Power-Analysis Based Side Channel Attack of FPGA Accelerated AES Encryption

Alexander Ingare & Hamza Iqbal

Introduction and Overview

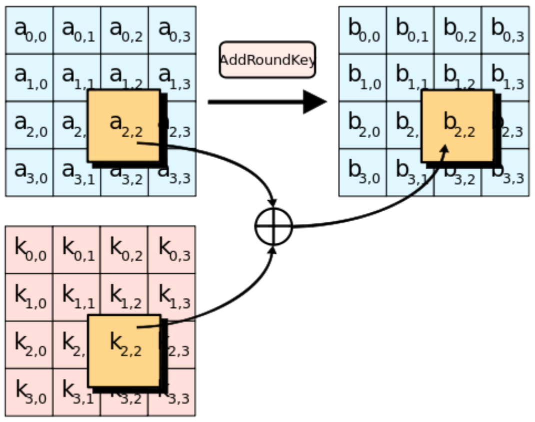
The rapid growth of communication networks and information sharing and storage technologies exhibits a growing global issue of data protection and privacy. Sensitive information such as personal messages, financial transactions, and even medical records, require encryption protection to ensure the right 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.

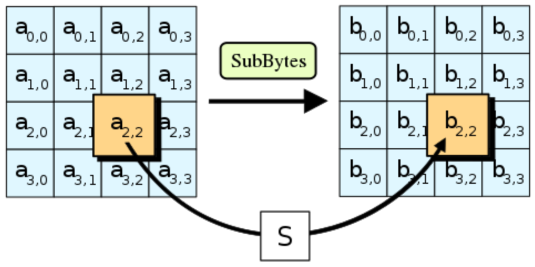
AES encryption is a variant of the Rijndael block cipher, a family of symmetric-key (same key used for encryption and decryption) ciphers with different key and block sizes. While the original Rijndael cipher supports key and block sizes in multiples of 32 bits, ranging from 128 to 256 bits, the three 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:

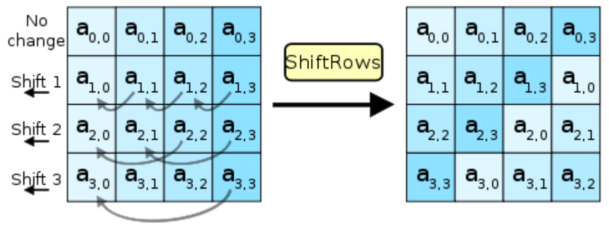
1. Key Expansion: The AES key scheduling algorithm expands a short key into several round keys for each transformation round
2. Initial round key addition (AddRoundKey): Each byte of the state is bitwise XORed with a byte of the round key
3. Initial transformation rounds (9, 11, or 13 rounds):
    - 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

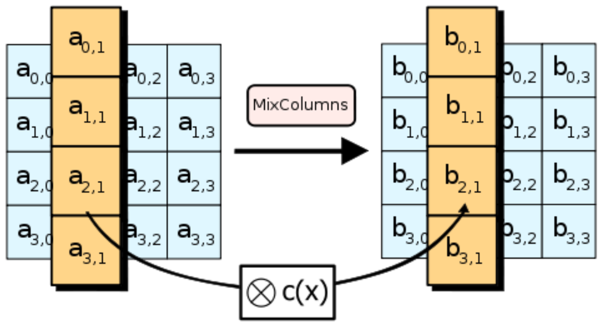




- ShiftRows: Each row in the state is cyclically shifted several times based on the index of the row in the state
- MixColumns: The four bytes of each column of the state are combined using an invertible linear transformation
- addRoundKey

4. Final transformation round
    - SubBytes
    - ShiftRows
    - AddRoundKey





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

Project Description

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 cipher key and plaintext message as inputs for encryption and a cipher key, 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 cipher key and plaintext message for encryption and a cipher key and ciphertext for decryption.

As we moved to implement AES encryption in Vitis HLS, we decided to find instead 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 cipher key, 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 encryption and decryption algorithm. Despite our C simulation working in Vitis, the associated hardware IP generated by Vivado failed to run in Jupyter Notebook 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. After a considerable amount of debugging, we discovered two errors that seemed to resolve our problems. The first error involved the buffers allocated by Numpy for DMA sending and receiving. Instead of allocating buffers with a data type of uint32, we changed it to be of type uint8 since we were transferring characters (8 bits). The other issue involved the order in which we were setting the output stream's last signal (TLAST). After moving the assignment of the last signal outside of the main streaming while loop, our overlay began working. Once we verified that our streaming interface was working for AES-128 encryption and decryption, we decided to modify our top-level function to include AES mode selection (AES-128, AES-192, AES-256). This parameter 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 chose to stick to streaming just one block (128 bits or 16 bytes) of plaintext with the varying AES modes.

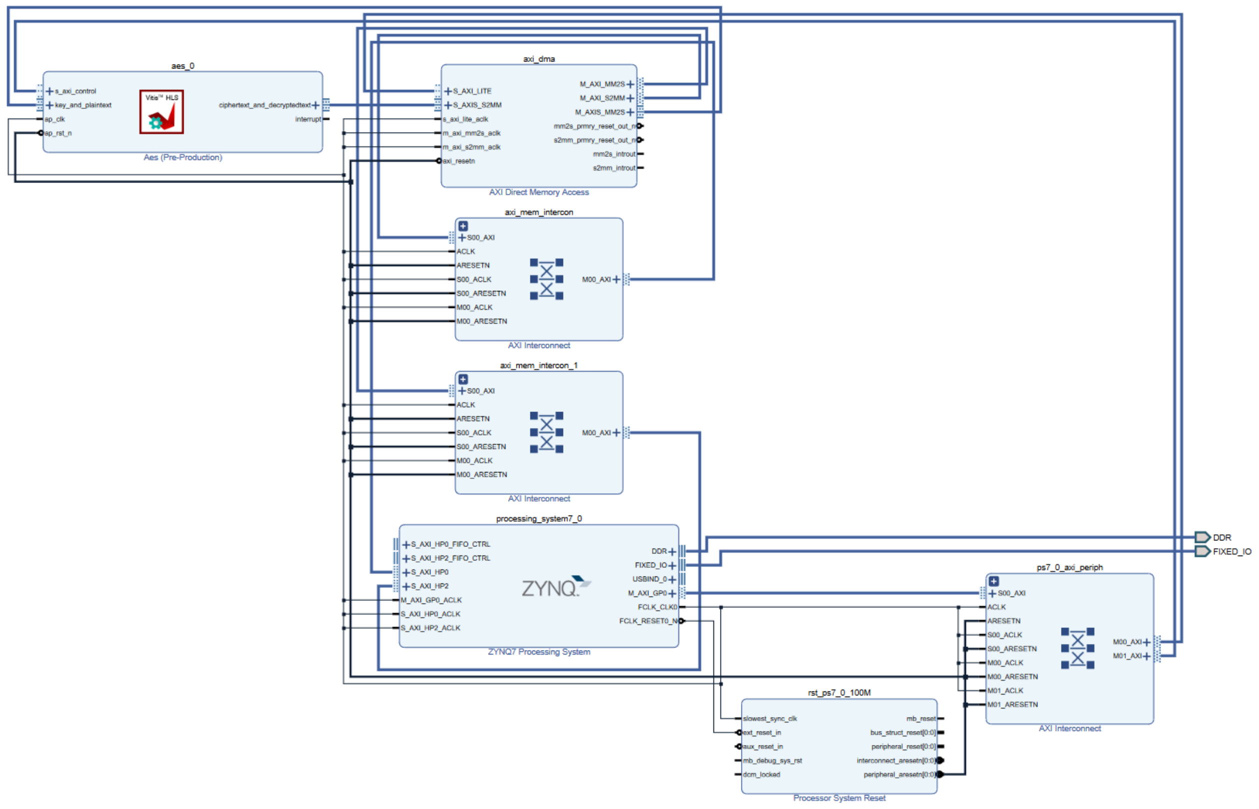


Figure 1. Vivado Block Diagram for AXI-Stream Implementation

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 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 to generate a rapidly oscillating signal. The reason that this type of circuit can be used as a power monitor is that 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.

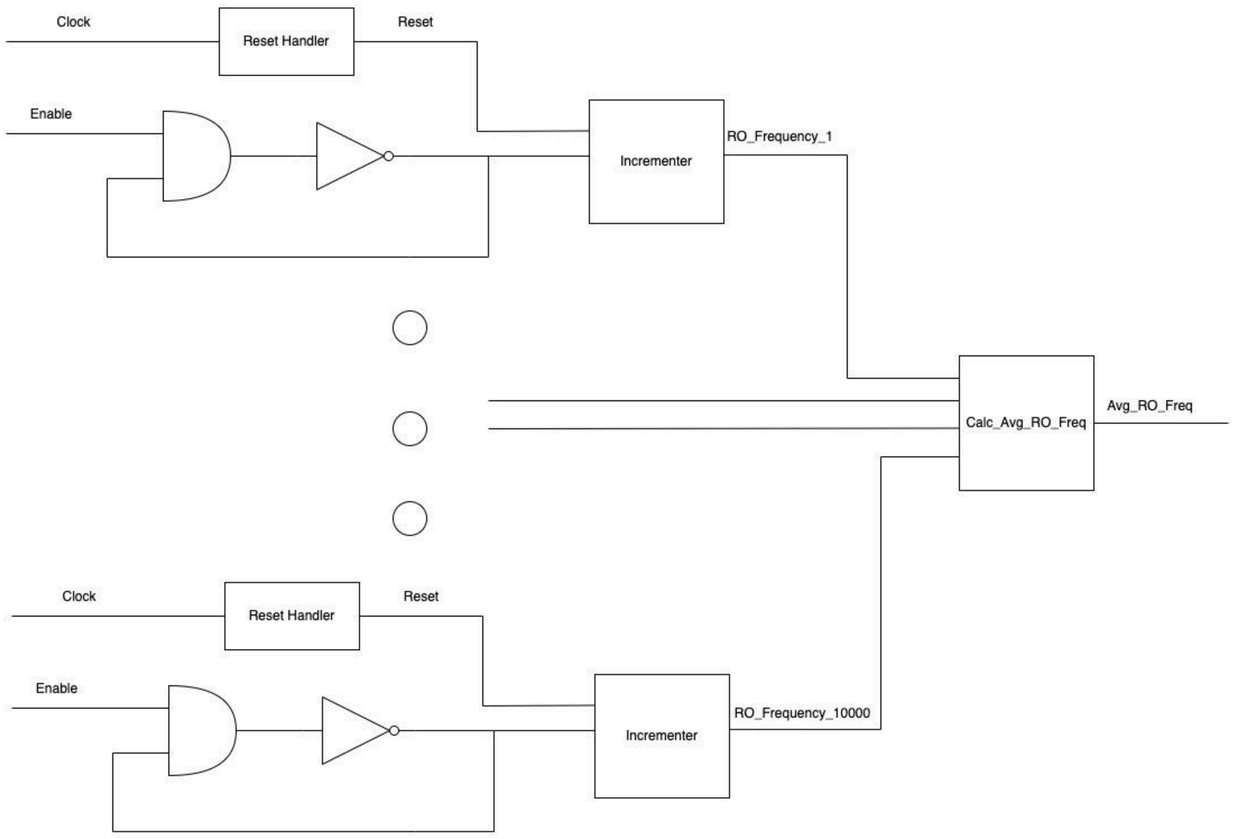


Figure 2. Power Monitor Circuit

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 that 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.

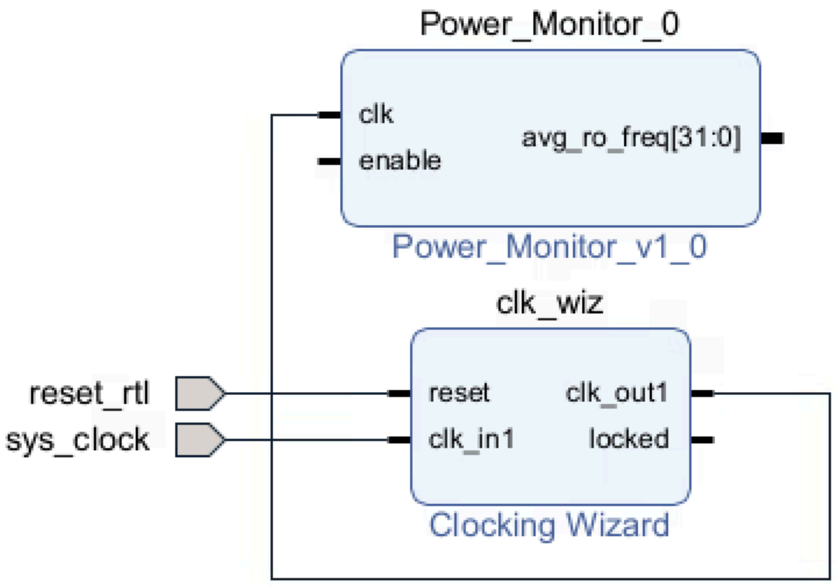


Figure 3. Power Monitor Block Design

We then integrated this IP into the rest of the project. The first step of this process was to modify the existing AES IP to include extra ports for the power monitor so that we could interact with it. Once that was done we regenerated the IP and recreated our original block diagram, but this time we added the power monitor as part of the design as well.

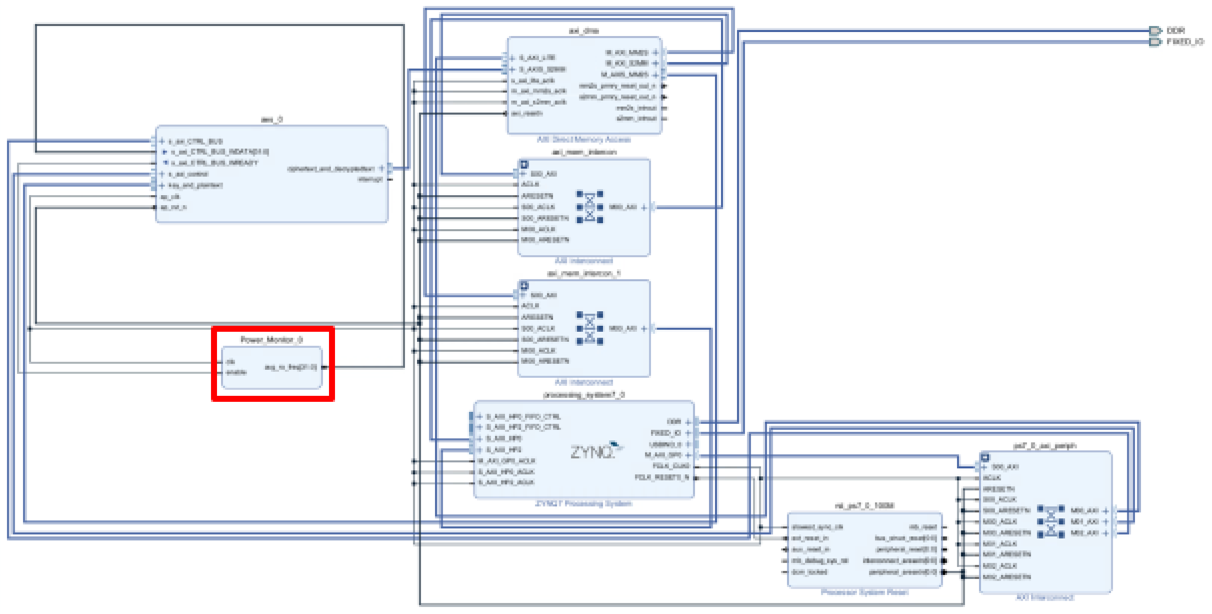


Figure 4. AES-Power Monitor Block Design

Using this block diagram we generated a bitstream successfully but when we tried to validate on the board we got error messages saying that key parts were missing. Upon further analysis, we realized that the implemented design had not been implemented at all and we were just using an empty bitstream. This prompted us to go back to the drawing board and make sure that we were able to generate an implemented design.

To implement the design we realized we needed some way to force Vivado to put our design on the board. After much research, we found that this could be done using directives directly in the code. Using these directives we were able to force Vivado to implement our design.

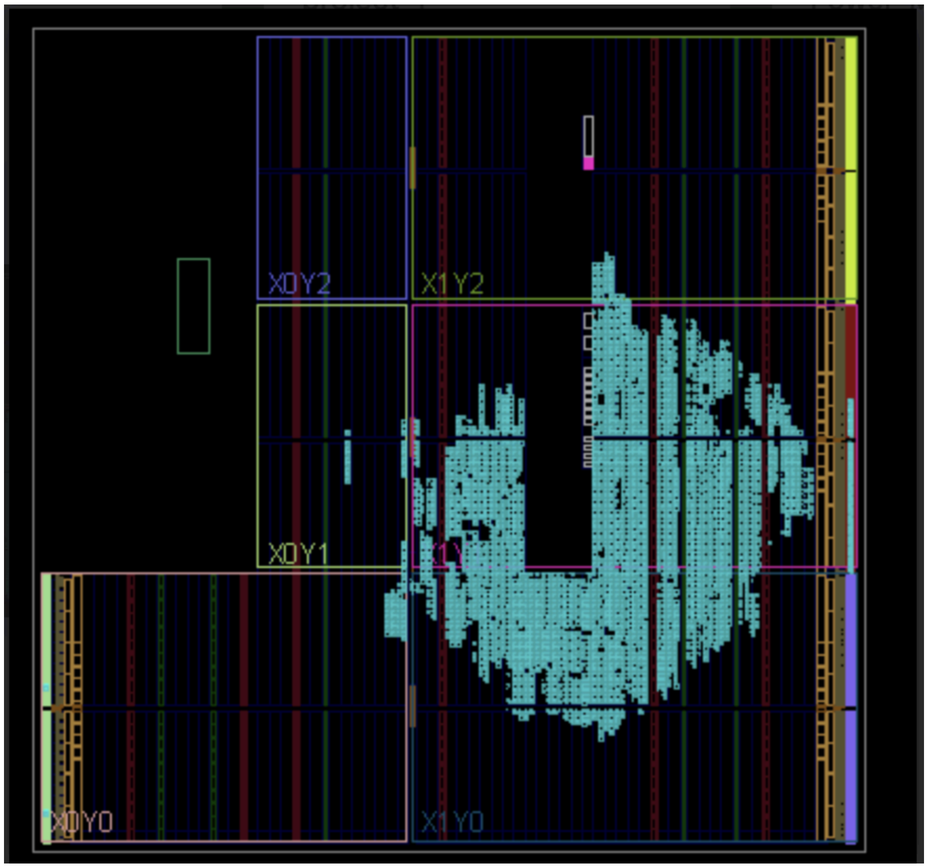
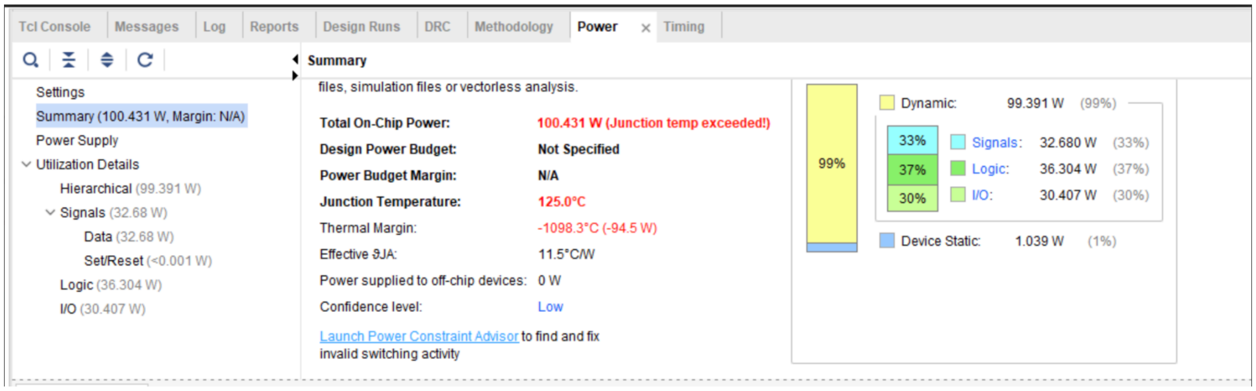


Figure 5. Power Monitor Floorplanning

Unfortunately, at this point, we realized two issues: the design is relatively large and it draws way too much power. In the figure below you can see that the predicted temperature of the board with this implementation was 125 degrees Celsius.


Tcl Console | Messages | Log | Reports | Design Runs | DRC | Methodology | Power | Timing

Settings
Summary (100.431 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (99.391 W)
Signals (32.68 W)
Data (32.68 W)
Set/Reset (<0.001 W)
Logic (36.304 W)
I/O (30.407 W)

Summary

files, simulation files or vectorless analysis.

Total On-Chip Power:	100.431 W (Junction temp exceeded!)
Design Power Budget:	Not Specified
Power Budget Margin:	N/A
Junction Temperature:	125.0°C
Thermal Margin:	-1098.3°C (-94.5 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

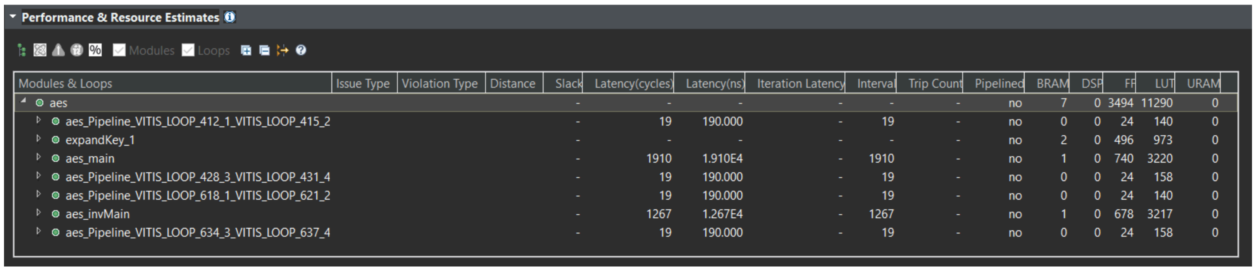
99%	Dynamic:	99.391 W	(99%)
33%	Signals:	32.680 W	(33%)
37%	Logic:	36.304 W	(37%)
30%	I/O:	30.407 W	(30%)
	Device Static:	1.039 W	(1%)

Figure 6. Power Monitor Chip Summary

To avoid this issue we scaled down the design by a factor of 100 and this allowed us to be in the desired temperature, power, and size range.

Results

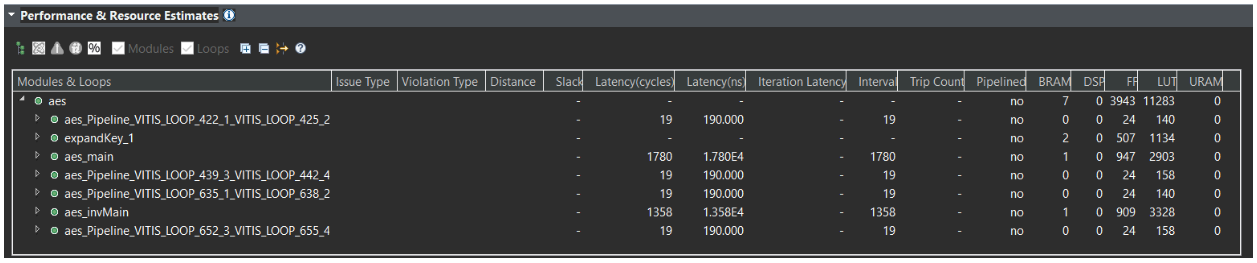
Our initial design space exploration was done on our earlier AXI4 implementation and involved replacing most of the loops within our HLS code with directives to flatten and/or unroll loops. This resulted in the following synthesis summary for our AXI4 implementation:


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	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 7. AES AXI4 Unroll/Flatten Optimizations

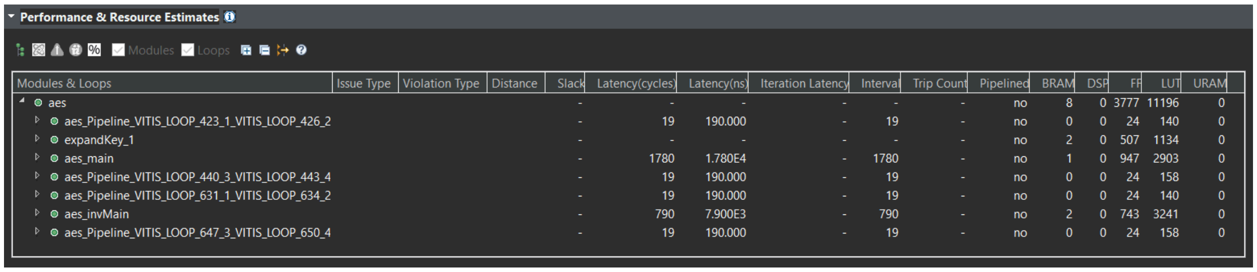
We then attempted to see the impact of using all performance directives instead.


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 8. AES AXI4 PERFORMANCE Directive Optimization

Combining the better of the two different optimizations for aes_main and aes_invmain, we then got our initial cumulative 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 9. AES AXI4 PERFORMANCE & Flatten/Unroll Directive Optimizations

After we had written and verified a working AXI-Stream algorithm for all AES modes, we used the information from our initial design-space exploration as a starting point in our finalized optimizations for AXI-Stream. However, we noticed that from our initial exploration, a total latency summary was missing for our entire IP. The underlying cause for this is that throughout our algorithm, there exist various loops with variable tripcounts. To prevent this, we added tripcount directives for all variable tripcount loops to get a latency statistic for the entire IP. We also went through each function, including ones with the newly added tripcount directives, and selectively added the best optimizations. For example, we used loop unrolling for loops that had a known, static, end condition and pipelining for more complex data-dependent loops. On the other hand, some functions had simultaneous reads of our global substitution and permutation boxes when unrolled, so we forced the synthesis tool to synthesize them as 2-port ROMs. But perhaps most importantly, since some of the largest latency loops within our program had variable tripcounts and we had just added the loop tripcount directives, we could now use performance directives with curated target_ti values to further optimize our algorithm. Below is a list of compounding function optimizations (each function optimization is built on the last) with details for each optimization performed.

Function Optimized	**Optimization Description**
Baseline	Default algorithm with custom Initiation Interval pipelining to satisfy II violations
expandKey	Bind_storage directive for Sbox, array partitioning for a temporary array, performance directive (target_ti=8) for function loop #1, and pipeline directive for function loop #2
subBytes	Unroll directive
createRoundKey	Pipeline directive for function loop #2
aes_main	Performance directive (target_ti=500)
invSubBytes	Unroll directive

invShiftRows	Unroll directive
shiftRow/shiftRows	Pipeline and unroll directives
invMixColumns	Improved initiation interval pipelining (from baseline)
aes_invMain (final optimization)	Performance directive (target_ti=600)

Table 1. Function Optimization Descriptions

Optimization	**Latency (cycles)**	**BRAM**	**FF**	**LUT**
Baseline	11828	9	1785	7806
expandKey	10386	9	4405	10751
subBytes	10334	9	4555	11235
createRoundKey	9753	9	4548	11516
aes_main	8665	9	4811	12256
invSubBytes	8595	8	4929	12711
invShiftRows	8057	8	4990	12703
shiftRow/shiftRows	8025	8	5003	12770
invMixColumns	7973	8	5018	13011
aes_invMain	7379	9	5169	13126

Table 2. Optimization Synthesis Results

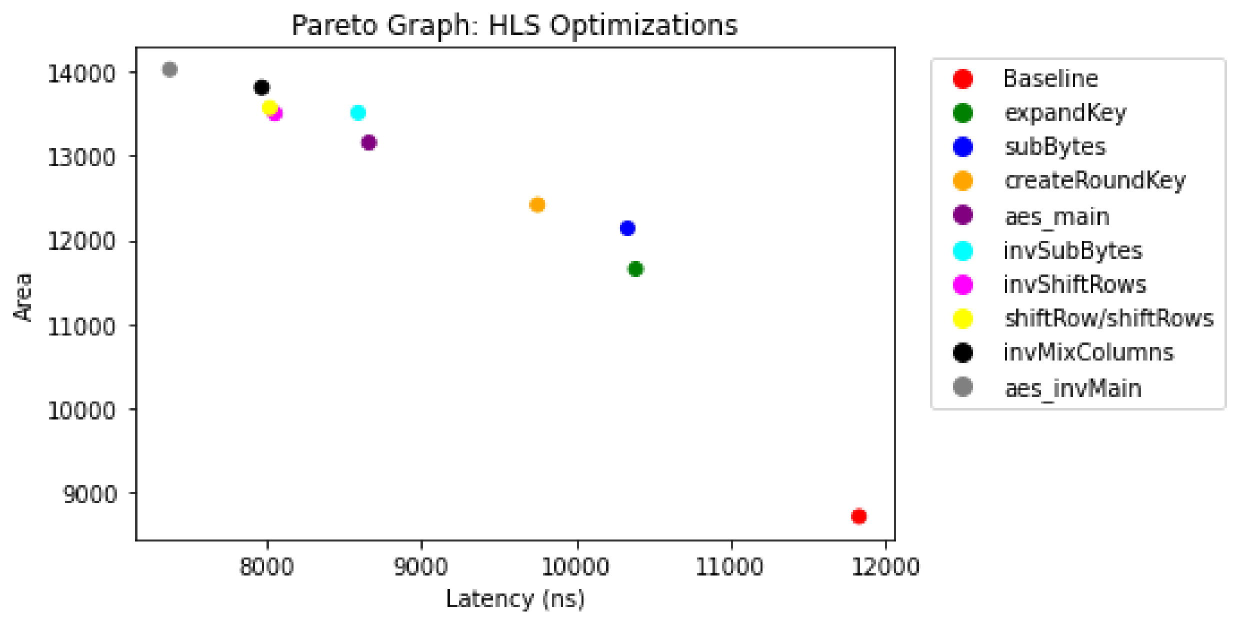


Figure 10. Pareto Graph of HLS Optimizations

Before we decided to compare our hardware and software implementations, we thought it would make much more sense to convert our AES C algorithm into Python. Our thinking was that any future observed hardware and software timing results/differences would be attributed to the platform of implementation instead of the efficiency of the implemented algorithm. After verifying that the new Python software implementation runs the same algorithm as our hardware, we collected timing results of baseline and optimized hardware, as well as our Python and C implementations on the PYNQ-Z2 board.

Algorithm/Implementation	**Encryption & Decryption Time (ms)**
Baseline Hardware: AES-128	6.011
Baseline Hardware: AES-192	15.700
Baseline Hardware: AES-256	11.939
Optimized Hardware: AES-128	1.812
Optimized Hardware: AES-192	1.786
Optimized Hardware: AES-256	1.778
Python Software: AES-128	61.778
Python Software: AES-192	65.350

Python Software: AES-256	79.375
C Software: AES-128	0.043
C Software: AES-192	0.050
C Software: AES-256	0.054

Table 3. AES Timing Results

For the sake of comparison, below is a table of our initial software algorithm’s timing results.

Initial Software AES Implementation	**Time (ms)**
PyCryptodome (CBC AES-256 Encryption & Decryption)	0.801
Initial EBC AES-256 Encryption Algorithm	4.860

Table 4. Initial Software AES Implementation Timing Results

Overall, our optimized hardware implementation had the following average speedups over the following implementations:

AES Implementation	**Average Speedup ($T_{implementation}$ / $T_{optimized\_HW}$)**
Baseline Hardware	6.27338
Python Software	38.43477
C Software	0.02736

Table 5. Optimized Hardware Speedups

After creating a usable bitstream for our power monitor we uploaded it to the hardware and did some initial testing. We used the same Python script that we used when initially testing the board but with a few modifications to write to the power monitor control registers as well. The results of this test found that upon enabling the power monitor we caused the hardware to trigger a reset. While this was not the result we had hoped for, this is promising because it shows that our design does in fact work somewhat. It would seem that there are some safety checks on the board that we hadn’t accounted for. A potential remedy to this would be to add buffers to our design to slow down the speed of the oscillation, thus losing some sensitivity but lowering dynamic power consumption. This should help make the design actually usable.

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 test benches 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

There are a few things that can be improved and added to this project if given more time. The first possibility would be to consider operating on more than just one 128-bit block of plaintext at a time since practical use cases of AES encryption don't just operate on 16 characters. Along the same idea of practical AES usage, implementing other AES modes (such as Cipher Block Chaining (CBC), Cipher Feedback (CFB), Propagating Cipher Block Chaining (PCBC), Output Feedback (CFB), and Counter (CTR)) would be another extension still within the scope of the project's theme.

On the power side channel attack side of things, the next steps are to implement buffers to make our design usable. Once we can reliably get power reading from the board the next step would be to analyze these readings to see if we can distinguish a 1 and a 0, and tune the hardware design accordingly. Once that is done we would create a Python script to analyze the results of the power reading and backtrace the AES algorithm to find the key.

References

[1] M. Parvizi, "m3y54m/aes-in-c," GitHub, Apr. 05, 2024. https://github.com/m3y54m/aes-in-c (accessed Apr. 17, 2024).

[2] NIST, "Federal Information Processing Standards Publication 197 Announcing the ADVANCED ENCRYPTION STANDARD (AES)," 2001. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf (accessed Apr. 17, 2024).

[3] J. Daemen and V. Rijmen, "Note on naming Rijndael Note on naming." Available: https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf#page=1 (accessed Apr. 17, 2024).

[4] Wikimedia.org, 2024 https://wikimedia.org/api/rest_v1/media/math/render/svg/63ac3cf2cb47d5a29c1210fca521f9e4e49e39b1 (accessed Apr. 17, 2024).

[5] "Advanced Encryption Standard," Wikipedia, Mar. 25, 2024. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#/media/File:AES-SubBytes.svg (accessed Apr. 17, 2024).

[6] "Advanced Encryption Standard," Wikipedia, Mar. 25, 2024. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#/media/File:AES-ShiftRows.svg (accessed Apr. 17, 2024).

[7] "Advanced Encryption Standard," Wikipedia, Mar. 25, 2024. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#/media/File:AES-MixColumns.svg (accessed Apr. 17, 2024).

[8] "Advanced Encryption Standard," Wikipedia, Mar. 25, 2024. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#/media/File:AES-AddRoundKey.svg (accessed Apr. 17, 2024).

[9] Wikimedia.org, 2024. https://upload.wikimedia.org/wikipedia/commons/5/50/AES_%28Rijndael%29_Round_Function.png (accessed Apr. 17, 2024).