AES module integration with MicroSoc

CEA Leti - LSCO

1 Introduction

This project focuses on the integration of the AES (Advanced Encryption Standard) hardware IP from the OpenTitan project into the MicroSoC platform built around the VexiiRISCV RISC-V core.

The MicroSoC is a minimal yet extensible SoC built on VexiiRiscv, supporting a basic set of peripherals (RAM, UART, etc.) and customizable through SpinalHDL. By integrating the AES IP as a memory-mapped peripheral, this project enables secure, hardware-accelerated encryption and decryption capabilities that can be controlled directly by the VexiiRiscv processor.

This report documents the complete process of integrating the AES module, from cloning and adapting the OpenTitan IP, modifying the AES wrapper for compatibility, incorporating the IP into the MicroSoC hardware design, and building the corresponding firmware to validate functionality. The project also explores the use of Fusesoc for dependency management.

2 AES HW IP Opentitan

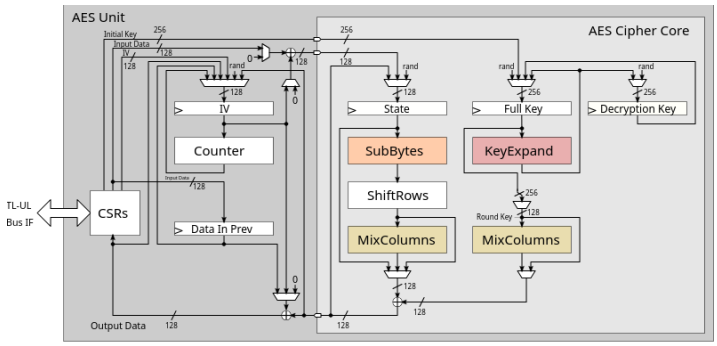

Figure 1: Opentitan HW IP

2.1 Overview of the OpenTitan AES Module

The OpenTitan AES hardware module supports AES-128, AES-192, and AES-256 in ECB, CBC, CFB, OFB, and CTR modes using a single shared data path. The module operates in either encryption or decryption mode at any one time and features internal key expansion to reduce configuration overhead.

Functionality and Operation Round keys are generated on-the-fly from an initial key provided via the register interface. For CBC and ECB decryption, a delay occurs after key updates, during which the decryption key is derived internally. In automatic mode, encryption/decryption starts once all input and IV (if required) registers are written. Manual triggering is also supported via control registers.

The module supports back-pressure: if output data is not read, it stalls until the processor reads the output registers. Input/output access is handled through TL-UL memory-mapped registers.

Architecture The AES core is based on an iterative Equivalent Inverse Cipher architecture with a 128-bit wide data path. Submodules such as SubBytes, ShiftRows, MixColumns, and AddRoundKey are shared across encryption and decryption. The key expansion module (KEM) operates in parallel with the cipher rounds.

Security Features The AES unit includes first-order masking (two shares) and supports multiple S-Box implementations, including Domain-Oriented Masking (DOM) by default. Registers are cleared with pseudo-random data on reset and after use. Shadow registers, sparse encoding, and multi-rail logic are used to mitigate fault injection attacks. Fatal errors result in the module entering a locked state until system reset.

Further Information For more details on architecture, configuration, masking, S-Boxes, and fault injection protection, refer to the official OpenTitan documentation at: `https://opentitan.org/book/hw/ip/aes/index.html`

2.2 Alert Disabling in the Module

While testing the AES IP in isolation, it was observed that the alert output becomes asserted and remains stuck, with the module's output remaining at zero. This suggests that the module has entered an alert state that is never cleared, likely due to the absence of a connected alert receiver that would normally acknowledge and handle such conditions in a full SoC environment.

The underlying cause lies in the behavior of the `prim_alert_sender` module, a generic alert signaling primitive used widely in OpenTitan IP blocks. This module is responsible for encoding alert events—such as security faults, integrity violations, or protocol errors—and sending them over a differential interface to a central alert handler. It uses a four-phase handshake protocol over differential signal pairs (`alert_p/n` and `ack_p/n`) to reliably signal and confirm the presence of an alert.

In addition to test-triggered and latched alerts, the module also detects **signal integrity faults**. If either the `ack` or `ping` differential inputs are incorrectly encoded, the module considers this a fault. In such cases, it drives an intentionally invalid value (`alert_p = 0`, `alert_n = 0`) on its differential output to signal a protocol-level error to the receiver.

However, in standalone testing of the AES IP, no alert receiver is present to acknowledge or reset the alert. As a result, any alert condition leaves the AES module locked in a waiting state, unable to proceed with normal operation.

To work around this during IP-level testing, the alert output signals were forced to `0` . This disables the standard alert signaling and prevents the module from stalling.

3 Using and adapting open titan AES HW IP

3.1 Fusesoc Tool

Fusesoc is a tool designed to manage and retrieve the necessary RTL files for hardware modules. It relies on a .core file to describe each IP's dependencies, source files, and configuration. This is particularly useful for complex IPs like AES, where manually selecting the required RTL files is impractical. For instance, the AES IP from OpenTitan requires over 200 SystemVerilog files, making Fusesoc an essential tool for correct and efficient integration.

3.2 The Correct AES IP Choice

The OpenTitan repository includes multiple AES-related IPs, each tailored for different purposes. Some are intended for simulation using tools like Verilator, while others are fully functional and ready for integration into SoC designs. Among these, two IPs are particularly relevant:

- **aes**: This is the main AES module used in the OpenTitan SoC and is designed to communicate via the TileLink UltraLite (TL-UL) bus. It is directly connected to the Ibex core in the OpenTitan architecture.
- **aes-wrap**: This module provides a top-level wrapper called `aes-wrap` that instantiates the main AES core. It includes an internal finite state machine (FSM) that handles all TL-UL transactions, making it easier to interface with external components. Only a simplified set of input and output signals (such as the AES key, input block, and result) are exposed, abstracting away the TileLink details.

The `aes-wrap` IP was selected for integration with the MicroSoC based on the VexiiRiscv core, as it aligns well with the SoC's peripheral model. In the MicroSoC design, peripheral control is demonstrated through the `PeripheralDemo.scala` file, which serves as a demo to show how to control memory-mapped peripherals via VexiiRiscv software-accessible interfaces. The `aes-wrap` module fits this model perfectly, allowing the processor to write inputs and control signals, and read back the encryption or decryption results directly. Further implementation details will be discussed in the following sections.

Using the following command:

```
fusesoc list-cores | grep aes
```

a list of AES-related hardware IPs from the OpenTitan project is displayed. These IP cores serve various purposes such as simulation, verification, modeling, and hardware integration. While most entries are targeted at design verification environments, some are specifically intended for SoC integration. This command is useful for identifying which AES IP is best suited for a given task.

```
lowrisc:dv:aes_cov:0                    : local : AES functional coverage interface & bind.
lowrisc:dv:aes_env:0.1                  : local : AES DV UVM environment
lowrisc:dv:aes_err_injection:0.1        : local : AES error injection modules and bind file.
lowrisc:dv:aes_model_dpi:0              : local : AES model DPI
lowrisc:dv:aes_ral_pkg:0                : local : <No description>
lowrisc:dv:aes_sim:0.1                  : local : AES DV sim target
```

```
lowrisc:dv:aes_sva:0.1                    : local : AES assertion modules and bind file.
lowrisc:dv:aes_test:0.1                   : local : AES DV UVM test
lowrisc:dv:aes_test_vectors:0             : local : Parse test vector files and output structured data
lowrisc:dv_verilator:aes_cipher_core_tb:0 : local : AES Cipher Core Verilator TB
lowrisc:dv_verilator:aes_sbox_tb:0        : local : AES SBox Verilator TB
lowrisc:dv_verilator:aes_wrap_tb:0        : local : AES Wrapper Verilator TB
lowrisc:fpv:aes_csr_assert:0              : local : <No description>
lowrisc:ip:aes:1.0                        : local : AES unit
lowrisc:ip:aes_wrap:1.0                   : local : AES wrapper for FI experiments
lowrisc:model:aes:1.0                     : local : AES model
lowrisc:model:aes_modes:0.6               : local : AES modes
```

3.3 AES-Wrap HW IP Extraction

The Fusesoc tool not only extracts the necessary RTL files, but also automatically generates a Makefile capable of creating a TCL script for setting up a Vivado project. This Vivado project, which includes all required files for the `aes-wrap` IP, was used as the foundation for integrating the AES hardware into the MicroSoC design.

```
fusesoc run --target=vivado lowrisc:ip:aes_wrap
```

4 Connecting AES-WRAP HW IP to MicroSoc Soc

The MicroSoC platform provides a modular framework for interfacing with custom hardware peripherals. It includes a file named `PeripheralDemo.scala`, which serves as a reference for how to connect and control peripheral IPs using the VexiiRISCV core. This mechanism is leveraged to integrate the OpenTitan-based `aes_wrap` IP as a memory-mapped peripheral.

The integration process begins by creating a BlackBox wrapper in SpinalHDL named `AesWrapBlackBox`, which declares the ports of the Verilog module `aes_wrap`. This BlackBox allows the Verilog IP to be instantiated inside SpinalHDL designs with type-safe, high-level connectivity.

To incorporate the AES hardware into the SoC, a new area named `AesWrapFiber` was defined. This fiber connects to the SoC fabric via a TileLink bus node. Internally, it instantiates the BlackBox and provides logic to map the IP's I/O signals to memory-mapped addresses accessible from the processor.

The `aes_wrap` peripheral was assigned the base address `0x10005000` on the TileLink bus. Each 32-bit register of the AES peripheral—such as input data blocks, key segments, output data, and control signals—was mapped to a unique word-aligned offset from this base. This allows the VexiiRISCV core to control the AES engine by writing to and reading from these memory-mapped registers in software.

Once the hardware was properly connected and memory-mapped, software could be developed to drive the peripheral. By writing input blocks, keys, IVs, and control signals to the mapped addresses, and reading back the results, the processor can perform AES operations via hardware.

4.1 MicroSoc SpinalHDL code modifications

4.1.1 Integration of AES Hardware Wrapper into `microsoc` SoC

To enable hardware-accelerated AES cryptographic operations within the `microsoc` platform based on the VexiiRISCV core, two SpinalHDL modules were added:

- `PeripheralAesWrapFiber` – a tilelink-mapped peripheral that acts as a bridge between the TileLink bus and the AES wrapper logic.
- `AesWrapBlackBox` – a blackbox instantiation of the Verilog AES wrapper module (`aes_wrap.sv`).

PeripheralAesWrapFiber This module defines a new peripheral area connected to the TileLink bus. It exposes registers for AES input data, key, IV, mode selection, and output. The logic is implemented within a `Fiber` block to allow flexible, lazy instantiation.

The core functionality is defined inside a `Component` named `PeripheralAesCore`, which includes:

- Register-mapped access to the AES inputs: 128-bit input block, 256-bit key, 128-bit IV.
- Output access to the 128-bit AES result and a data-valid signal.
- Control register for mode selection (`decrypt_mode`) and start trigger.
- Address decoding with support for both read and write operations, using a memory-mapped interface.

In addition, the `start` signal is implemented as a memory-mapped register (`startReg`) that allows software to control the AES operation. It is not edge-sensitive and does not enforce a rising edge; the Verilog wrapper reacts to the level of the signal as managed by software.

AesWrapBlackBox The `AesWrapBlackBox` module instantiates the Verilog AES FSM wrapper (`aes_wrap.sv`) inside SpinalHDL as a blackbox:

```
class AesWrapBlackBox extends BlackBox {
  setDefinitionName("aes_wrap")
  ...
}
```

This module connects:

- Clock and reset lines (`clk_i`, `rst_ni`)
- Input signals: `aes_input`, `aes_key`, `aes_iv`, `start`, and `aes_decrypt_i`
- Output signals: `aes_output`, `data_valid`, and alert lines (`alert_recov_o`, `alert_fatal_o`)

The blackbox declaration uses `noIoPrefix()` to ensure that the Verilog module's port names are used directly without a prefix (io.), ensuring compatibility with the original RTL.

Integration into the SoC These additions make the AES module accessible via the TileLink bus of the SoC, allowing software running on the VexiiRISCV CPU to:

- Configure AES keys and IV
- Write input ciphertext or plaintext
- Set the operation mode (encryption/decryption)
- Trigger the operation
- Read the resulting output and detect when it is valid

Address Mapping The AES wrapper is accessed via 32-bit aligned addresses, with each 128-bit value split across 4 consecutive registers. The layout aligns with the software driver as follows:

Signal	Address Offset (Hex)
`aes_input[0]`	0x001
`aes_input[1]`	0x002
`aes_input[2]`	0x003
`aes_input[3]`	0x004
`aes_key[0]`	0x005
...	...
`aes_key[7]`	0x00C
`aes_output[0]`	0x00E
...	...
`aes_output[3]`	0x011
`data_valid`	0x012
`decrypt_mode`	0x013
`aes_iv[0]`	0x014
...	...
`aes_iv[3]`	0x017
`start`	0x019

4.2 AES OpenTitan FSM Changes

The AES wrapper module from the OpenTitan project was modified to improve testability, control, and support for various AES modes. The following key changes were made to the FSM:

1. **Introduction of a `start` Signal:** A new input signal, `start`, was added to explicitly trigger the FSM from software after input data is loaded. This ensures precise initiation of the AES operation.
2. **Rising Edge Detection on `start`:** To avoid multiple unintended FSM cycles due to slow deassertion of the `start` signal, a one-cycle pulse (`start_pulse`) was generated using edge detection logic (`start_r` and `start_r_prev`).

3. **Latched Start Control:** A flag `start_latched` was introduced to retain the start condition once asserted. Initially, the start pulse lasted only one clock cycle, which was too short to be reliably detected when the FSM was in the `IDLE` state, causing it to be missed. By latching the signal, it remains asserted throughout the entire duration of the `IDLE` state, ensuring the FSM consistently detects the trigger and initiates exactly one AES operation per request.

4. **Extended Mode Support (CTR, CBC):** Inputs `aes_iv` (initialization vector) and `aes_decrypt_i` (decryption mode selector) were introduced, allowing the wrapper to support modes beyond ECB, such as CTR.

5. **IV Handling in FSM:** The FSM was updated to properly load the initialization vector (IV) data into the AES core when required by the selected mode. In the original implementation, the IV was hardcoded to zero and not even connected as an input to the `aes_wrap` module, effectively causing it to be ignored. The modifications ensure that valid IV data is now provided to the core as needed.

6. **Improved FSM Restart Behavior:** The FSM now returns to the `IDLE` state after completing an operation instead of staying in `FINISH`. This change enables repeated AES operations without requiring a reset.The FSM only restarts when a new input is received, as explained with the `start` protocol earlier, ensuring one operation per distinct trigger.

7. **Data Valid Output Signal:** A new output, `data_valid`, was introduced to indicate when a valid AES output is available. This signal is critical for ensuring proper synchronization with downstream modules, such as the input of the FIFO, which uses it to enable writing only when the AES output is fully processed. Without this mechanism, the same data could be written multiple times, potentially corrupting the configuration when interfacing with components like ICAP.

8. **Debugging Support (Optional):** An ILA instance (`ila_3`) was temporarily added for debugging internal FSM signals like `data_valid`, `start_pulse`, and state transitions.

4.3 AES Controller Software Description

The following software interacts with a memory-mapped AES hardware module. It is designed for testing and operating the AES engine. The software performs key loading, IV setup, UART input reading, AES triggering, and output retrieval.

Memory Mapping and Initialization The AES peripheral is memory-mapped starting at address `0x10005000`, with various registers defined as offsets. The base addresses are mapped to volatile pointers for read/write access:

```
constexpr std::uintptr_t AES_INPUT_BASE   = 0x10005004;
constexpr std::uintptr_t AES_KEY_BASE     = 0x10005014;
constexpr std::uintptr_t AES_OUTPUT_BASE  = 0x10005038;
constexpr std::uintptr_t AES_DONE_REG     = 0x10005048;
constexpr std::uintptr_t AES_MODE_REG     = 0x1000504C;
constexpr std::uintptr_t AES_IV_REG       = 0x10005050;
constexpr std::uintptr_t AES_START_REG    = 0x10005064;
```

In embedded systems, peripherals like the AES engine are accessed through memory-mapped I/O. Each hardware register is assigned a fixed memory address. To interact with these registers in software, the base address (e.g., `AES_INPUT_BASE`) is cast to a pointer:

```
volatile uint32_t* aes_input = reinterpret_cast<volatile uint32_t*>(AES_INPUT_BASE);
```

- `reinterpret_cast` converts the integer address to a pointer.
- `uint32_t*` means the hardware is accessed as 32-bit words.
- `volatile` tells the compiler not to optimize away reads/writes, which is critical since hardware can change these registers independently.

This lets the software write and read directly to and from hardware using standard memory access syntax.

AES Key and IV Loading A 256-bit AES key and a 128-bit IV are preloaded into the corresponding registers:

```
uint32_t key[8] = { 0x11111111, ... };  // 256-bit key
for (int i = 0; i < 8; ++i)
    aes_key[i] = key[i];

uint32_t iv[4] = { 0x33333333, ... };  // 128-bit IV
for (int i = 0; i < 4; ++i)
    aes_iv[i] = iv[i];
```

AES Mode Setup The AES module mode register is configured for decryption:

```
*aes_mode = 1;
```

This enables decryption mode, which is important for modes other than CTR where direction matters.

UART Interface and Input Reading The software enters an infinite loop, waiting for ciphertext blocks from the UART. Each block begins with a header byte (`0xAA`):

```
std::byte header;
soc::uart.read(std::span(&header, 1));
if (header != std::byte(0xAA)) continue;

std::byte buffer[16];
soc::uart.read(std::span(buffer));
```

Once the data is received, it is packed into 32-bit words and written to the AES input registers:

```
for (int i = 0; i < 4; ++i) {
    aes_input[i] =
        (buffer[i * 4 + 0] << 0)  |
        (buffer[i * 4 + 1] << 8)  |
        (buffer[i * 4 + 2] << 16) |
        (buffer[i * 4 + 3] << 24);
}
```

Triggering the AES Operation The AES computation is initiated using a rising edge on the `start` signal:

```
*aes_start = 1;
for (volatile int i = 0; i < 100; ++i);  // short delay
*aes_start = 0;
```

This ensures that the hardware FSM detects a clean rising edge.

Reading the Decrypted Output After the computation, the output is read from AES output registers:

```
uint32_t output[4];
for (int i = 0; i < 4; ++i)
    output[i] = aes_output[i];
```

Output Display The decrypted result is formatted into hexadecimal and printed over UART:

```
print_hex("Decrypted : ", output, 4);
```

Looping Behavior The loop then restarts, waiting for another ciphertext block over UART.