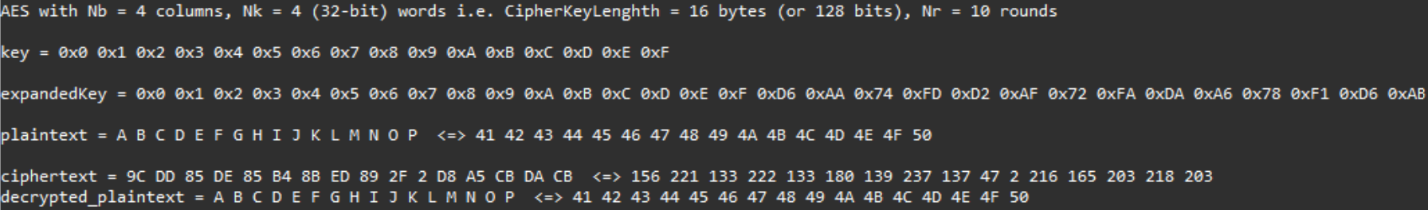
**Encrypted Radio Communication Using FPGAs – Phase 2**

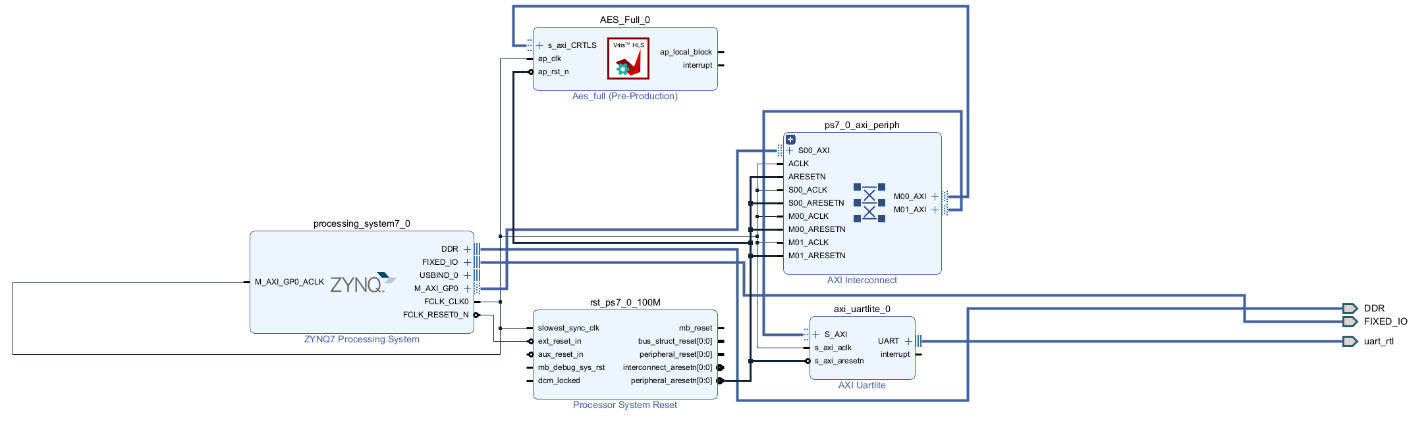
**Project Description**

We focused on extending the existing FPGA-based system by adding a new radio communication block. The existing design already included a processing system (PS) and an AES encryption block, and our task was to integrate the new functionality in a way that allowed communication between all components.

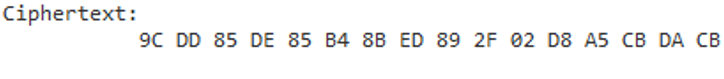


*Figure 1. Vitis HLS AES Logic Test.*

We started by designing and verifying the logic for the radio module using Vitis High-Level Synthesis (HLS). This allowed us to verify the logical functionality of the existing AES design in C/C++ and simulate the behavior (see Figure 1 for verification). After validating the HLS module, we transitioned to Xilinx Vivado, where we incorporated the generated IP core into the existing block design (see Figure 2 for block design). This design included the ARM processing system, AES encryption hardware, and various AXI interconnects.

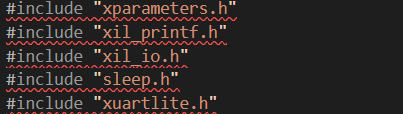


*Figure 2. Generated Block Design with UART Block*



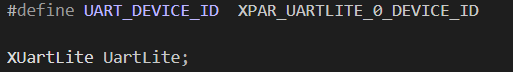
*Figure 3. Ciphertext Generated in Hardware Matches what was Expected from Software Test*

Special attention was required to correctly map the new IP into the existing block interface as appropriate, and to properly assign memory addresses without conflicting with existing peripherals. We exported the .xsa design file to the Vitis IDE through Vivado to verify the hardware design. In the IDE, we developed and tested C code to run on the ARM processor, facilitating communication with the radio block and verifying system behavior on the physical board (see how the output ciphertext in Figure 3 matches what was expected from software test in Figure 1). The test code (AESfunctions.cpp, transmit.c, and receive.c) can be found in the github repo. An explanation of the code can be found below.



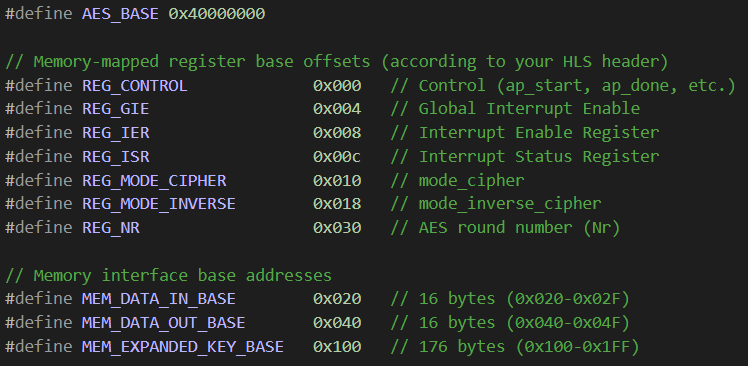
*Figure 4. Module Includes*

In addition to standard C function imports like sleep and printf, the imports used for this module include xparameters.h for address mappings, xil\_io.h for memory-mappped IO operations, and xuartlite.h for UART communication (Figure 4).



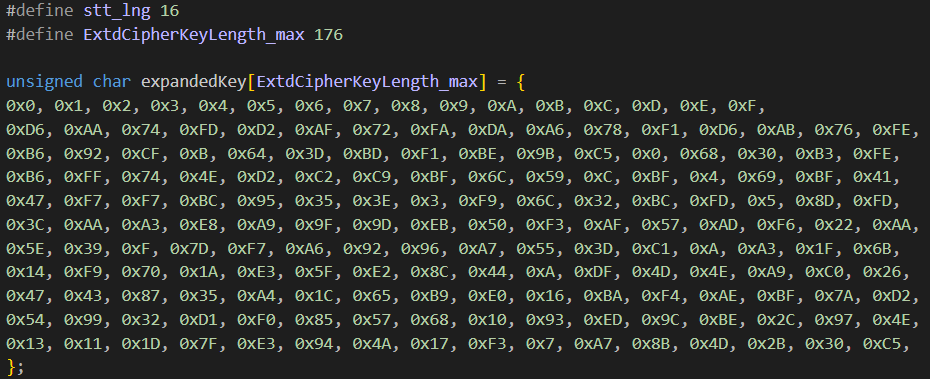
*Figure 5. UART Hardware Peripheral Declaration*

Figure 5 above shows the definition of the device ID for the UARTLite instance, enabling UART communication between the PYNQ board and the HC-12 radio module. UartLite is an instance of the XUartLite driver, representing the UART hardware peripheral.



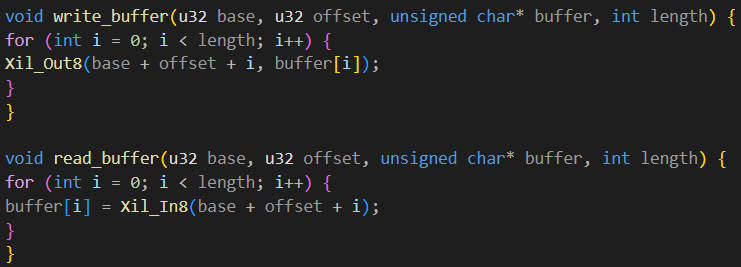
*Figure 6. AES Address Definitions*

Figure 6 shows the address mapping for the AES accelerator. The AES\_Base was found in Xilinx Vivado to be 0x40000000 (Vivado gives you the base address for all peripheral modules after synthesis). Next, the memory-mapped register offsets are defined. Using the open-source AES block as a reference, we found the addresses offsets of he base 0x40000000 as defined above.



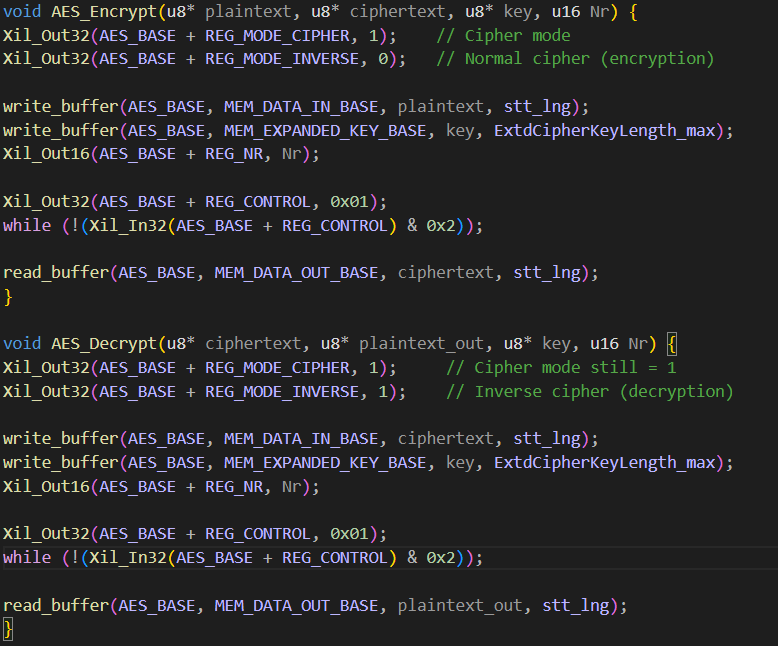
*Figure 7. Expanded Key*

Using the same expanded key as defined in the Vitis HLS C/C++ tester function (see Figure 7), we then attempted to get the same ciphertext from our hardware implementation (this was successful. See Figure 3).



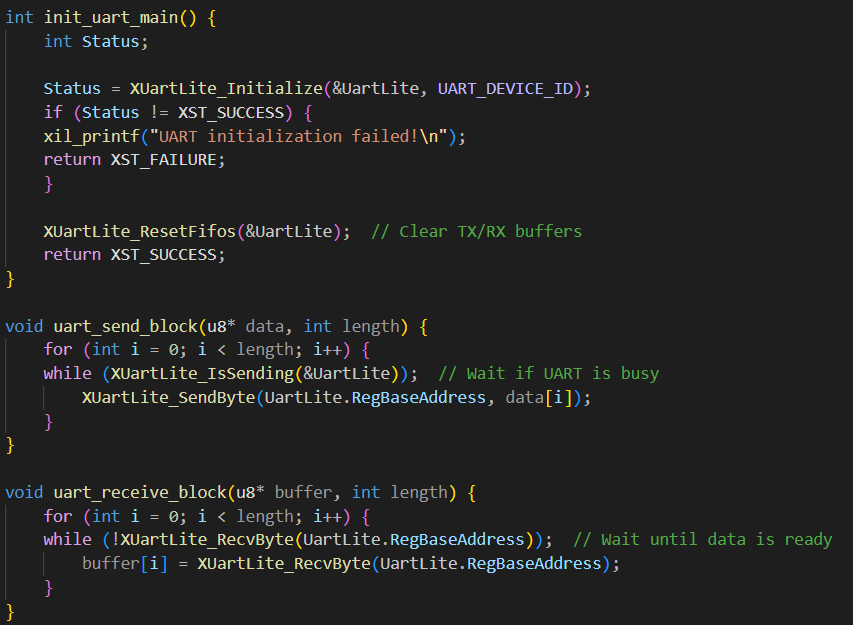
*Figure 8. write\_buffer and read\_buffer functions*

The write and read helper functions (Figure 8) enable writing byte-by-byte into/from memory-mapped hardware registers. These functions, which use the xuartlite.h module for UART communication, take in plaintext or ciphertext and load it into a buffer.



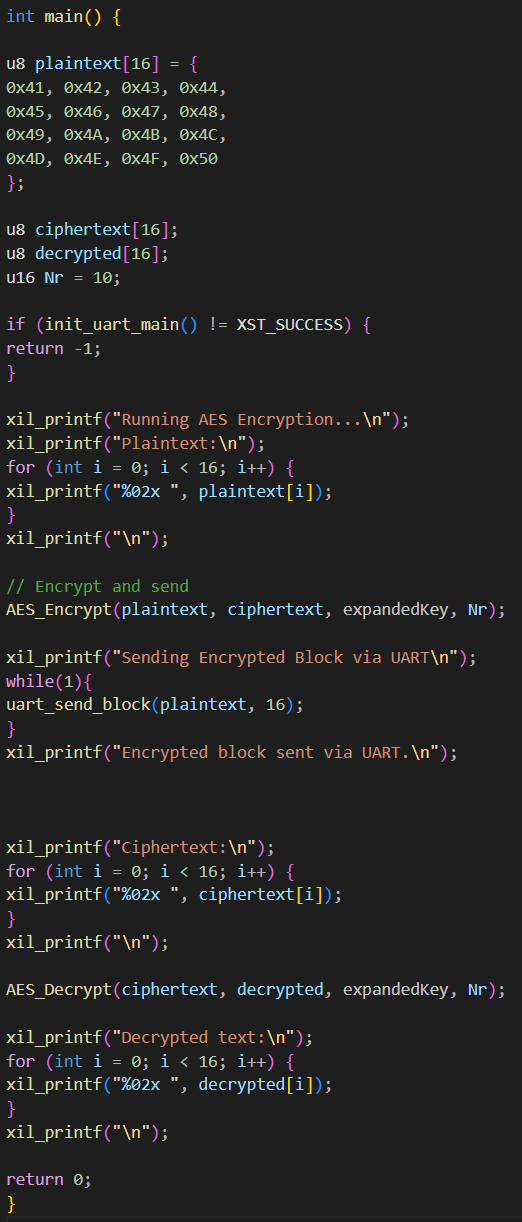
*Figure 9. AES Encryption and Decryption Logic*

The AES\_Encrypt function (Figure 9) sets the encryption mode (cipher mode = 1), then writes the plaintext and expanded key into the appropriate hardware buffers as defined by the constants at the top of the file. The number of AES rounds for this test was set to be 10. The AES core was then started by writing 0x01 to REG\_CONTROL. The status bit is polled until the encryption is complete and then the ciphertext is read back from the output buffer. The AES\_Decrypt function is almost the exact same except that inverse mode is set to 1 at the beginning and inputs are ciphertext while the outputs are the corresponding plaintext.



*Figure 10. UART Transmission and Reception Logic*

The init\_uart\_main() function initializes the UARTLite instance and clears the UART buffers. If initialization fails, the function throws an error. The uart\_send\_block function sends data one byte at a time over UART and waits (polls) if UART is busy. The uart\_receive \_block function, after waiting for data availability, receives data one byte at a time from UART.



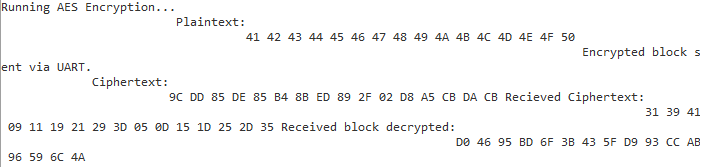
*Figure 11. Whole Main Function for transmit.c*

The main function for the transmit code (Figure 11) prepares a plaintext array of ASCII characters ‘A’ through ‘P’ (0x41 through 0x50). It then defines the output buffers for the generated ciphertext and decrypted text and initializes the UART. It prints the initial plaintext to the console and then encrypts the plaintext using the AES hardware module. Next, it sends the ciphertext via UART. Data is transmitted serially over the UART link to the HC-12 radio which then sends the ciphertext to the other HC-12 radio which is interfacing with another PYNQ board with the same AES hardware.



*Figure 12. Main Function for receive.c*

The receive.c file looks very similar to the transmit.c file with some slight adjustment to its main function (Figure 12). In receive.c, the function receives the ciphertext transmitted over radio and prints it to the console, then decrypts the ciphertext and prints the decrypted text to the console. As of right now, the text received from the radio does not match the ciphertext we sent (see difference between Ciphertext and Received Ciphertext in Figure 13). We know that this is not an error of the encryption block because we already verified that functionality earlier (Figure 3). We assume the issue lies in the way we are transmitting data and plan on resolving this issue by reading the datasheets for the HC-12 radio module more thoroughly for nuances in clocking that may result in poor reception.



*Figure 13. Transmitted Ciphertext and Received Ciphertext Mismatch*

Through this project, we successfully implemented a complete workflow from high-level design to physical deployment on the FPGA board. We demonstrated the ability to:

* Develop and simulate logic with Vitis HLS.
* Integrate and connect new IP blocks using Vivado.
* Configure address maps and AXI interfaces.
* Develop software drivers in Vitis IDE to control hardware.
* Test functionality on physical hardware.

Moving forward into the final phase of this project, we plan on addressing the remaining communication issues which, after being addressed, will allow for a fully functional encrypted radio link between FPGA systems.

**Resources Used**

<https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual>

<https://docs.amd.com/r/2024.1-English/ug1399-vitis-hls/pragma-HLS-expression_balance>

<https://github.com/Xilinx/PYNQ/blob/master/boards/Pynq-Z1/base/vivado/constraints/base.xdc>

<http://venividiwiki.ee.virginia.edu/mediawiki/index.php/ToolsXilinxLabsRTLHLSAES>

<http://venividiwiki.ee.virginia.edu/mediawiki/index.php/ToolsXilinxLabsRTLHLSIP>

<https://testprotect.com/appendix/AEScalc>