04/11/2024
Alexander Ingare
Hamza Iqbal

Advanced Encryption Standard Preliminary Project Report

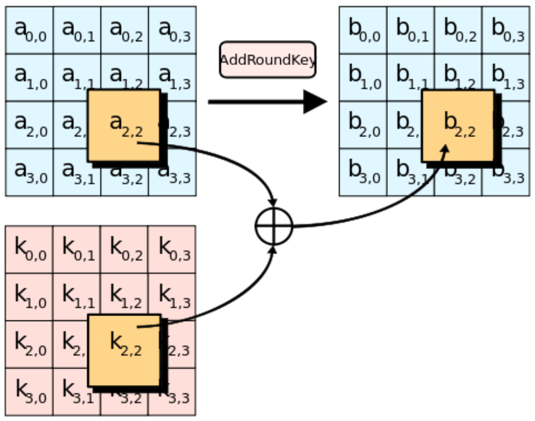
The rapid growth of communication networks and information sharing and storage technologies exhibits a growing global issue of data protection and privacy. Sensitive information such as personal messages, financial transactions, and even medical records, require encryption protection to ensure the right to privacy. Advanced Encryption Standard (AES; also known as Rijndael Encryption), adopted by the U.S. National Institute of Standards and Technology (NIST) in 2001, has been the standard encryption scheme to protect U.S. classified information and sensitive data around the world.

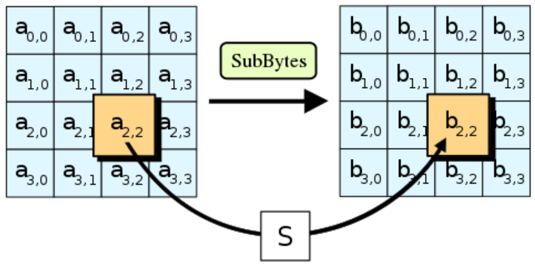
AES encryption is a variant of the Rijndael block cipher, a family of symmetric-key (same key used for encryption and decryption) ciphers with different key and block sizes. While the original Rijndael cipher supports key and block sizes in multiples of 32 bits, ranging from 128 to 256 bits, the three variants chosen by NIST that make up AES encryption only support block sizes of 128 bits along with key sizes of 128, 192, and 256 bits. Since AES is built on a design principle known as a substitution-permutation network (S-box substitutions and P-box permutations), a larger key size indicates a larger amount of transformation rounds to convert the input plaintext into the output ciphertext. For each key length (128, 192, 256), it takes 10, 12, and 14 transformation rounds, respectively. These transformations operate on intermediate results each known as a state. In AES, a state is a 4x4 column-major order array of the 16 bytes that make up a 128-bit block.

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

A high level description of the algorithm is as follows:

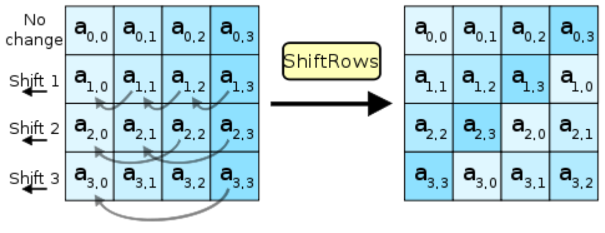
1. Key Expansion: The AES key scheduling algorithm expands a short key into several round keys for each transformation round
2. Initial round key addition (addRoundKey): Each byte of the state is bitwise XORed with a byte of the round key
3. Initial transformation rounds (9, 11, or 13 rounds):
   - subBytes: Each byte of the state is replaced with another based on a lookup table (S-box) with the lookup index being the numerical value of the state byte

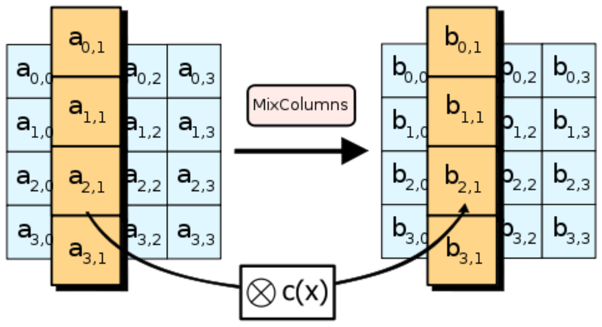




- shiftRows: Each row in the state is cyclically shifted a number of times based on the index of the row in the state
- mixColumns: The four bytes of each column of the state are combined using an invertible linear transformation
- addRoundKey

4. Final transformation round
    - subBytes
    - shiftRows
    - addRoundKey


No change
Shift 1
Shift 2
Shift 3
ShiftRows
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,0}$
$a_{2,2}$ $a_{2,3}$ $a_{2,0}$ $a_{2,1}$
$a_{3,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$


MixColumns
$\otimes$ c(x)
$a_{0,1}$ $a_{1,1}$ $a_{2,1}$ $a_{3,1}$
$b_{0,1}$ $b_{1,1}$ $b_{2,1}$ $b_{3,1}$

Since the nature of these transformations are friendly to both software and hardware implementations, AES encryption can be easily optimized on both ends.

Our initial software implementation for AES encryption included two different approaches. The first approach involved using the PyCryptodome Python library. This approach uses the Cipher Block Chaining (CBC) block cipher mode of operation, which requires a cipherkey and plaintext message as inputs for encryption and a cipherkey, initialization vector, and ciphertext for decryption. Our other approach was an Electronic Codebook (ECB) mode of operation without the use of external libraries. This initial EBC approach was only able to perform encryption whereas the PyCryptodome library utilized both encryption and decryption. Compared to CBC, this approach only requires a cipherkey and plaintext message for encryption and a cipherkey and ciphertext for decryption.

As we moved to implement AES encryption in Vitis HLS, we decided to instead find a simple ECB implementation in C that was based on the same tutorials that we learned AES from. Using a base implementation from GitHub, we then modified the code to make it compatible with the Vitis compiler and created a top-level function handling the Advanced eXtensible Interface (AXI) for the hardware overlay that also performs encryption and decryption. As a first implementation, we used an AXI4 interface for transferring cipherkey, plaintext, ciphertext, and decrypted-text character arrays and utilized a testbench that passed in a 128-bit key and 16 character plaintext message to verify our AES algorithm. Despite our C simulation working in Vitis, the associated hardware IP generated by Vivado failed to run in Jupyter notebooks as an overlay and left the output buffers unchanged. As we ran into issues with the data transfer in the Jupyter notebook, we received advice to change our interface to an AXI-Stream interface, which was the basis for our second and final AXI implementation.

The streaming interface originally consisted of two input streams (one for the cipher key and one for the plaintext) along with two output streams (one for the ciphertext and one for the decrypted text). However, the Vivado DMA blocks only utilized one streaming input and output, so we opted to combine the two input streams into one and the two output streams since the size of the stream was always of a fixed length. For a very long time we ran into an issue with the streaming interface either hanging forever (presumably waiting for a TLAST signal) or not producing an output stream. Two errors we found that seemed to resolve our problems were changing the allocated buffers in Python to be of type uint8 rather than uint32, and to set the output stream's last signal outside of the streaming loop. Once we verified that our streaming interface was working for AES-128, we decided to modify our top level function to include AES mode selection (128, 192, 256) that is set by an AXI4-LITE hardware register associated with the IP. Since there is a hard limit on how much can be streamed to the DMA at once (in our case 64 elements), we stuck with streaming just one block of plaintext (16 bytes) with the varying AES modes.

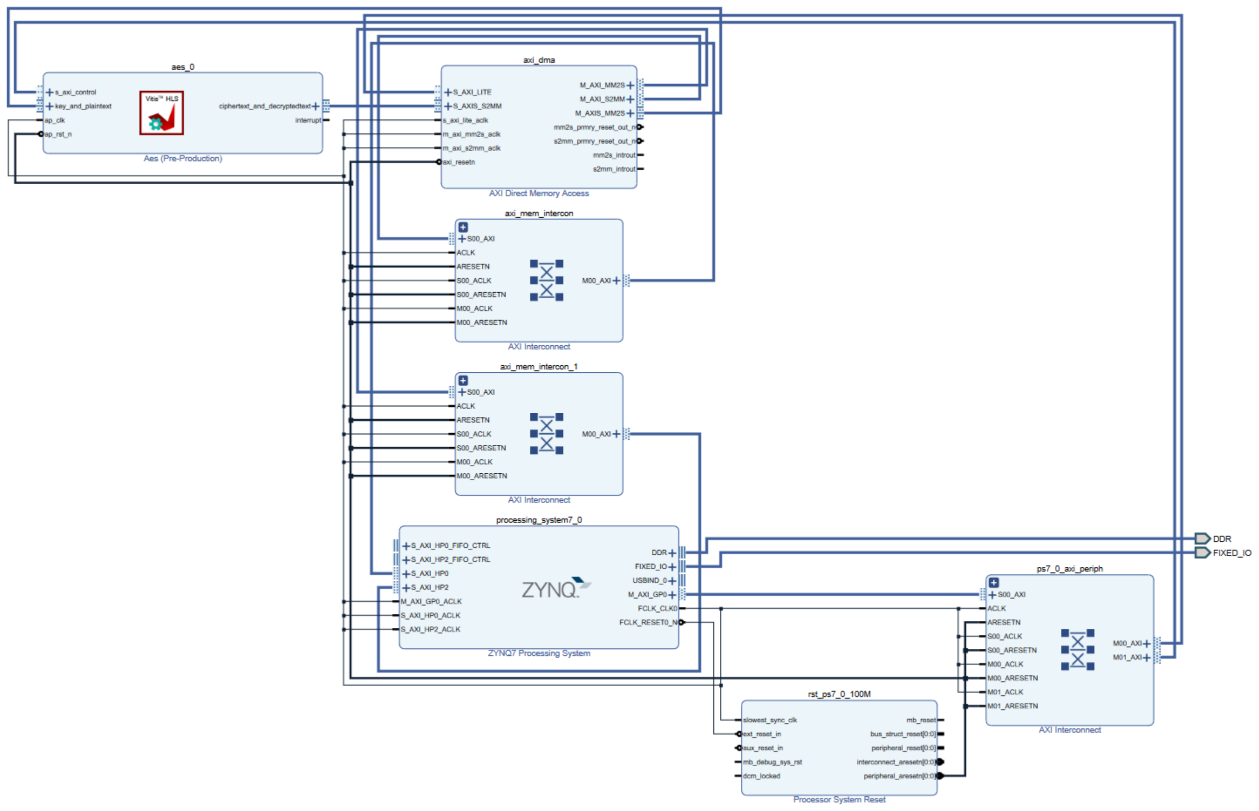


Figure 1. Vivado Block Diagram for AXI-Stream Implementation

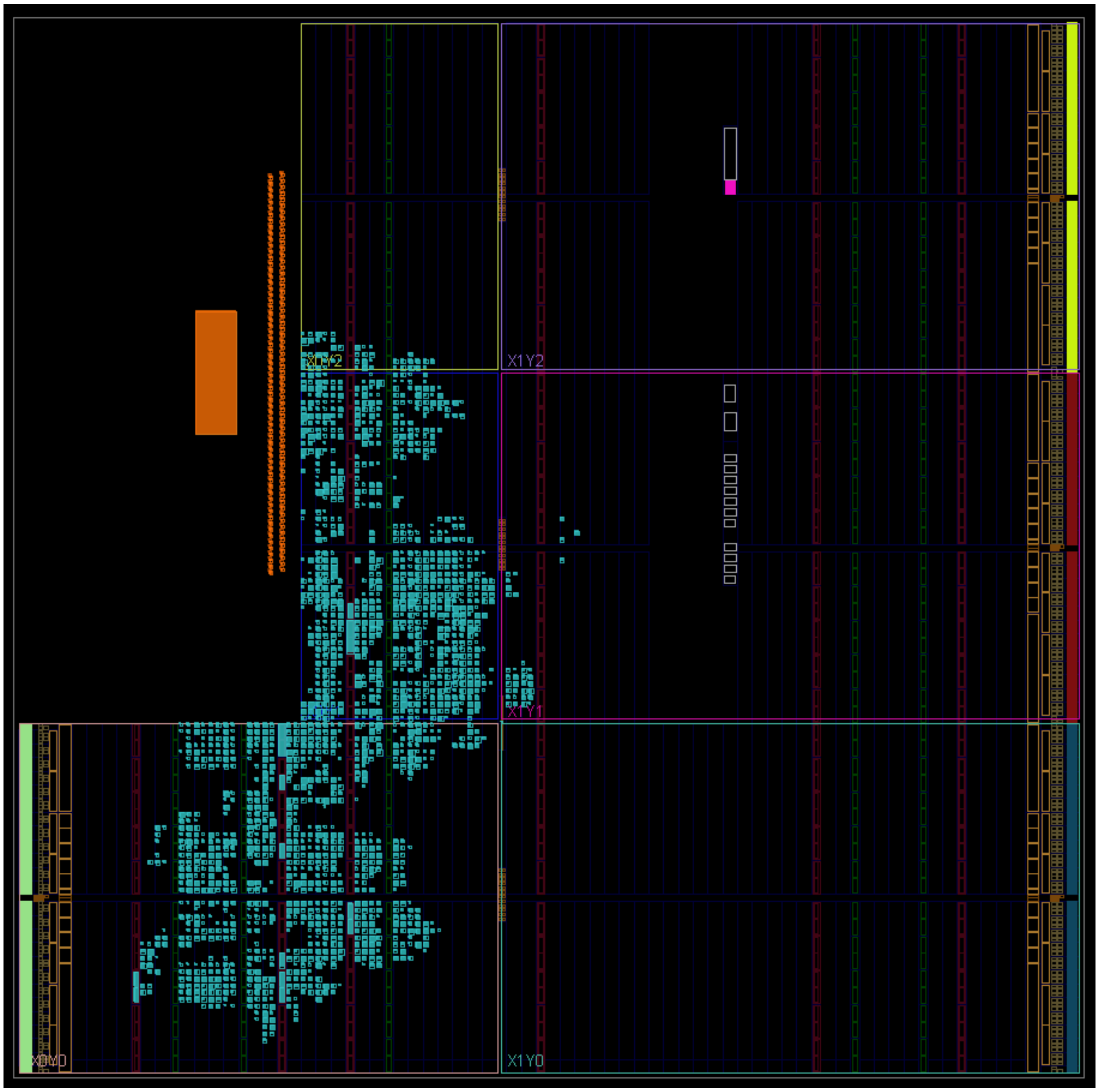
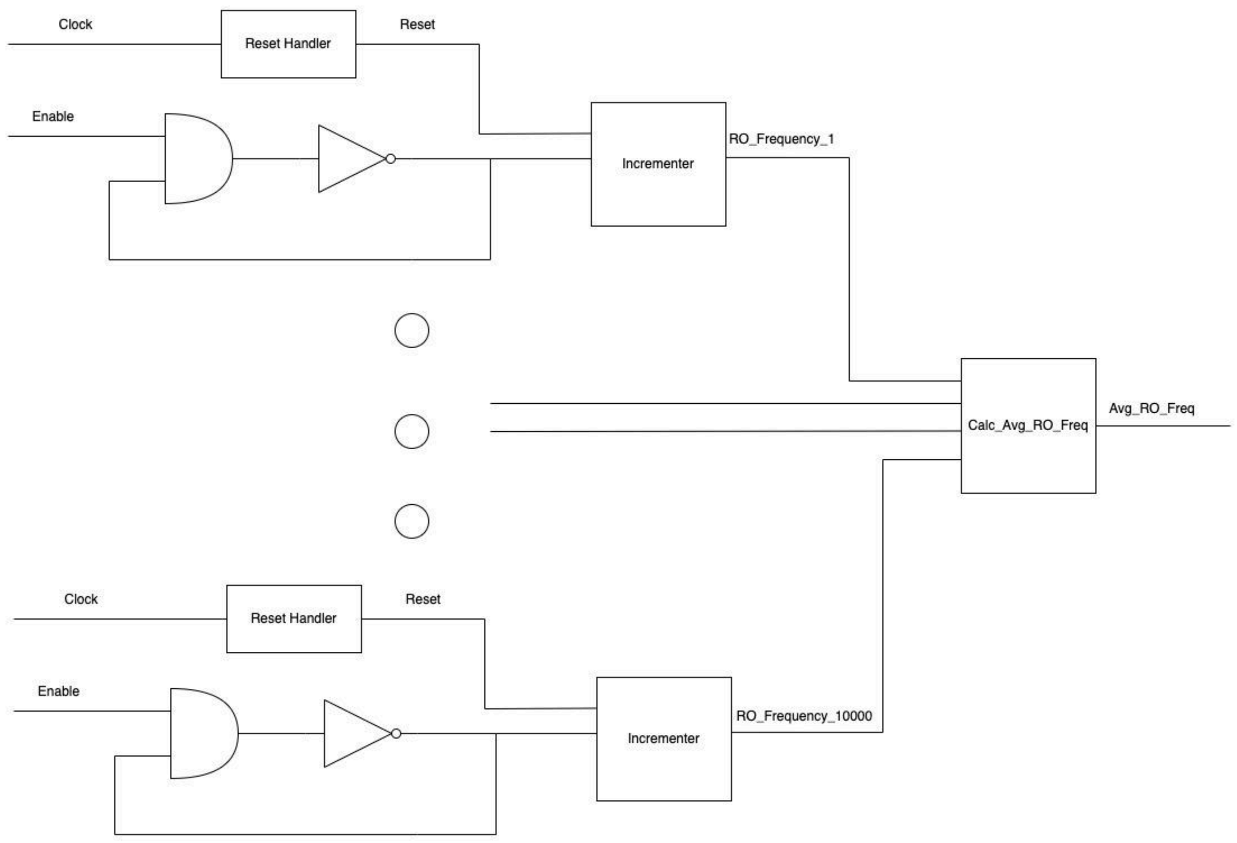
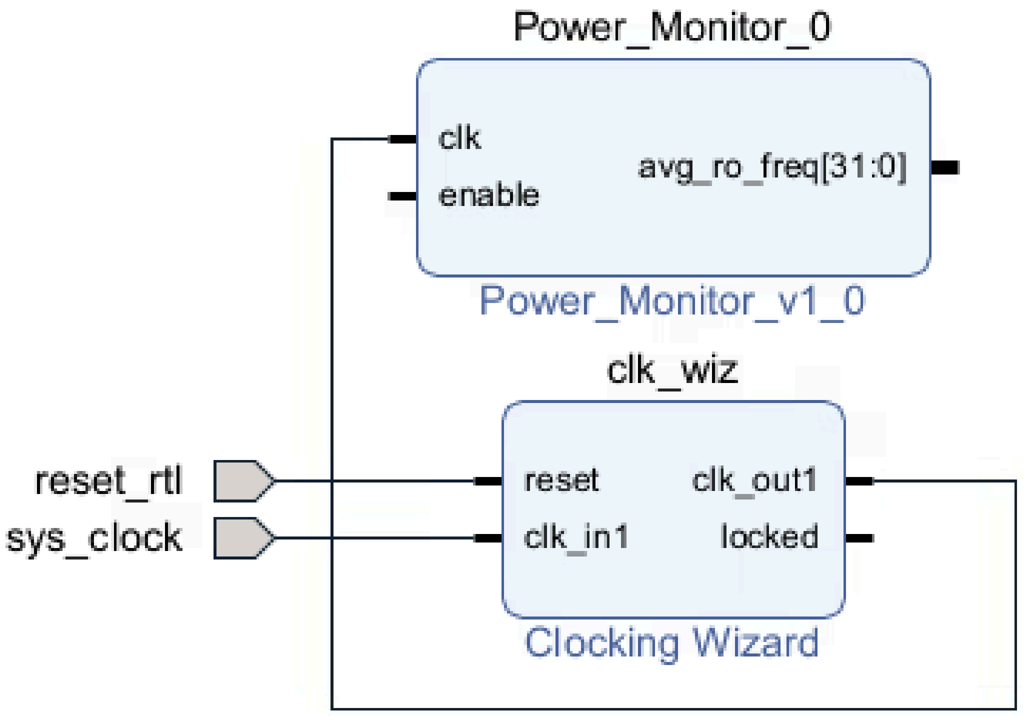


Figure 2. Vivado Device Chip Layout

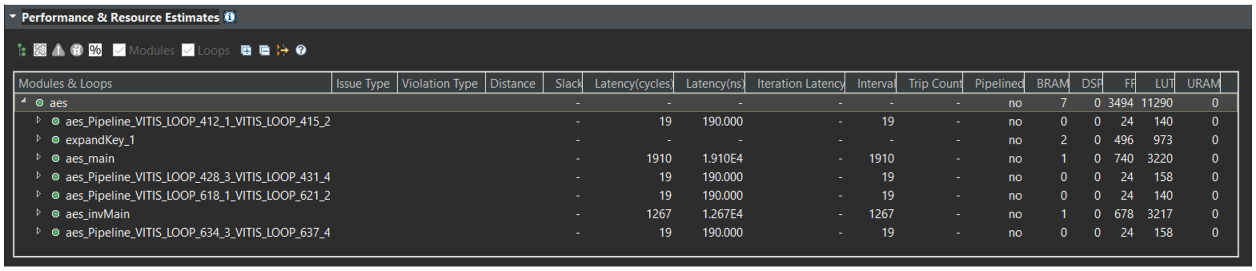
Once we had developed the AES module in hardware it was time to move on to part two of our project, which is to build a power monitor on the PL in order to enable a power analysis side channel attack of our AES module. The power monitor we decided to create is based on ring oscillators. A ring oscillator is simply an AND gate and an inverter connected to each other in order to generate a rapidly oscillating signal. The reason that this type of circuit can be used as a power monitor is because when there is higher power consumption on the board due to the nature of the power distribution unit, the function of each gate slows down to a measurable amount. This means that when there is a larger power draw the average frequency of the array of ring oscillators should slow down enough to create a measurable difference. Another key aspect enabling this is that the system clock is unaffected by this so we can still use that as a reference with which to take our measurements. We used the following design for the power monitor circuit.



In order to design this circuit we needed to design a new IP block to integrate into our design using System Verilog. The reason we needed to move down a level of abstraction is because this design involves gate level logic, which cannot be implemented using Vitis HLS. Initially we tried using a Verilog implementation but that proved to be futile because we couldn't pass a bus of 32 bit integers between modules in Verilog, but that capability is available in System Verilog. Since the syntax is pretty much the same we just had to change the file type to get it to work. We were able to create a successful implementation and create an IP from that which will be integrated with our AES module.



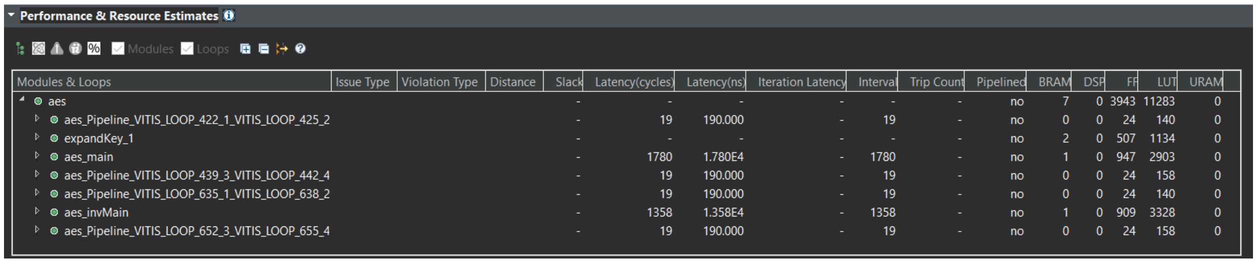
Results:
Our initial design space exploration involved replacing the majority of the loops within our HLS code with directives to flatten and/or unroll loops. This resulted in the following synthesis summary for our AXI4 implementation:



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
aes				-	-	-	-	-	-	no	7	0	3494	11290	0
aes_Pipeline_VITIS_LOOP_412_1_VITIS_LOOP_415_2				-	19	190.000	-	19	-	no	0	0	24	140	0
expandKey_1				-	-	-	-	-	-	no	2	0	496	973	0
aes_main				-	1910	1.910E4	-	1910	-	no	1	0	740	3220	0
aes_Pipeline_VITIS_LOOP_428_3_VITIS_LOOP_431_4				-	19	190.000	-	19	-	no	0	0	24	158	0
aes_Pipeline_VITIS_LOOP_618_1_VITIS_LOOP_621_2				-	19	190.000	-	19	-	no	0	0	24	140	0
aes_invMain				-	1267	1.267E4	-	1267	-	no	1	0	678	3217	0
aes_Pipeline_VITIS_LOOP_634_3_VITIS_LOOP_637_4				-	19	190.000	-	19	-	no	0	0	24	158	0

Figure 3. AES AXI4 Unroll/Flatten Optimizations

We then attempted to see the impact with using all performance directives instead, and we received the following synthesis results:

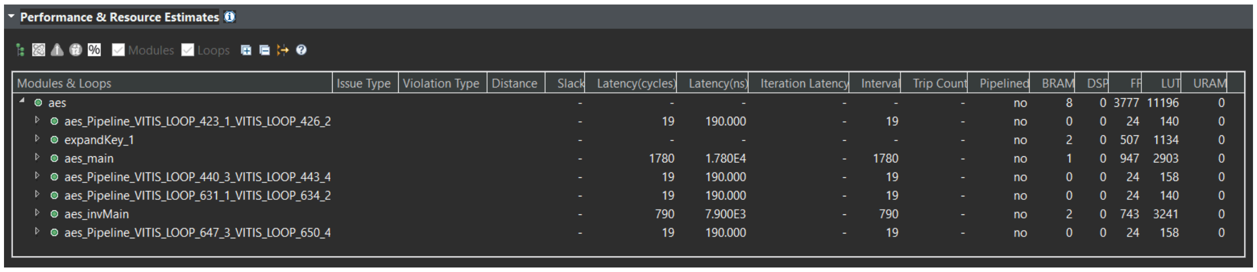


Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
aes				-	-	-	-	-	-	no	7	0	3943	11283	0
aes_Pipeline_VITIS_LOOP_422_1_VITIS_LOOP_425_2				-	19	190.000	-	19	-	no	0	0	24	140	0
expandKey_1				-	-	-	-	-	-	no	2	0	507	1134	0
aes_main				-	1780	1.780E4	-	1780	-	no	1	0	947	2903	0
aes_Pipeline_VITIS_LOOP_439_3_VITIS_LOOP_442_4				-	19	190.000	-	19	-	no	0	0	24	158	0
aes_Pipeline_VITIS_LOOP_635_1_VITIS_LOOP_638_2				-	19	190.000	-	19	-	no	0	0	24	140	0
aes_invMain				-	1358	1.358E4	-	1358	-	no	1	0	909	3328	0
aes_Pipeline_VITIS_LOOP_652_3_VITIS_LOOP_655_4				-	19	190.000	-	19	-	no	0	0	24	158	0

Figure 4. AES AXI4 PERFORMANCE Directive Optimization

Combining the better of the two different optimizations for aes_main and for aes_invmain, we then got our final optimization results prioritizing latency:

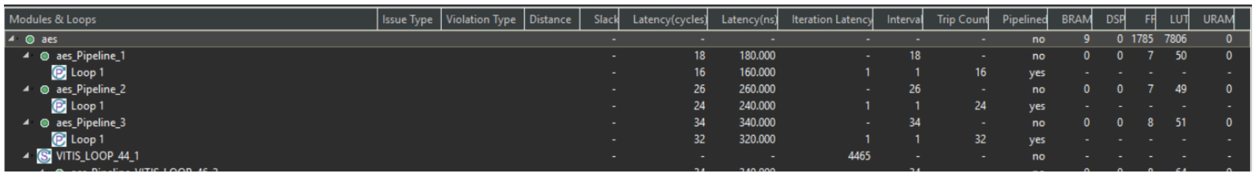


Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
aes				-	-	-	-	-	-	no	8	0	3777	11196	0
aes_Pipeline_VITIS_LOOP_423_1_VITIS_LOOP_426_2				-	19	190.000	-	19	-	no	0	0	24	140	0
expandKey_1				-	-	-	-	-	-	no	2	0	507	1134	0
aes_main				-	1780	1.780E4	-	1780	-	no	1	0	947	2903	0
aes_Pipeline_VITIS_LOOP_440_3_VITIS_LOOP_443_4				-	19	190.000	-	19	-	no	0	0	24	158	0
aes_Pipeline_VITIS_LOOP_631_1_VITIS_LOOP_634_2				-	19	190.000	-	19	-	no	0	0	24	140	0
aes_invMain				-	790	7.900E3	-	790	-	no	2	0	743	3241	0
aes_Pipeline_VITIS_LOOP_647_3_VITIS_LOOP_650_4				-	19	190.000	-	19	-	no	0	0	24	158	0

Figure 5. AES AXI4 PERFORMANCE & Flatten/Unroll Directive Optimizations

The unoptimized baseline AXI-Stream implementation has the following synthesis summary:



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
aes				-	-	-	-	-	-	no	9	0	1785	7806	0
aes_Pipeline_1				-	18	180.000	-	18	-	no	0	0	7	50	0
Loop 1				-	16	160.000	1	1	16	yes	-	-	-	-	-
aes_Pipeline_2				-	26	260.000	-	26	-	no	0	0	7	49	0
Loop 1				-	24	240.000	1	1	24	yes	-	-	-	-	-
aes_Pipeline_3				-	34	340.000	-	34	-	no	0	0	8	51	0
Loop 1				-	32	320.000	1	1	32	yes	-	-	-	-	-
VITIS_LOOP_44_1				-	-	-	4465	-	-	no	-	-	-	-	-

Figure 6. AES AXI-Stream Baseline Summary

The timing results of our unoptimized/baseline AES encryption overlays are as follows:

Hardware AES Mode	**Encryption & Decryption Time (seconds)**
AES-128	0.0018680095672607422
AES-192	0.0018033981323242188
AES-256	0.0017251968383789062

Table 1. Hardware AES Implementation Timing Results

Software AES Implementation	**Time (seconds)**

PyCryptodome (Encryption + Decryption)	0.0008010864258*
Our AES algorithm	0.004860877990722656

Table 2. Initial Software AES Implementation Timing Results

*It's important to note that the PyCryptodome, since it requires an external Python library, was benchmarked on a Python kernel on our laptops rather than on the PYNQ-Z2 board. Our AES algorithm we wrote from scratch, however, was run on the board.

Lessons Learned:
One of the biggest challenges we faced was in our understanding of the streaming interface and recognizing where and what bugs would occur even when they would not produce error logs. It was incredibly frustrating seeing our C testbenches and simulations working perfectly and our hardware overlay not working at all. If we were to do something differently, it would have been to start with the C implementation of AES encryption and convert that to Python to create a software implementation we could compare to in our Jupyter notebooks. Similarly, we would have just started with an AXI-Stream interface instead of starting with an AXI4 interface since it ended up being a dead-end.

Future Work:
For the future, we plan on recreating our AES algorithm (C code) in Python instead to directly compare the effectiveness of a hardware implementation of the same algorithm instead of our two initial software implementations (which are of different algorithms). Since the PyCryptodome based implementation is already heavily optimized and our raw AES Python implementation only has encryption (to a very broken degree), it would make more sense to recreate our simple AES algorithm in Python and compare to see if the hardware optimizations truly had an impact on performance. Since we also have only explored baseline optimizations for our AXI4 implementation, we will also plan to use the same, if not more optimizations for our streaming implementation and directly compare these optimizations both to our unoptimized hardware implementation and to our old and new software implementations. If we had more time, we would also consider operating on more than just one 128-bit block of plaintext at a time.

On the power side channel attack side of things we plan on integrating our power monitor into the HLS module. This will be done using the registers and we will directly read from the registers in our Python code. The plan is to sample the frequency output every 100 picoseconds and visualize the power consumption data. Once we are able to visualize the data the next step will be to analyze it and try to extract the AES key manually. Once this is done we will create a script to automatically post process the power analysis data and derive the key. Some challenges we foresee with this approach are timing of our sampling, and isolating the correct steps in the encryption process to help us recover the key.

References:

https://github.com/m3y54m/aes-in-c

https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf#page=1

https://wikimedia.org/api/rest_v1/media/math/render/svg/63ac3cf2cb47d5a29c1210fca521f9e4e49e39b1

https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#/media/File:AES-SubBytes.svg

https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#/media/File:AES-ShiftRows.svg

https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#/media/File:AES-MixColumns.svg

https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#/media/File:AES-AddRoundKey.svg