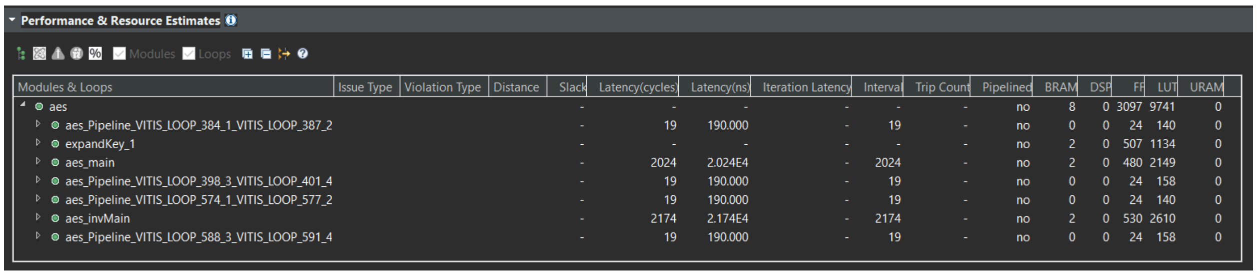
03/28/2024
Alexander Ingare
Hamza Iqbal

Project Update 2 Report

We faced multiple obstacles in our attempt to implement the hardware overlay for AES encryption. To create the hardware aspect of our project, we first had to convert the Python code we wrote for Project Update 1 into C code. Fortunately, the types of operations that AES encryption requires are much easier to accomplish with C. Because of this, we completed our AES implementation (both encryption and decryption modes in C) which originally only had a working encryption mode in Python. To test this new C code, we wrote a simple C testbench in the main function with golden output data files to compare with and verified that encryption worked and that decryption resulted in the original input plaintext. This code is located within our Github repository in /Project_Update_2/C/, where the make command can be used to compile the C code into an executable binary.

The next step in implementing a hardware overlay was to convert our C code into working Vitis HLS C code. As a first attempt, we created an AXI4 interface to transfer char arrays and perform computations on those char arrays. To validate that the AXI4 source files worked, we also made a testbench, similar to the simple C testbench, that also validates the ciphertext and decrypted-text output of our AXI4 code against the same golden data files. After validating the simulation, we also had to validate that it would synthesize properly. Initially, the AXI4 code had numerous Initiation Interval Violations (II Violations). Analyzing the Synthesis Summary and Schedule Viewer with “Go To II Violation” for these violations explicitly showed what Initiation Interval was required to resolve these violations. For every II Violation, we added pragma directives that specified loop pipelining with these new initiation intervals. This constituted our first hardware “optimization”, to get a first baseline successful synthesis.



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	3097	9741	0
aes_Pipeline_VITIS_LOOP_384_1_VITIS_LOOP_387_2				-	19	190.000	-	19	-	no	0	0	24	140	0
expandKey_1				-	-	-	-	-	-	no	2	0	507	1134	0
aes_main				-	2024	2.024E4	-	2024	-	no	2	0	480	2149	0
aes_Pipeline_VITIS_LOOP_398_3_VITIS_LOOP_401_4				-	19	190.000	-	19	-	no	0	0	24	158	0
aes_Pipeline_VITIS_LOOP_574_1_VITIS_LOOP_577_2				-	19	190.000	-	19	-	no	0	0	24	140	0
aes_invMain				-	2174	2.174E4	-	2174	-	no	2	0	530	2610	0
aes_Pipeline_VITIS_LOOP_588_3_VITIS_LOOP_591_4				-	19	190.000	-	19	-	no	0	0	24	158	0

Figure 1. AES AXI4 Synthesis Summary: Performance & Resource Estimates

After exporting the synthesized implementation, we then built the respective IP in Vivado to get the necessary overlay files to upload to the PYNQ board. The corresponding block diagram (below) was simple: The ZYNQ7 Processing System combined with block/connection automation to the AES IP.

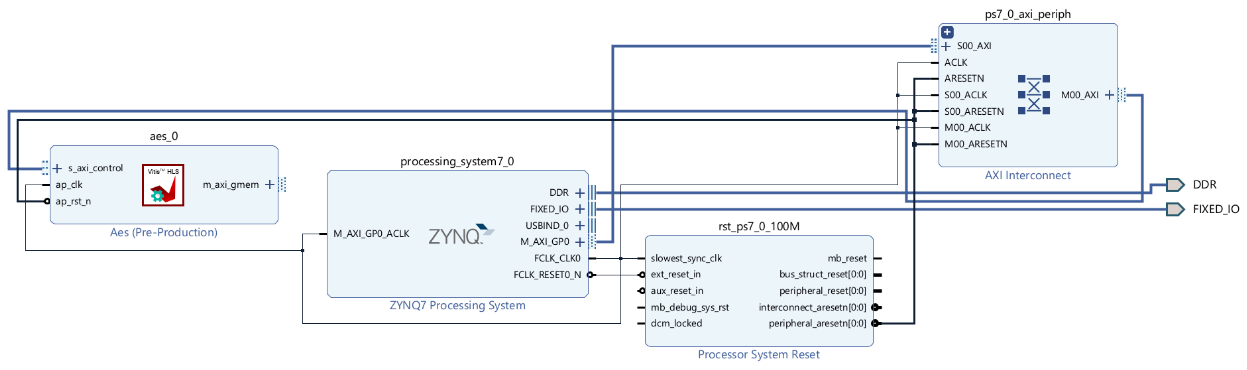


Figure 2. AES AXI4 Vivado Block Diagram

The corresponding device chip layout also had the following area utilization:

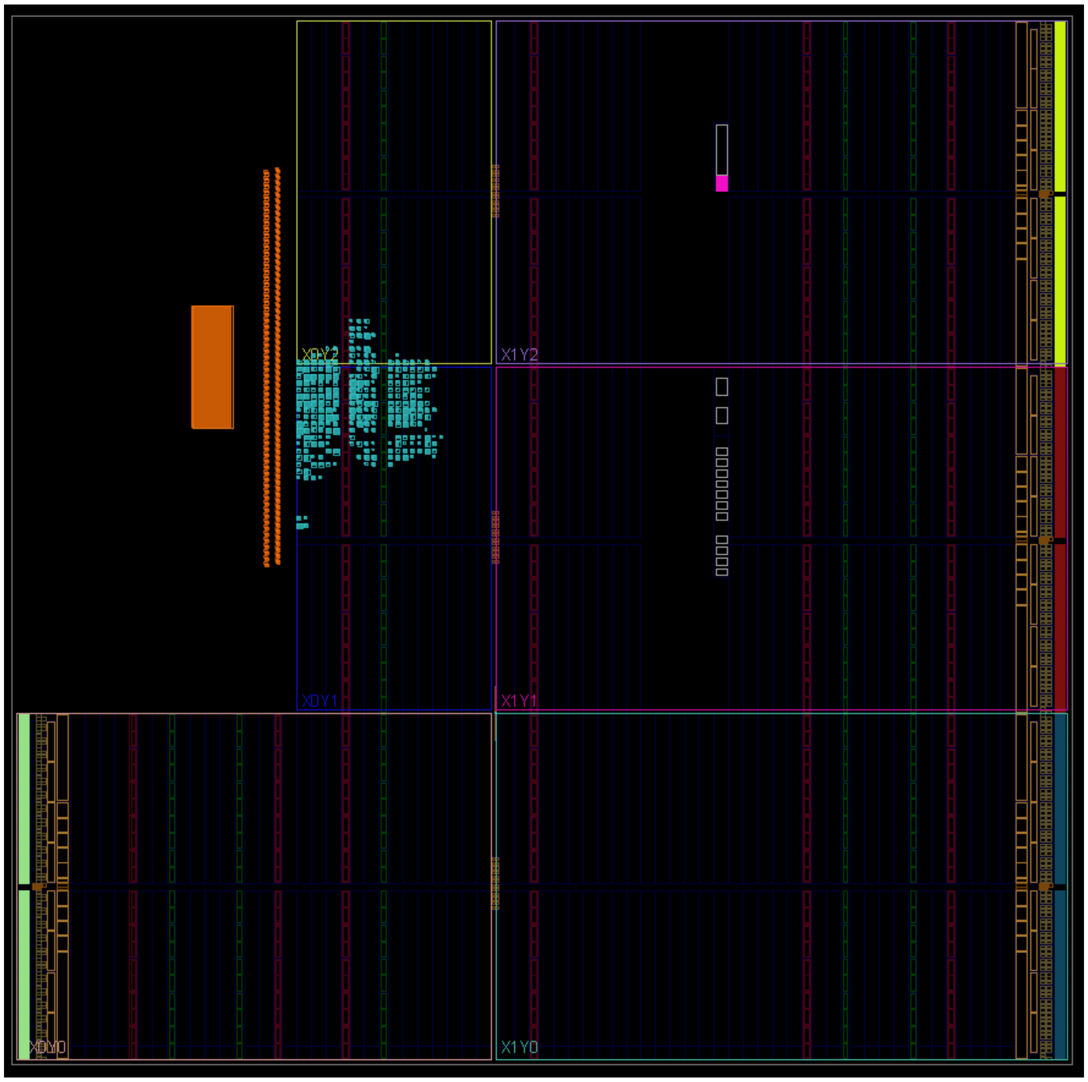
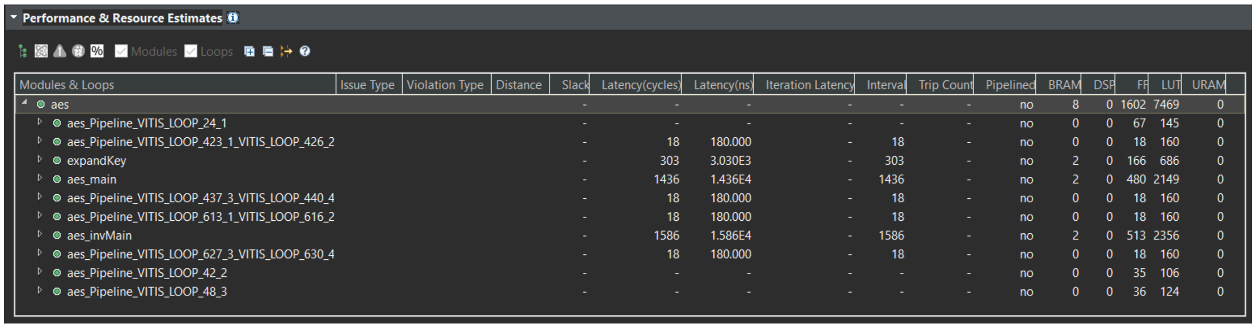


Figure 3. AES AXI4 Vivado Device Chip Layout

However, the first obstacle we encountered was with the actual IP on the board. After uploading the overlay files to the board and running the Jupyter Notebook, the output ciphertext and decrypted text remained all zeros. Even with the help of the TAs verifying that our code “worked”, the cause was still unknown, and we were told to try a new implementation with AXI-Stream instead to try and resolve our issues.

Similarly to the AXI4 implementation, our AXI-Stream implementation in Vitis had a similar testbench that was slightly modified to account for the new interface being used. The source file also had slight modifications in a similar vein. For synthesis, we also added pipelined for loops with preset Initiation Intervals to satisfy the resulting II Violations that appeared for our AXI-Stream 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	8	0	1602	7469	0
aes_Pipeline_VITIS_LOOP_24_1				-	-	-	-	-	-	no	0	0	67	145	0
aes_Pipeline_VITIS_LOOP_423_1_VITIS_LOOP_426_2				-	18	180.000	-	18	-	no	0	0	18	160	0
expandKey				-	303	3.030E3	-	303	-	no	2	0	166	686	0
aes_main				-	1436	1.436E4	-	1436	-	no	2	0	480	2149	0
aes_Pipeline_VITIS_LOOP_437_3_VITIS_LOOP_440_4				-	18	180.000	-	18	-	no	0	0	18	160	0
aes_Pipeline_VITIS_LOOP_613_1_VITIS_LOOP_616_2				-	18	180.000	-	18	-	no	0	0	18	160	0
aes_invMain				-	1586	1.586E4	-	1586	-	no	2	0	513	2356	0
aes_Pipeline_VITIS_LOOP_627_3_VITIS_LOOP_630_4				-	18	180.000	-	18	-	no	0	0	18	160	0
aes_Pipeline_VITIS_LOOP_42_2				-	-	-	-	-	-	no	0	0	35	106	0
aes_Pipeline_VITIS_LOOP_48_3				-	-	-	-	-	-	no	0	0	36	124	0

Figure 4. AES AXI-Stream Synthesis Summary: Performance & Resource Estimates

Since we were using AXI-Stream, we also had to use DMA in our Vivado design to handle the conversion between the memory-mapped input and output ports to the input and output streams. Using the example TCL script “axi-dma-example.tcl” from a previous class tutorial along with our newly made AXI-Stream IP, we created the following block diagram and device chip layout:

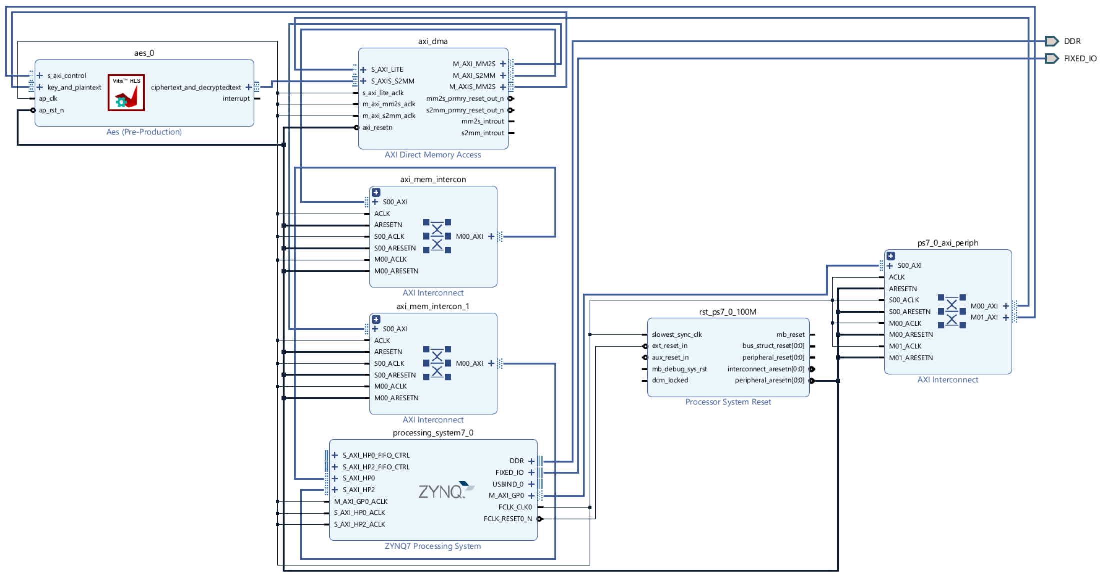


Figure 5. AES AXI-Stream Vivado Block Diagram

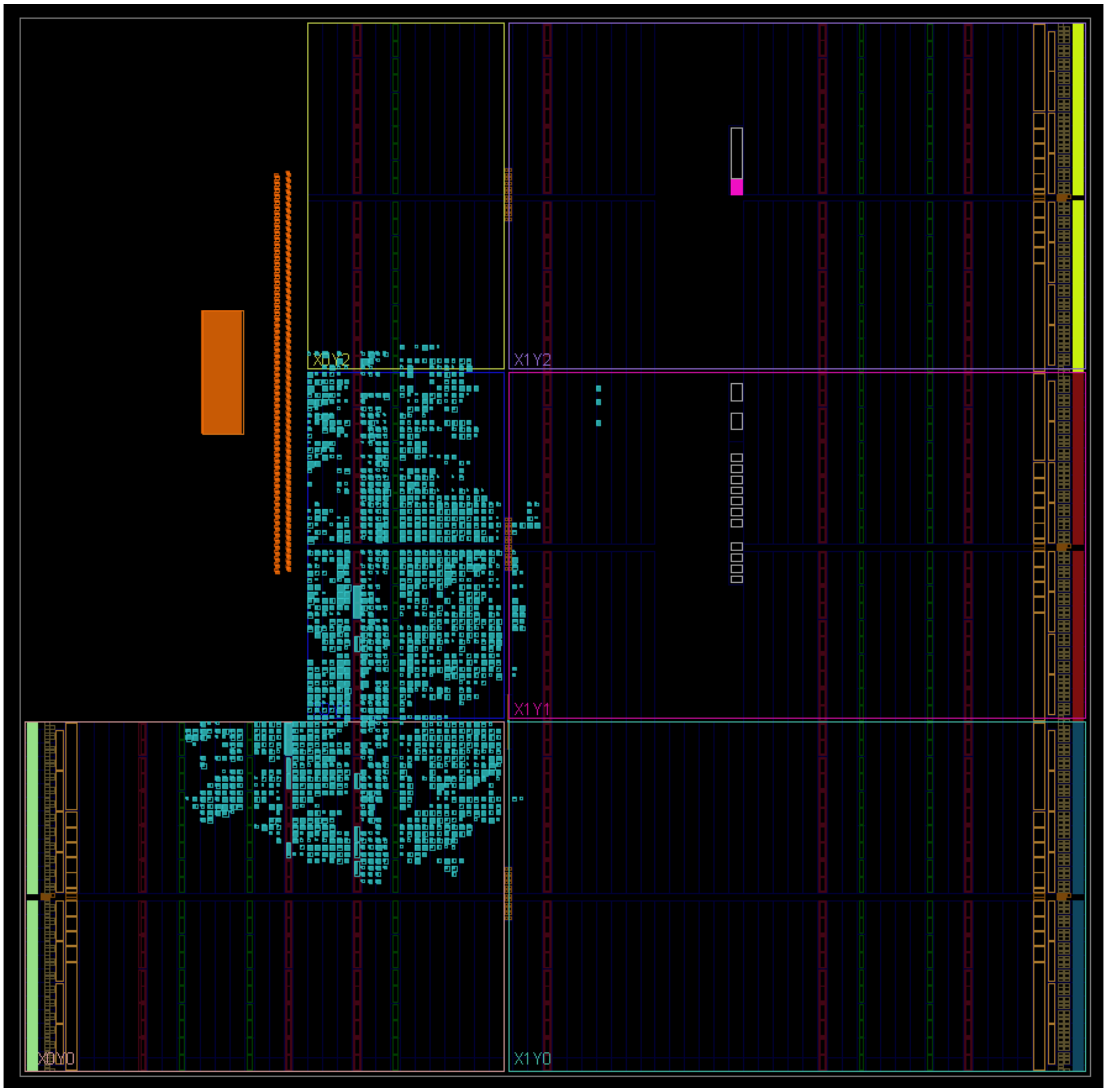


Figure 6. AES AXI-Stream Vivado Device Chip Layout

After generating the bitstream and uploading the overlay files to the PYNQ board as we did with our AXI4 implementation, we hit our second roadblock. The output streams again were reading as all zero. At this point, we started debugging with Yicheng the reason(s) why this was happening. To summarize our findings, our Vivado block design was correct, since a preliminary test with copying the input stream directly to the output stream was successful. However, the moment we tried doing any form of computation, the output stream would never update and would hang forever. We noticed that many other groups faced the same exact problem of output streams being zero when computation was performed, but none of us (including Yicheng) could figure out why. At this point, we had tried various methods of debugging at Yicheng's suggestion, but to no avail: A dedicated Vivado Integrated Logic Analyzer (ILA), test inputs and outputs, adding DMA waits, simple char array copying, etc… For the next two weeks, we will attempt to further optimize our AES circuit and figure out the underlying cause that is making AXI-Stream fail, or go back to our initial AXI4 implementation instead.

In addition to our hardware implementation of AES encryption, we have already begun developing the circuit for the second part of our project. We have created a Verilog hardware description for the ring oscillators and are currently exploring how we can implement many instances of the ring oscillator and use optimal placement and routing constraints for optimal sensing. The plan for the future is to get the Verilog-based power monitor to work with a test bench and characterize its performance with a power-consuming circuit (a series of power-consuming buffers). Once that has been verified, we will turn the Verilog description into a module of our hardware design and implement it along with our AES accelerator. Once that has been achieved we will use the power monitor to capture power traces from the AES accelerator. Once we have singled out a single step where we can see the individual bits we will use that to backtrace and recover the original key.