Documentation on

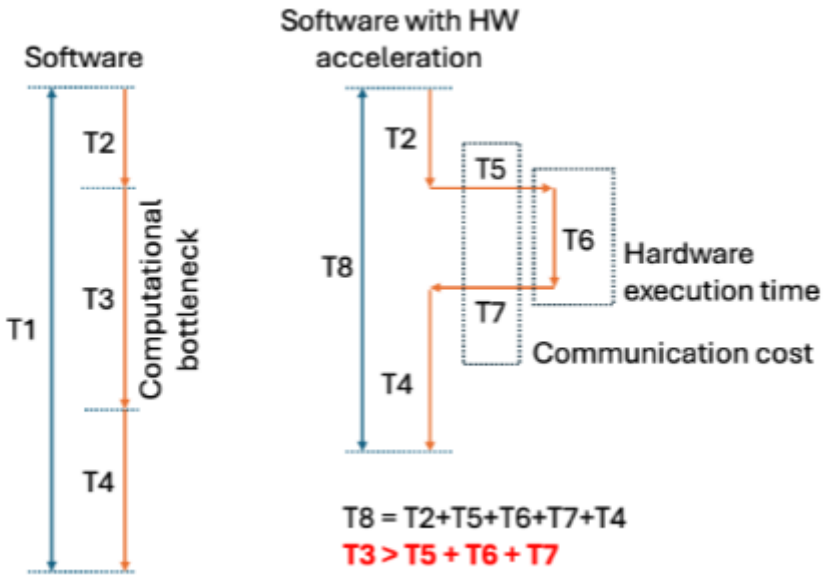
# AES_Core: Hardware Accelerator for AES Cryptographic Algorithm

ECE 510 – Hardware for AI and ML

Neil Austin Tauro - 956372423

## Overview:

AES_Core is a hardware accelerator designed to speed up the enciphering and deciphering operations of the AES (Advanced Encryption Standard) algorithm. The main motivation for developing this project is to offload the computationally intensive components of AES, particularly the block encryption and decryption functions, from a general-purpose CPU to custom digital hardware. This approach significantly improves performance by reducing latency and freeing up processor resources for other tasks. The AES_Core accelerator is implemented in Verilog and is intended for integration into larger digital systems, delivering high-throughput, low-energy, and reliable AES processing.




## Success Criteria:

This project was considered a success based on the following parameters:

- I was successfully able to benchmark and identify bottlenecks in the software algorithm's performance.
- I was successfully able to design, implement, and test the hardware accelerator for AES block cipher operations.

- I was successfully able to simulate the software and hardware together using cocotb, verifying their functional correctness and integration.
- I was successfully able to synthesize the RTL design using OpenLane 2, demonstrating its readiness for eventual chip implementation.

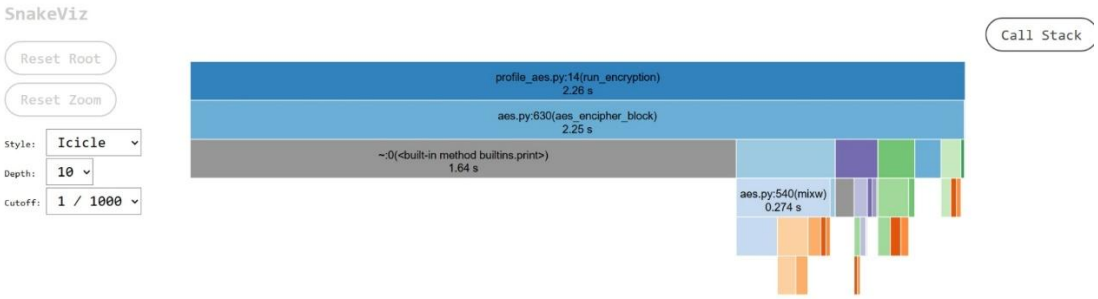
While the current implementation is functional, there are several opportunities for further improvement:

- Implement pipelining within the AES_Core to increase throughput and maximize performance.
- Integrate SPI communication to enable convenient data transfer to and from the accelerator in a larger system-on-chip context.

## Project Workflow - Python:

I started this project by referring to the AES encryption algorithm implemented in secworks' AES repository. To identify performance bottlenecks, I profiled the aes.py script using SnakeViz, this revealed that the aes_encipher_block(self, key, block) and aes_decipher_block(self, key, block) functions were consuming a significant amount of execution time. Specifically, their execution totalled approximately 2.26 seconds for 1000 runs, which translates to 2.26 milliseconds per call.

Looking further into the profile, I found that the mixcolumn() function within aes_encipher_block(self, key, block) was consuming the most processing time. This function operates on blocks of 128 bits with 256-bit keys. The following snapshot from SnakeViz shows the breakdown of execution time.




Search:

| ncalls | tottime | percall | cumtime | percall | filename:lineno(function) |
|---|---|---|---|---|---|
| 30000 | 1.645 | 5.482e-05 | 1.645 | 5.482e-05 | ~:0(<built-in method builtins.print>) |
| 36000 | 0.12 | 3.334e-06 | 0.2741 | 7.614e-06 | aes.py:540(mixw) |
| 1000 | 0.07613 | 7.613e-05 | 2.255 | 0.002255 | aes.py:630(aes_encipher_block) |
| 288000 | 0.07325 | 2.543e-07 | 0.07325 | 2.543e-07 | aes.py:221(gm2) |
| 126000 | 0.06217 | 4.934e-07 | 0.06217 | 4.934e-07 | aes.py:208(w2b) |
| 50000 | 0.05393 | 1.079e-06 | 0.1073 | 2.146e-06 | aes.py:294(substw) |
| 144000 | 0.05335 | 3.705e-07 | 0.08908 | 6.186e-07 | aes.py:230(gm3) |
| 126000 | 0.04839 | 3.84e-07 | 0.04839 | 3.84e-07 | aes.py:199(b2w) |
| 10000 | 0.02871 | 2.871e-06 | 0.05841 | 5.841e-06 | aes.py:600(shiftrows) |
| 10000 | 0.01807 | 1.807e-06 | 0.1077 | 1.077e-05 | aes.py:580(subbytes) |
| 10000 | 0.0164 | 1.64e-06 | 0.03873 | 3.873e-06 | aes.py:352(next_128bit_key) |
| 9000 | 0.0146 | 1.622e-06 | 0.2887 | 3.208e-05 | aes.py:556(mixcolumns) |
| 1000 | 0.01434 | 1.434e-05 | 0.124 | 0.000124 | aes.py:380(key_gen128) |
| 10000 | 0.01302 | 1.302e-06 | 0.01302 | 1.302e-06 | aes.py:504(get_rcon) |
| 11000 | 0.009703 | 8.821e-07 | 0.009703 | 8.821e-07 | aes.py:519(addroundkey) |
| 10000 | 0.00464 | 4.64e-07 | 0.00464 | 4.64e-07 | aes.py:342(rolx) |
| 11000 | 0.002969 | 2.699e-07 | 0.002969 | 2.699e-07 | ~:0(<method 'append' of 'list' objects>) |
| 1 | 0.002477 | 0.002477 | 2.257 | 2.257 | profile_aes.py:14(run_encryption) |
| 1000 | 0.0006152 | 6.152e-07 | 0.0006152 | 6.152e-07 | ~:0(<built-in method builtins.len>) |
| 1 | 1.32e-05 | 1.32e-05 | 1.32e-05 | 1.32e-05 | ~:0(<method 'disable' of '_lsprof.Profiler' objects>) |
| 1 | 8e-07 | 8e-07 | 8e-07 | 8e-07 | aes.py:129(__init__) |

Showing 1 to 21 of 21 entries

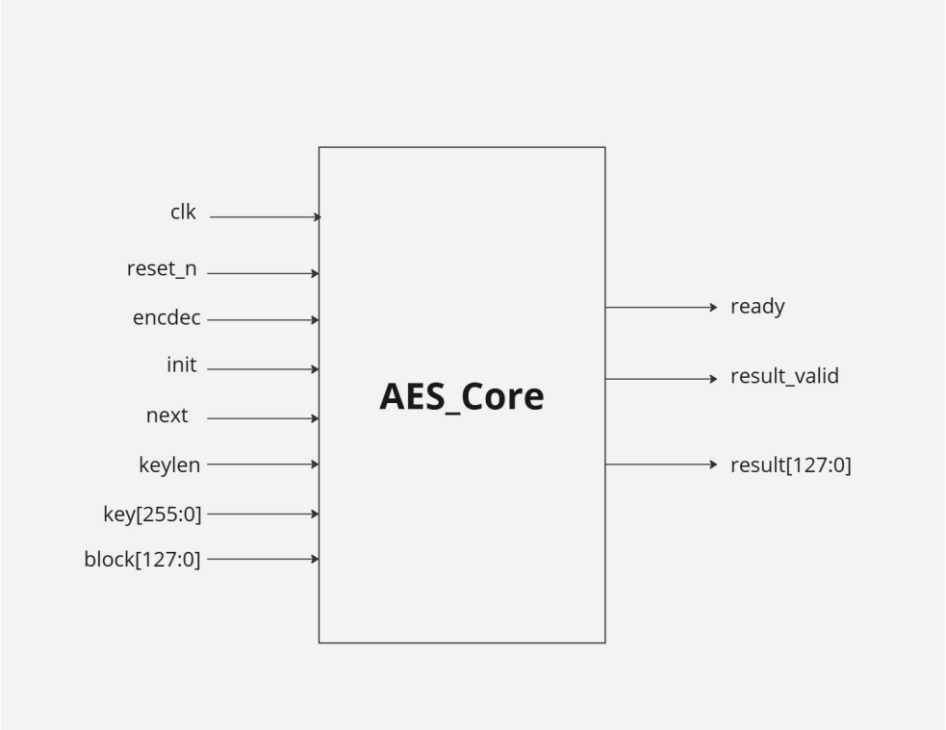
SnakeViz Docs

*Prompt used to generate the profiling script with ChatGPT:*
"Write a Python script to profile aes.py using cProfile and visualize it with SnakeViz."

Initially, I attempted to accelerate the mixcolumn() function in hardware; however, I found this approach was not effective due to its dependency structure and lack of pipelining, which resulted in a bottleneck. Consequently, I decided to accelerate both aes_encipher_block(self, key, block) and aes_decipher_block(self, key, block) instead by designing custom modules in Verilog.

## Project Workflow – RTL and cocotb:

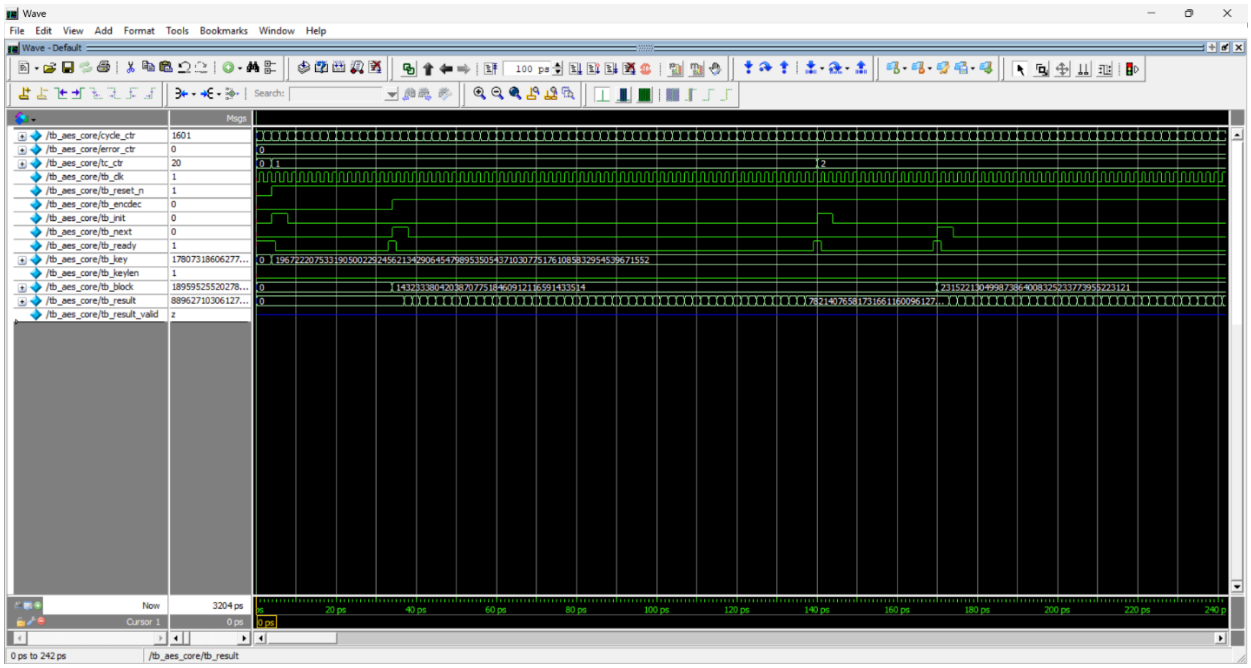
I chose to accelerate the aes_encipher_block(self, key, block) and aes_decipher_block(self, key, block) functions. With the help of ChatGPT (for debugging) and by referring to the secworks' AES repository, I designed the hardware accelerator and implemented it in Verilog as aes_core.v. Below is the architecture of the aes_core:




Signal Descriptions:

- clk: Clock signal for synchronization.
- reset_n: Active-low reset.
- encdec: Selects encryption (1) or decryption (0).
- init: Initializes the AES core with the provided key.
- next: Loads the following plaintext or ciphertext block for processing.
- ready: Indicates when the core is ready to process new data.
- key: The 256-bit cipher key used by AES(uses both 128/256).
- keylen: Flag to enable 256-bit (1) or 128-bit (0) key.
- block: The 128-bit plaintext or ciphertext block.
- result: The 128-bit encrypted or decrypted output.
- result_valid: Flag indicating that the output is valid.

I initially simulated the Verilog design in ModelSim to validate its functionality. The following waveform shows that everything was correct and matching the expected output.




Integration with Software:
To enable AES hardware acceleration from within the software stack, I made the following modifications:

Modified aes.py to aes_hw.py:

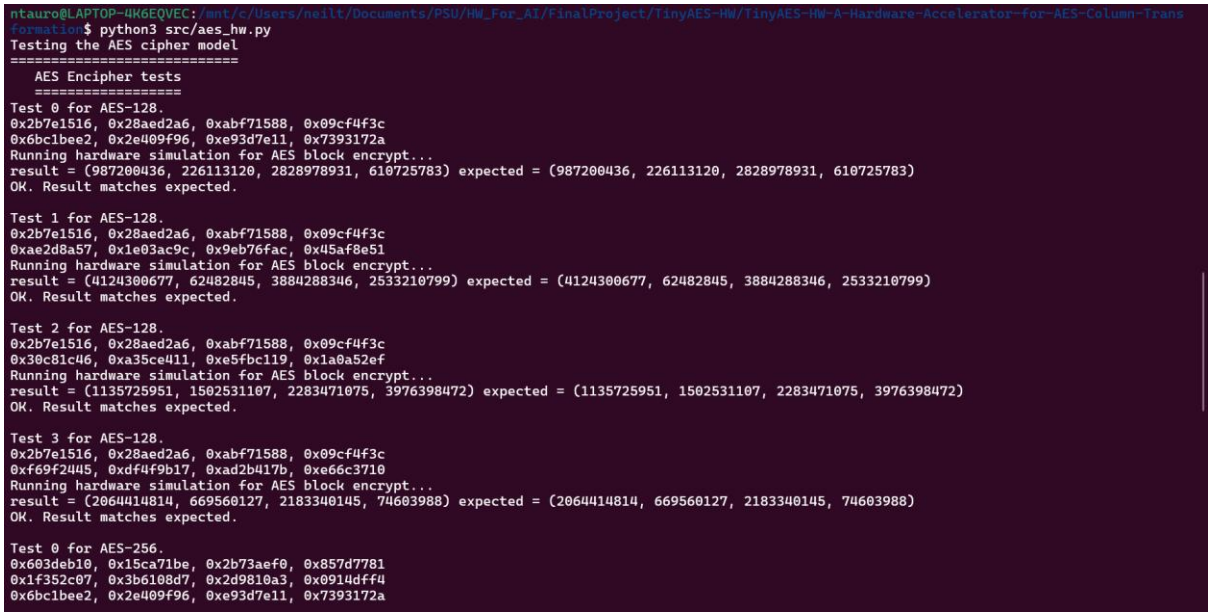
```
#-------------------------------------------------------------------
def aes_encipher_block(self, key, block):
    """
    Encrypt a block using AES in Python or offload to hardware.
    """
    if self.mode == "hardware":
        # Hardware acceleration: call Verilog module via wrapper
        return aes_encrypt_block_hw(key, block)
    else:
        # Original Python implementation
```

Wrapper File (aes_wrapper.py):
I used ChatGPT to generate a wrapper file aes_wrapper.py that would enable communication between the Verilog AES core and the Python script.
*Prompt used to generate wrapper script:*
"Write a wrapper in Python to communicate with Verilog AES Core (with inputs: clk, reset_n, encdec, init, next, ready, key, keylen, block; and output: result, result_valid)"
I then used ChatGPT to further refine this wrapper until it integrated smoothly with aes_hw.py.

cocotb Testbench (tb_aes_hw.py):
ChatGPT also generated a cocotb testbench (tb_aes_hw.py) for verifying the functionality of the Verilog AES core.

With these files in place, the wrapper, the cocotb testbench, and the updated aes_hw.py, I was

successfully able to run aes_hw.py with cocotb and validate its functionality against the software implementation.




Installations and Run:
To run the cocotb testbench with the Verilog AES core, first make sure you have installed the necessary libraries:

*pip install cocotb cocotb-test pytest*

Once the installation is complete, you can run the main script:

*python3 src/aes_hw.py*

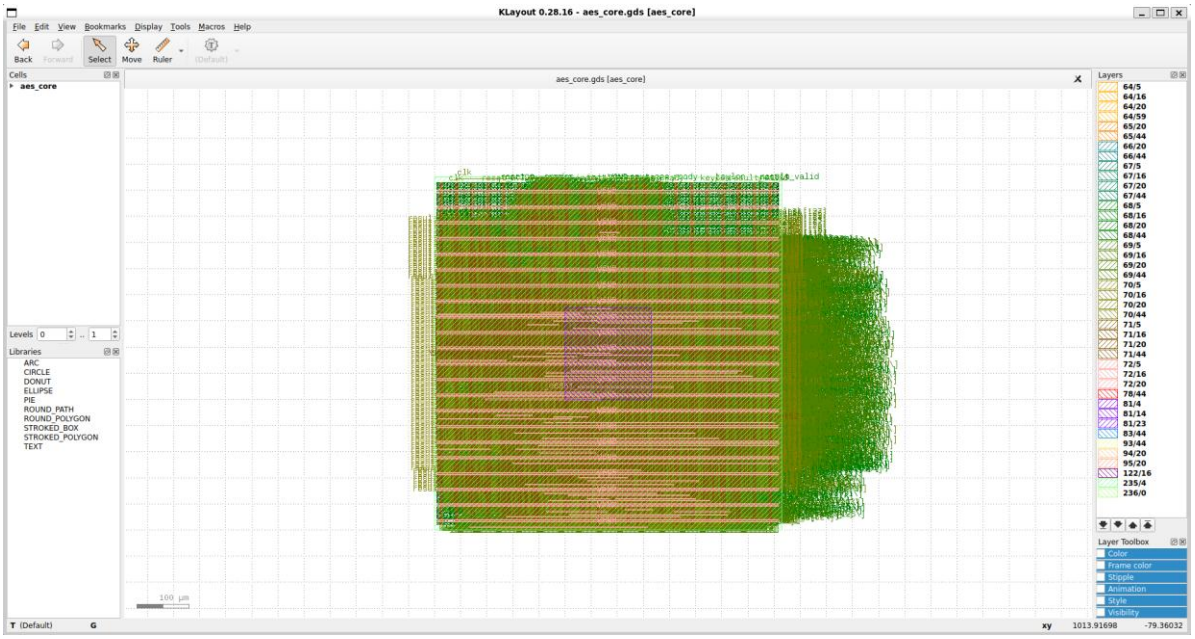
This script will execute the cocotb testbench alongside the Verilog simulator, verifying the functionality of the AES hardware accelerator.

## Synthesis using OpenLane 2:

Initially, I performed the installation and setup of OpenLane 2 following the instructions provided in OpenLane's repository. Once the environment was set up, I used the following command to start the synthesis flow:

*openlane --dockerized ./config.json*

After a successful flow with 74 process stages, OpenLane generated the GDSII file for the AES core. Below is a picture of the GDSII output:




This GDSII file serves as the final physical representation of the chip, reflecting the placement and routing of standard cells for the implemented AES accelerator.

## Synthesis results:

| | |
|---|---|
| Speed | 14.7 ns |
| Area | 201.2 $mm^2$ |
| Power | 0.0348 W |

## Acceleration Analysis:

- Software (Python): 2.26 ms per operation
- Accelerator (Verilog): 14.7 ns per operation

Speed-up:

Speed-up = SW/HW = 2.26ms/14.7ns = 153933

The accelerator is roughly 153933× faster than the software implementation.

But with the integration of SPI Communication:

SPI transfer for 32bits = 2.74 ms

SPI transfer for 256bits = 21.92 ms

So total latency with SPI = (21.92 ms) x 2 + 0.0000147 ms

= 43.84 ms

Speed-up:

Speed-up = SW/HW = 2.26ms/43.84ns = 0.0515

The accelerator + SPI communication is about 0.0515× as fast as the software, meaning software is roughly 19.4 times faster in this case. Hence, SPI slows down the operation.

But with the integration of PCIe Communication:

SPI transfer for 256bits = 8.12 ns

So total latency with PCIe = (8.12 ns) x 2 + 14.7 ns

= 30.94 ns

Speed-up:

Speed-up = SW/HW = 2.26ms/30.94ns = 78086

The accelerator is roughly 78086× faster than the software implementation if PCIe communication is used.

## Related Work:

1. Pipelined AES with Right-Skewed ECA-Based S-box
    a. Authors: Qiang Deng et al.
    b. Key Idea: Proposes a novel S-box for AES using right-skewed Elementary Cellular Automata (ECA), aiming for low-area, high-throughput implementation.
    c. Findings: Achieves up to 35.5 Gbps throughput with 352.073 kGE area and 100.831 Mbps/kGE efficiency, surpassing prior designs in throughput-to-area ratio.
    d. Relevance: This design directly influenced our project's pipelined AES implementation in future work, particularly the use of custom S-boxes and clock-divided pipelining to optimize area and speed.
    e. Link: https://dl.acm.org/doi/pdf/10.1145/3705754.3705760

2. Architecture Design and Hardware Implementation of AES Encryption Algorithm
    a. Authors: Hongling Wei et al.
    b. Key Idea: Focuses on system-level design and FPGA-based implementation of AES using ModelSim and Quartus II for simulation and testing.
    c. Findings: Provides a complete design and simulation framework for AES, validating core functionalities on hardware.
    d. Relevance: Serves as a foundational study for our work, especially in terms of simulation methodology and verification environment for FPGA-based AES hardware.
    e. Link: https://ieeexplore.ieee.org/document/9421585

3. AES Core GitHub Repository
    a. Link: https://github.com/secworks/aes

b. Key Idea: A public repository offering a clean, modular AES core in Verilog, suitable for FPGA and ASIC use.
c. Relevance: Provided reference logic for the hardware development.

## Future Work:

While the current implementation demonstrates a significant speed-up over pure software, there are several improvements that can be made to further enhance its performance and functionality:

- SPI Communication Integration:
  Implement SPI communication between the wrapper (aes_wrapper.v) and the hardware accelerator (aes_core.v) directly on the FPGA. This will enable convenient data transfer and control signals, and allow for a more realistic evaluation of the accelerator's performance in a complete system.
- Pipelining for Higher Throughput:
  Optimize the aes_core.v architecture by adding pipeline stages. This will enable faster processing of consecutive blocks of data and increase throughput, making the accelerator even more effective for high-speed applications.

## Conclusion:

This project successfully demonstrates the implementation of a hardware accelerator for AES-128/256 encryption. The custom accelerator, implemented in Verilog, significantly outperforms pure software execution by reducing latency from milliseconds down to nanoseconds. Our comparison shows a dramatic speed-up, validating the potential for hardware acceleration in high-throughput, low-latency applications.
While the current implementation focuses on functionality and raw performance, there is still room for further improvement. Integrating SPI communication and adding pipeline stages will enable greater data transfer flexibility and even higher processing speeds. Overall, this project serves as a strong foundation for developing high-performance, application-specific processors that can accelerate compute-intensive workloads.