didn't include it in our final design because of the asymmetries that it introduced as we were only able to incorporate the pipelined version into the encoder.

III. Block Diagrams

A. Design

Figure 1: AES Encoder Block Diagram details the internal design of the AES encoder. At a high level, the encoder follows the easily sub-divisible steps detailed in FIPS 197 [2]. Our engine takes in two inputs apart from clock and reset: a 128-bit (block size aligned) input data, and a key of variable size, depending on the variant (AES128, AES192, or AES256). It outputs a ciphertext of block size. Of note are the SBox and Galois Field Multiplication lookup tables, and the saturating counter. The lookup tables are used as part of the encryption process, while the counter produces a valid signal to prevent consumers from sampling data before the pipeline is primed [3].

Figure 2: AES Decoder Block Diagram shows the inverse cipher, or decoder of the AES engine. Apart from minor tweaks to the final round as specified in FIPS 197, the data path is familiar. The Galois Field lookup tables are for different scalars than the encryption process, but function similarly.

B. Top-Level Testbench

The top-level testbench is laid out in Figure 3: Top Level Testbench (Directed Test). The original scheme, with the limitations discussed above, transported the encoder and decoder outputs back up to the HVL testbench through SCEMI pipes. An optimization of this project was to eliminate the return data path by using assertions on the emulator; previously we needed to transmit back a 128-bit block per clock cycle for each of the different encoder or decoder instantiations in order to verify the output correctness in the HVL testbench. Because of this optimization we only have to send data from and never back to the host, which alleviated a great portion of the communication overhead plaguing the original design.

We also investigated other methods of communication between the HVL and Transactor portions of the top-level testbench including SystemVerilog virtual interfaces but were unable to achieve as significant of a speedup as we did using SCEMI pipes. Other methods of transfering data back to the

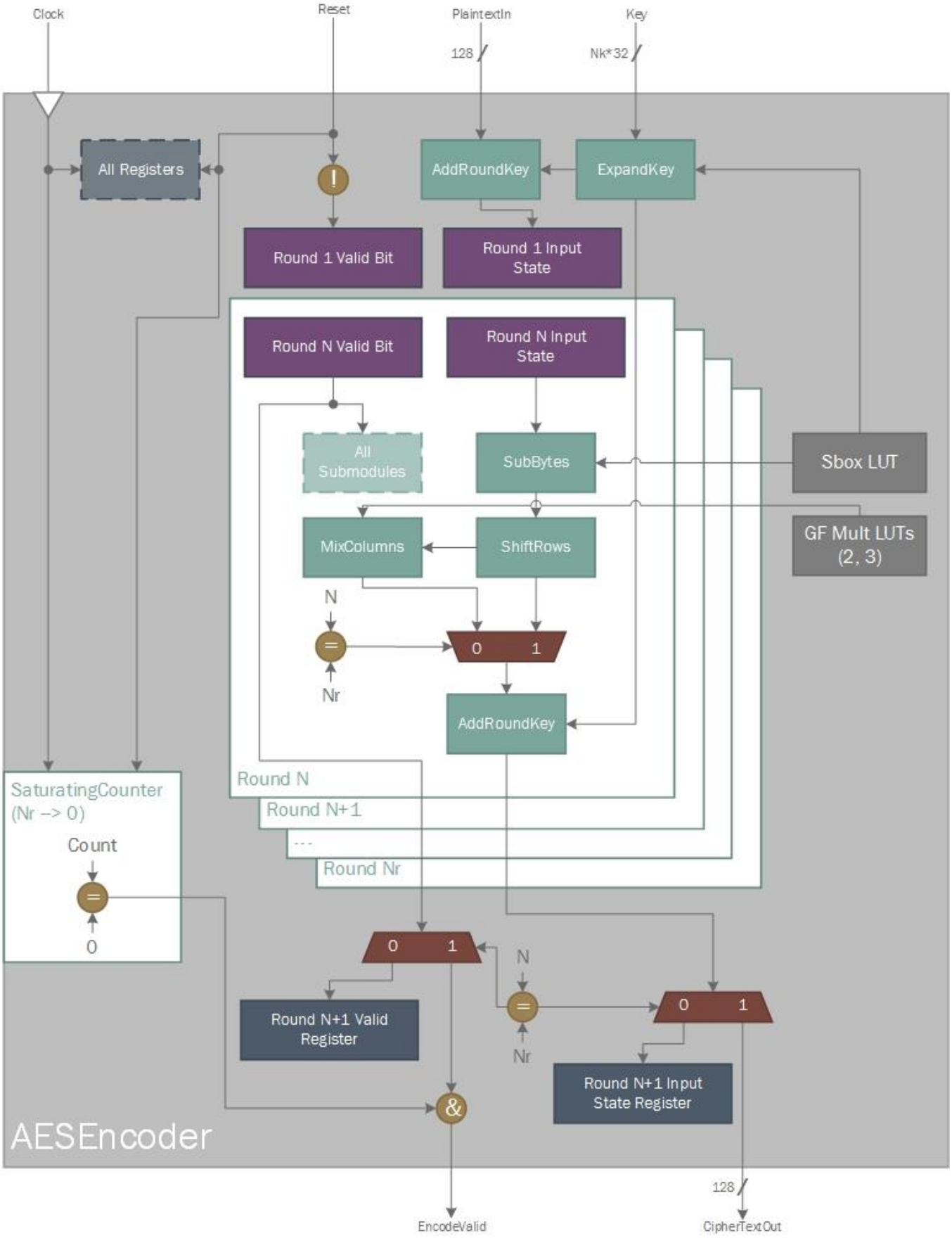

Figure 1: AES Encoder Block Diagram

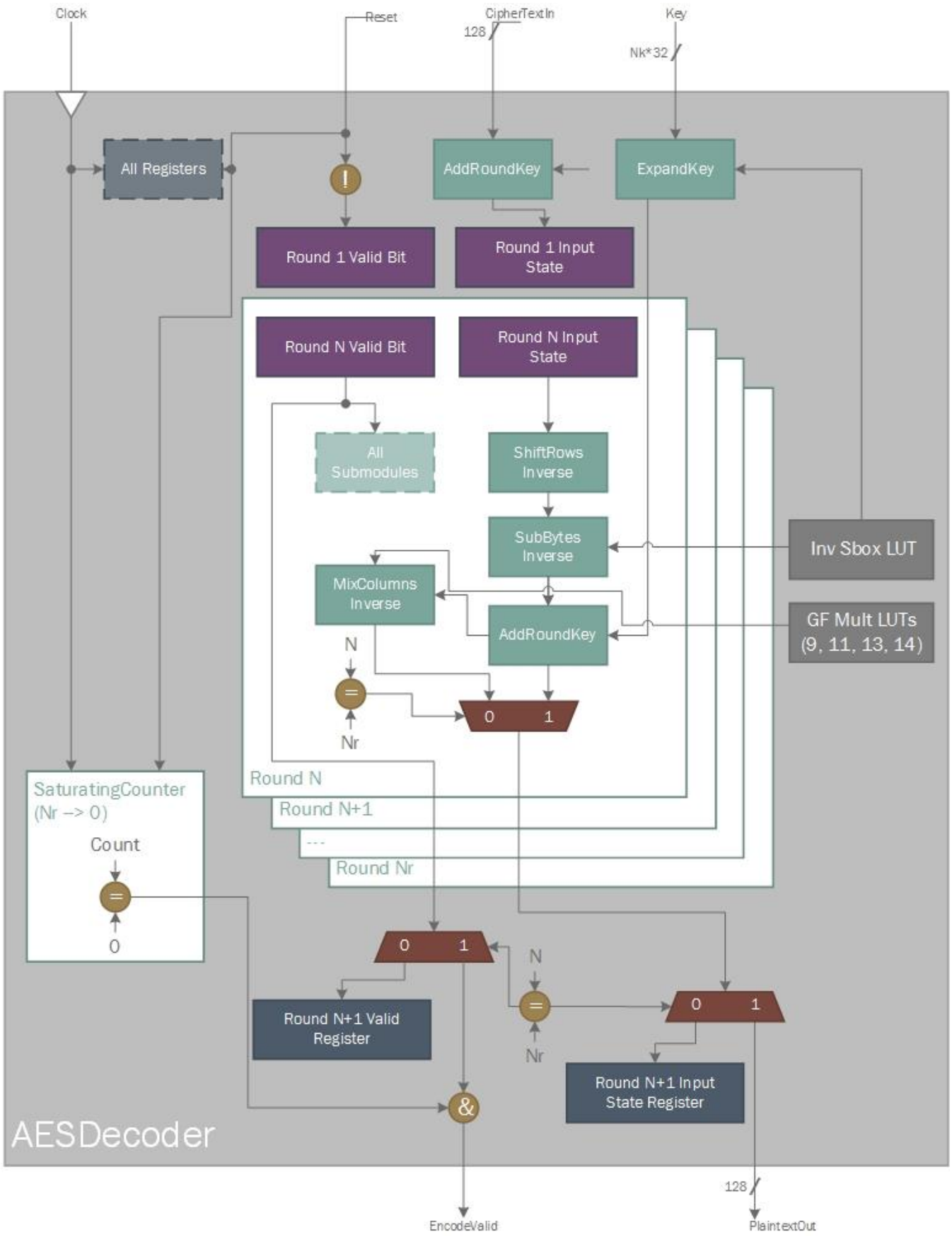

Figure 2: AES Decoder Block Diagram

HVL portion of the testbench would allow for additional modeling flexibility but exploring these methods was outside of the scope of this project.

In our directed test model, we send down a plaintext block with a 256-bit key, along with an expected ciphertext for each key size of AES. Using substrings of the key up to the appropriate key size, we feed the same plaintext into parallel instantiations of each variant of AES in the encode direction and feed the different expected values in the decode directions. We check the output of each encoder module with assertions using expected data shipped down in the original packet, or in the case of the decoders we compare the output to the plaintext data. This approach verifies each direction of each variant is working correctly, since our test vectors were randomly pre-generated and our expected answers were also pre-generated using a battle-tested C library, LibTomCrypt.

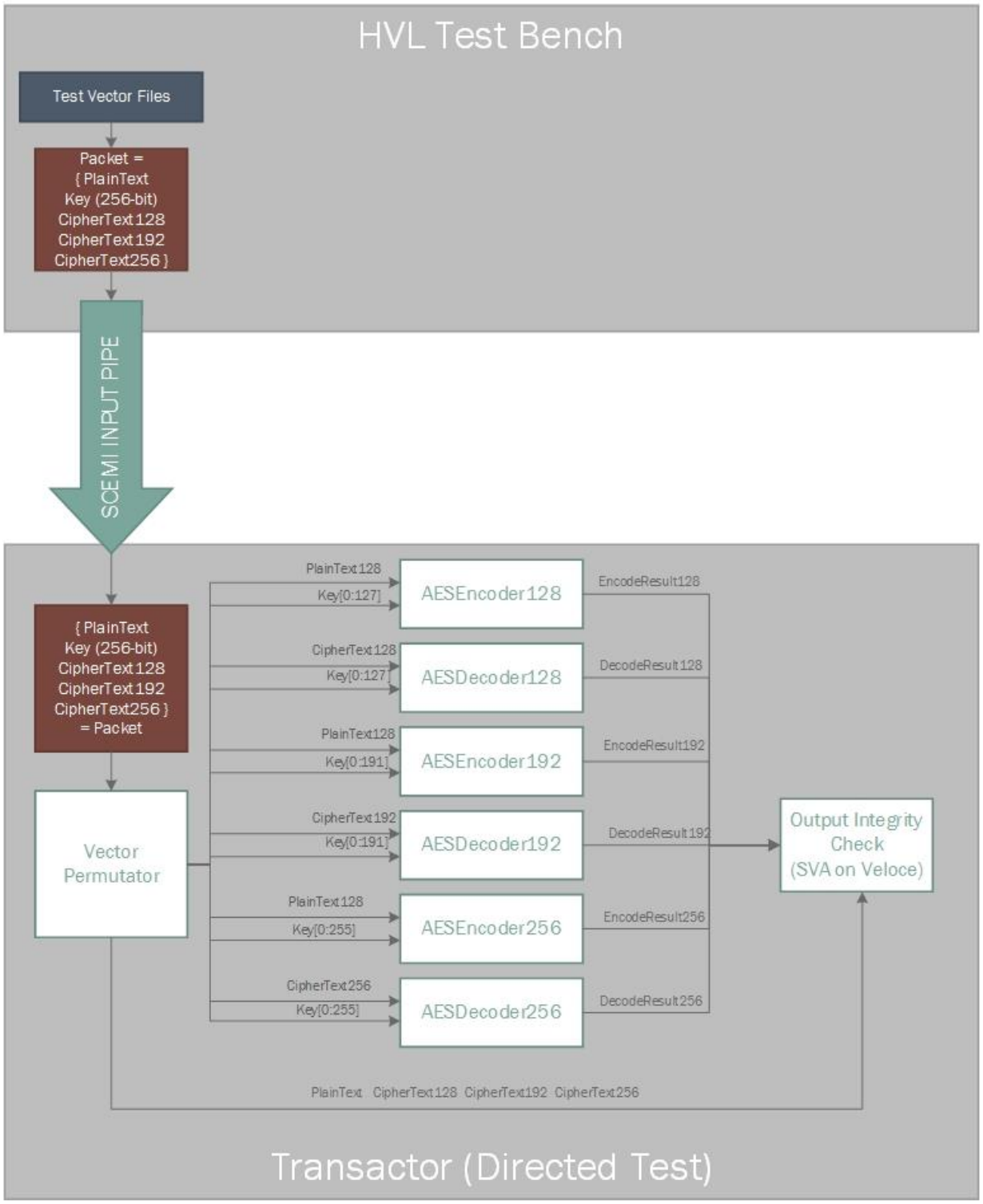

Figure 3: Top Level Testbench (Directed Test)

Figure 4: Top Level Testbench (Seeded Test) shows our more efficient scheme that we call seeded testing. The encoders and decoders for each key size are chained; we only check that the original transformation performed on the input plaintext data can be reversed. We also verify that the intermediate encrypted data is different from the input data. Similarly, we instantiate decoders chained to encoders and input the different ciphertext values that are passed along with communicated packet. Since we don't need to verify intermediate values, we also can permutate the key, plaintext, and ciphertext on the emulator to create several hundred unique test cases from one communication packet. This greatly reduces the number of

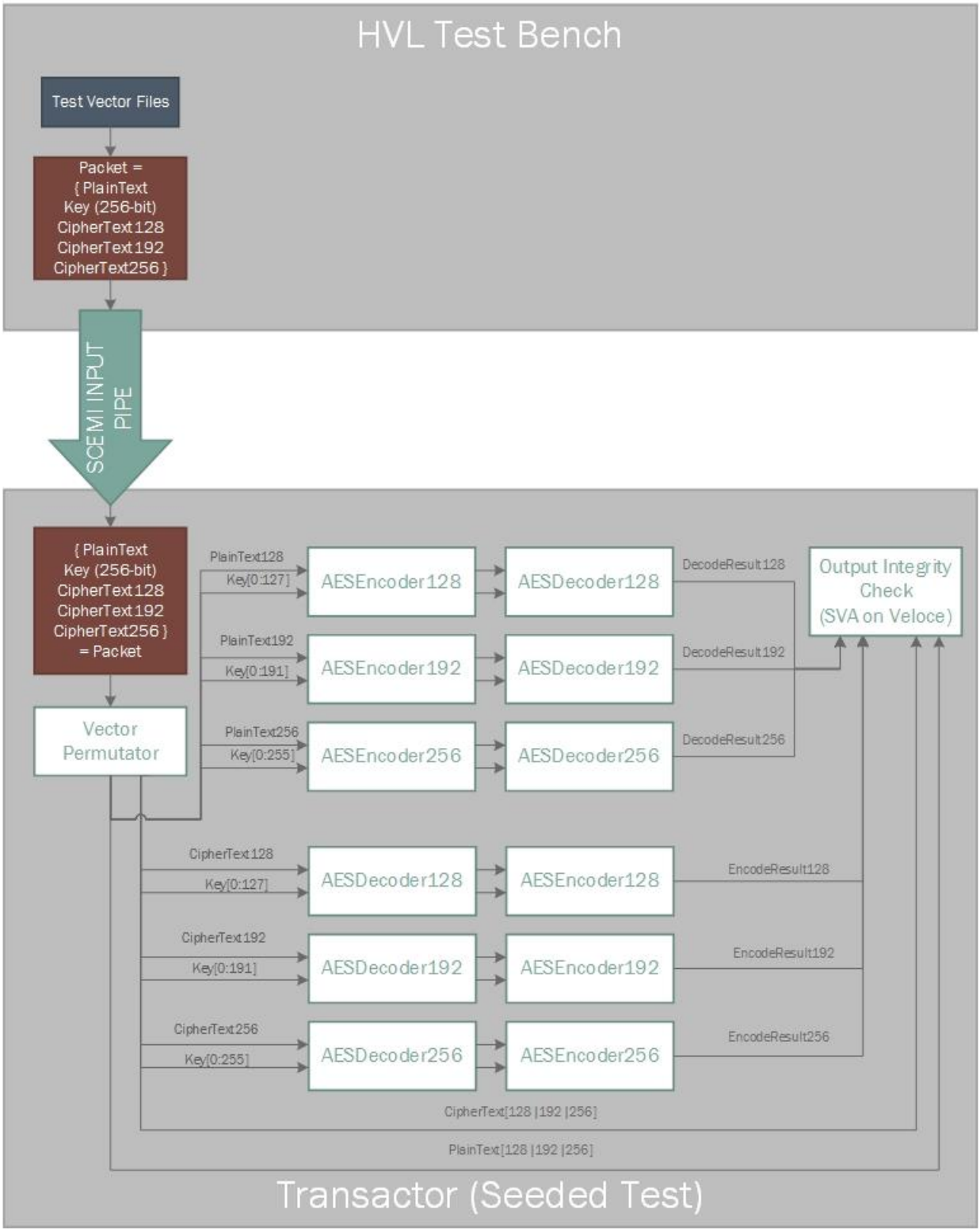

Figure 4: Top Level Testbench (Seeded Test)

Simulation and Emulation Execution Time for Different Configurations

Key Size(s)	128-bit	128-bit, 192-bit	128-bit, 192-bit, 256-bit
Simulation Runtime	3,192 s	6,888 s	11,373 s
Emulation Runtime	27 s	26 s	30 s
Emulation Speedup over Simulation	118x	265x	379x
Emulator HDL Time Advance (Throughput)	75.70%	71.49%	73.01%
Emulator Clock Speed	740 kHz	724 kHz	628 kHz

Table 1: Simulation and Emulation Execution Time for Different Configurations

synchronization points between the HVL and HDL portions of the testbench.

This strategy gives us confidence in the design while being efficient by allowing less data to be transferred to the Transactor. It could however mask potential bugs that were replicated in both the encoder and decoder data paths. To offset that risk, we coupled this approach with the directed testing discussed above.

IV. Results

We executed 5,000 directed tests and used 25,000 seeds (creating 6,425,000 tests) with the 128, 192, and 256-bit key encoders and decoders instantiated in parallel, simulating in Questa and emulating on the PSU Veloce Solo machine. Table 2: Simulation and Emulation Execution Time for Different Configurations and Figure 5: Simulation and Emulation Execution Time and Speedup shows our simulation and emulation execution time and the speedup relative to the simulation time. We achieve almost 400x improvement over simulation by using emulation and bring a multi-hour long task down to a half minute, making it practical to very quickly verify the functionality of the AES encoder and decoder over a huge range of input test vectors. Given that we don't achieve 100% throughput if we could use a concurrent data streaming method of communication to transfer data between the HVL testbench and the emulator we could achieve an additional 25% perofrmance improvement.

One of the challenges we encountered was that when we inferred RAM blocks for certain RTL elements, our design required a larger emulator than we had available to our team. While the logic of our design fit comfortably into the limits of the emulator, we required more than one logic board in order to synthesize the number of memory read ports we needed. When we modeled the specific RTL elements in our design as logic we fit comfortably into one logic board and were able to execute

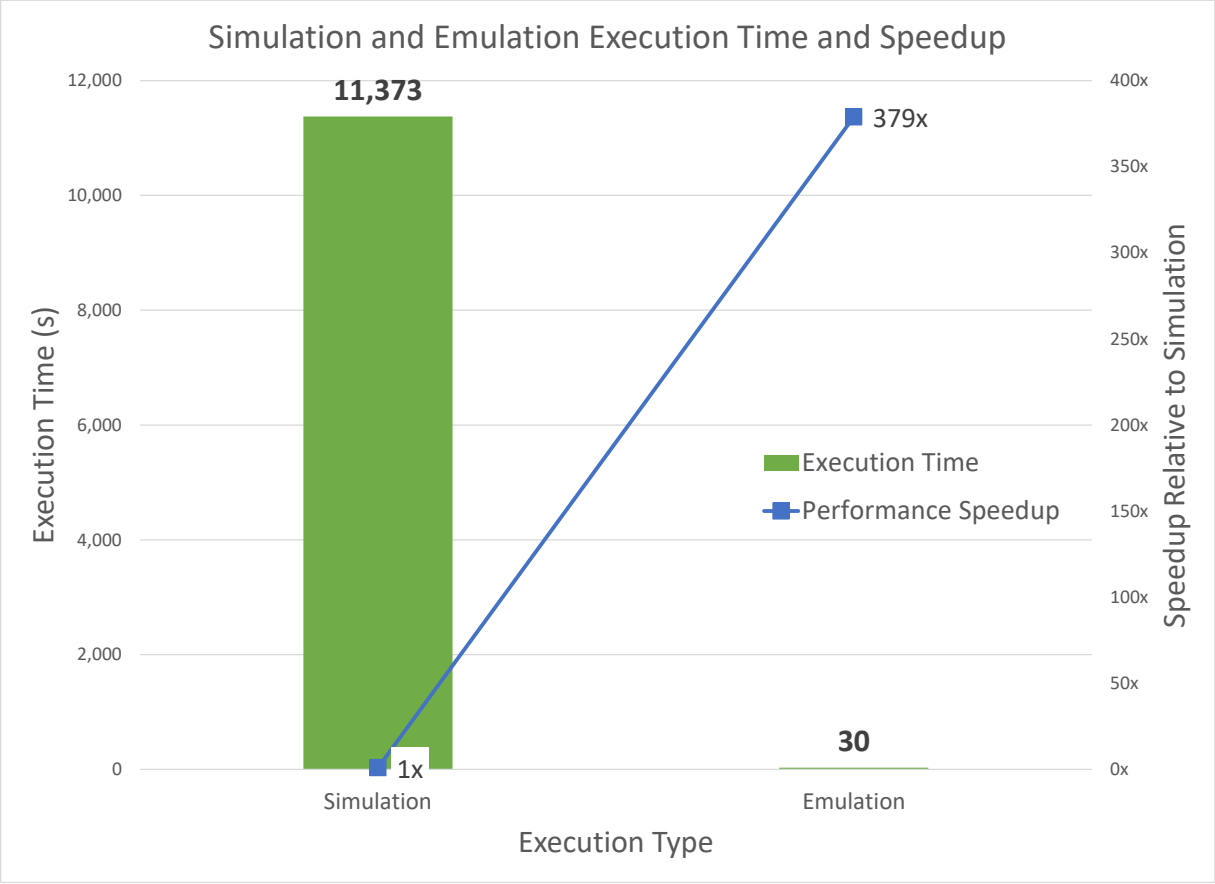

Figure 5: Simulation and Emulation Execution Time and Speedup

Simulation and Emulation Execution Time with and without inferred RAMs

	128-bit Key w/ Inferred RAMs	128-bit Key w/out Inferred RAMs
Simulation Runtime	3,174 s	3,192 s
Emulation Runtime	64 s	27 s
Emulation Speedup over Simulation	50x	118x
Emulator HDL Time Advance (Throughput)	87.59%	75.70%
Emulator Clock Speed	207 kHz	740 kHz

Table 2: Simulation and Emulation Execution Time with and without inferred RAMs

on the available emulator. Table 2: Simulation and Emulation Execution Time with and without inferred RAMs shows the simulation and emulation times with and without inferring RAMs for these particular elements; one thing to note is that we only simulate and emulate the 128-bit key values for the encoders and decoders because that is all that would fit on the available emulator in both modes. The way in which the elements are modeled has almost no impact on simulation, but has a much larger impact on how long emulation runs for. Modeling without RAMs executes faster on the emulator but also requires additional compilation time to route the additional logic. In our case, when RAMs are inferred our compile time is only 5 minutes compared to 10 minutes when RAMs are not inferred. Compilation time, emulation run time, and emulator resource utilization are all aspects that need to be considered when evaluating the verification acceleration strategy to employ.

One of the primary observations we made was that the simulation and emulation times scale differently as we add computation to the verification routine in the form of additional instantiations of different key size encoders and decoders. As we add additional module instantiations, the runtime of the simulation increases to account for the additional computation needed. With regards to emulation, even though we increase the computational burden and use additional Veloce resources all of the hardware executes in parallel so the runtime of the emulation doesn't increase.

Figure 6: Simulation and Emulation Execution Time for Different Numbers of Instantiated Encoder/Decoders shows this relationship and demonstrates that we can increase the

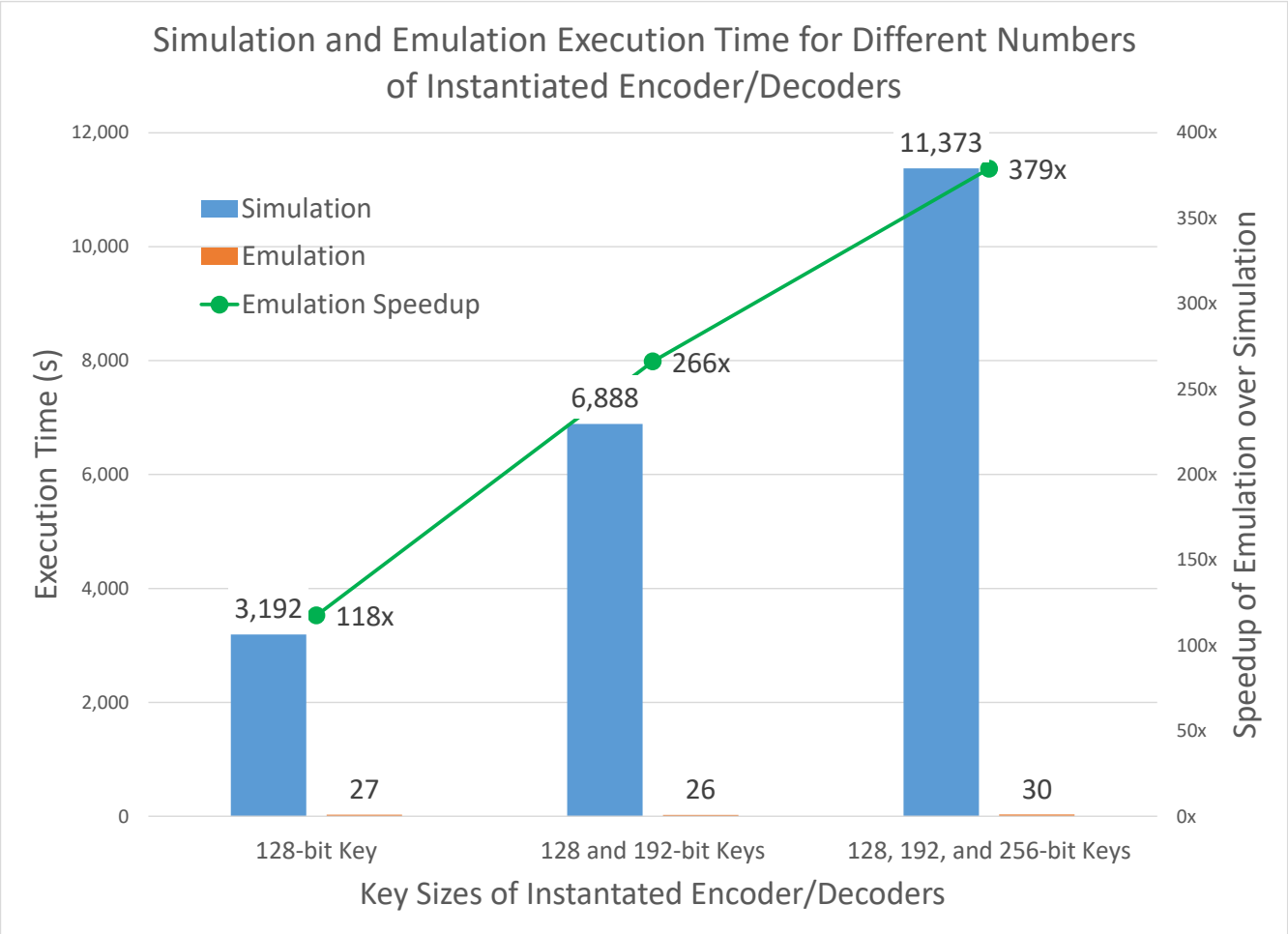

Figure 6: Simulation and Emulation Execution Time for Different Numbers of Instantiated Encoder/Decoders

speedup we achieve over simulation by adding more parallel hardware.

V. Conclusion

Emulation is an effective technology for accelerating the verification of large and complex RTL designs. If synchronization bottlenecks between the emulator and high level testbench can be eliminated, the design can execute much faster on the emulator than in simulation. While simulation runtime will increase with more computational load, emulation can handle additional computational load, due to parallel hardware execution, without increased runtime as long as the same critical path through the emulated logic can be maintained. The more work we are able to execute in parallel, the greater speedup we are able to achieve with emulation over simulation.

One of the challenges we encountered was the resource constraints of the Veloce emulator. Such device constraints can impose requirements on modeling certain RTL elements in order to effectively utilize the emulator's capacity. We also showed how modeling of certain RTL elements affects emulation runtime, but has a much smaller impact on simulation runtime.

This work could be extended in a couple of directions:

- More carefully tune the performance tradeoff of logic vs RAM for look up tables.
- Investigate advanced concurrent strategies for inbound and outbound streaming of data.
- Increase compiled frequency of the emulator by reducing the design critical path or finding additional compilation options.

References

[1] Advanced Encryption Standard. (2016, October 12). In *Wikipedia, The Free Encyclopedia.* Retrieved 13:29, October 12, 2016. (http://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=743995771)

[2] Pub, NIST FIPS. "197: Advanced encryption standard (AES)." Federal Information Processing Standards Publication 197 (2001): 441-0311. (http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf)

[3] S.-M. Yoo, D. Kotturi, D.W. Pan, J. Blizzard, An AES crypto chip using a high-speed parallel pipelined architecture, Microprocessors and Microsystems, Volume 29, Issue 7, 1 September 2005, Pages 317-326, ISSN 0141-9331, http://dx.doi.org/10.1016/j.micpro.2004.12.001. (http://www.sciencedirect.com/science/article/pii/S0141933104001632)