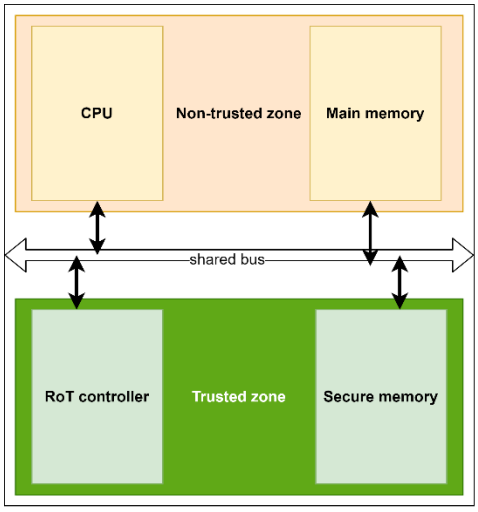
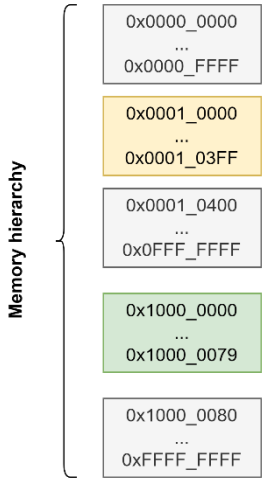
IAS0630 – Final project

For the final project, your task is to build a Root of Trust (RoT) unit that implements a series of security features. The RoT is part of a larger system as shown below:



The system has a shared bus, which is something that has to be taken into account.

The CPU on the non-trusted zone can read and write to the main memory freely. However, it can only read from some parts of the secure memory and it can only write to some parts of the secure memory. The RoT never writes or reads from the main memory. Furthermore, the addressing space is shared across the entire hierarchy. The system is based on a 32-bit architecture. Both memories assume 32 bits per address. The main memory contains 1024 addresses. Since the architecture is 32-bit, the entire address range is `0x00000000-0xFFFFFFFF`. The main memory takes addresses `0x0001_0000-0x0001_03FF`. The secure memory takes addresses `0x1000_0000-0x1000_0079`. The other regions (highlighted in gray) belong to other parts of the system and have no functionality related to the project.



Another aspect that has to be emphasized is that the secure memory is not really a memory. You are supposed to implement the storage of various security assets (keys, random bits, ciphertext, plaintext, etc.) in registers that are distributed across the trusted zone. This technique is called memory mapping of registers. For instance, one of the security features you will be required to implement is to have support for AES-128. Meaning that you will need registers that will hold 128-bit input key, input plaintext, and output ciphertext. These registers exist in the secure "memory" but are not really a memory.

The table below defines the security-related addresses within the Secure memory. All addresses start from `0x1000_0000`. The last two columns determine what type of access is possible (from the point of view of the CPU):

Address offset	Purpose	Read access	Write access
00	**Special**: Status register	**Y**	**N**
01-04	AES key	**Y**	**Y**
05-08	AES plaintext	**N**	**N**
09-12	AES ciphertext	**Y**	**N**
13-44	PUF signature	**N**	**N**
45-76	Encrypted PUF signature	**Y**	**N**
77-80	TRNG bits	**Y**	**N**
81	FSM bits	**N**	**Y**
82-126	No purpose	**N**	**N**
127	**Special**: Operation register	**Y**	**Y**

As can be seen on the table, the PUF signature takes 32 addresses and is 1024 bits long. This means that your PUF must generate 1024 bits of signature. Similarly, for the TRNG, there are four addresses reserved, implying that the TRNG will be asked to generate 128 bits. For AES-related registers, the size is always 128 bits for key and texts.

If the CPU attempts to read from an address it is not supposed to have access to (e.g., 0x1000_0005), the RoT should return all zeros. If the CPU attempts to write to a protected address, nothing should happen, the memory content should not change.

The status register and the operation register have very **special** purposes. The status register is a read-only 32-bit address that the CPU uses to understand what the RoT is currently doing. The meaning of the 32 bits of the status register is as follows: (MSB<->LSB)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

Bit 31: AES_dirty

Bit 30: PUF_dirty

Bit 29: TRNG_dirty

Bits 28-27-26: TRNG_count

Bits 25-06: Not allocated. Can be repurposed as temporary flags safely.

Bit 05: AES_key_loaded

Bit 04: AES_busy

Bit 03: PUF_busy

Bit 02: TRNG_busy

Bit 01: FSM_busy

Bit 00: ROT_busy = (AES_busy or PUF_busy or TRNG_busy or FSM_busy or multiple_PUF_uses)

On reset, the entire status register is zeroed.

The AES_key_loaded bit goes to 1 when the AES key has been loaded entirely, meaning that **consecutive** writes to the AES key addresses have happened in order, one after the other, without any gaps. Meaning, consecutive writes to addresses `01 02 03 04`.

The AES_dirty bit indicates that an AES encryption has already been performed.

The AES_busy bit goes to 1 when the AES encryption is active. It goes to 0 when the AES encryption is not performing.

The PUF_busy bit goes to 1 when the PUF is generating its signature. It goes to 0 when the generation is done.

The PUF_dirty bit indicates that a PUF generation has already occurred.

The TRNG_busy bit goes to 1 when the TRNG is generating random bits. It goes to 0 when the generation is done.

The TRNG_dirty bit goes to 1 when a generation of random bits has already been performed.

The bits TRNG_count keep track of how many times random bits where requested. Every time the operation *op_trng_gen* is executed, the TRNG_count counter should be incremented. If more than 5 *op_trng_gen* operations have been executed, no further *op_trng_gen* operations are allowed and the counter should not be incremented. However, if the counter is cleared by a reset, more TRNG operations are allowed.

The FSM_busy bit indicates that the process of deobfuscating the FSM is taking place. It goes back to 0 as soon as the deobfuscation process is finalized.

Finally, the ROT_busy bit indicates that any operation is currently taking place. Furthermore, if more than one *op_puf_gen* operation is requested, the ROT_busy bit should go to 1 and stay 1. It can only be cleared by a reset.

Regarding the operation register, it does not require actual storage, it is a **shadow address** to which the CPU can write to in order to indicate what operation the RoT should be doing. The list of supported operations is rather small:

op_nop (32'h0000), *op_fsm* (32'h0111), *op_status_clear* (32'h0222), *op_aes_run* (32'h000B), *op_aes_clear* (32'h000C), *op_puf_gen* (32'h1000), *op_puf_clear* (32'h1111), *op_trng_gen* (32'h2000), and *op_trng_clear* (32'h2111).

op_nop is a "do nothing" operation.

op_fsm instructs the RoT to apply the deobfuscating sequence (more details are given later). The sequence cannot be rejected at any intermediate clock cycle, otherwise it would become a timing side channel. This operation **must** be time-constant.

op_status_clear clears all bits of the status register except the ones related to tracking of activity (dirty bits and TRNG_count).

op_aes_run tells the RoT to start AES encryption. The CPU does not tell which data has to be encrypted, the RoT manages the data. Notice that the CPU cannot write to AES plaintext addresses 05-08. This becomes clear later when use cases are described.

ops_aes_clear tells the RoT to clear the AES key memory locations.

op_puf_gen tells the RoT to start generating the 1024-bit long PUF. It can be called many times, but should only execute **once**.

op_puf_clear tells the RoT to clear PUF signature. This operation does not affect the encrypted PUF signature which should be kept, i.e., addresses 13-44 are cleared but 45-76 are kept. Because these addresses are not really a memory, they can be cleared in a single clock cycle.

op_trng_gen tells the RoT to start generating 128 random numbers.

op_trng_clear tells the RoT to clear the TRNG bits registers.

Use cases and testbenches

There are two big use cases that will be utilized to validate your code. For each use case, a separate testbench will be provided. However, these testbenches will only be provided when your project is being graded. If your code does not pass the testbenches, you will still be given a chance to fix your code. If you check the schedule, you will notice that there are two dates marked for project presentation, one being a backup date exactly for this reason.

The first use case goes as follows (testbench#1):

1. CPU configures FSM
2. CPU waits until RoT comes out of busy state
3. CPU loads AES key into the RoT by writing to the respective addresses
4. CPU waits until RoT says AES key has been loaded
5. CPU instructs the PUF to generate a 1024-bit signature
6. CPU waits until RoT comes out of busy state
7. CPU will make 8 calls to AES encryption. Between separate calls, the CPU waits until RoT comes out of busy state. The RoT understands that on each call, 128 bits of the PUF are to be encrypted at a time and stored on the respective registers
8. CPU reads the 1024 bits related to the encrypted PUF signature, 32 bits at a time

The second use case goes as follows (testbench#2):

1. CPU configures FSM
2. CPU waits until RoT comes out of busy state
1. CPU instructs the TRNG to generate 128 bits of random data
2. CPU waits until RoT comes out of busy state
3. CPU reads all 128 TRNG-generated bits, 32 at a time

Each use case will be a separate testbench. Furthermore, a third testbench (testbench#3) will be utilized. This testbench attempts to violate all security features and guaranties that the RoT should implement. In other words, it tries to execute only bad behavior, which the RoT should reject/respect accordingly.

Recommendation: try to run tb2.v first; it is the simplest of the testbenches. Then try tb1.v. Only then try tb3.v, which is longer and more complex than the other two.

Grading

Your project will be graded according to how many of the expected security features (SF) you have implemented correctly.

SF1: Integration with the main CPU that respects the security boundaries, i.e., isolation between the non-trusted zone and the trusted zone. This implies that testbenches #1, #2, and #3 must execute successfully.

SF2: The RoT must contain an **integrated** PUF/TRNG unit that is based on ring oscillators (adapted from lab). The ring oscillators are supposed to be reused for both PUF and TRNG purposes. The PUF must generate a signature with 1024 bits by using the counter-based strategy explained in our lecture (see lecture 8, slide 21). The TRNG should generate 128 random bits at a time. For the TRNG architecture, a simple sampling strategy is sufficient (see lecture 9, slide 22).

-> For the PUF, the expectation is that calling it twice will give the same response. For the TRNG, calling it twice should give different responses. Since you cannot call the PUF twice due to the RoT restriction, you can show the PUF and TRNG behaviors by simulating them separated from the RoT. Suggestion: code the PUF/TRNG in the same Verilog module. You have to demonstrate the quality of the TRNG/PUF by **building your own testbench.**

SF3: The RoT must support AES-128 (https://ati.ttu.ee/~spagliar/teaching/ias0630/labs/aes.zip) (we have used AES-128 in lab 4). We are only interested in AES encryption, there is no need to support decryption. This is validated with testbench #1.

SF4: The RoT should contain an obfuscated FSM (adapted from lab). The 32 bits provided in “FSM bits” should be applied, one by one, MSB to LSB, to unlock the FSM. Meaning that the unlock sequence takes 32 clock cycles to be traversed. The correct sequence should be 11110000111100001010101010101010. Requirement: if any bit is presented out of order, the procedure is aborted and the FSM unlocking will only be attempted again if the *op_fsm* is executed again. Requirement: no other operations should be executed before the FSM is properly unlocked.

Your grade is between 0 and 50. SF2 is worth 20 points, 10 coming from the feature itself and another 10 from the testbench. SF1, SF3, and SF4 are worth 10 points each.

Rules of the game

The project is individual, not a group assignment.

Do not use SystemVerilog features.

The control logic of the RoT must be implemented with a **single FSM**. You can assume that the RoT only has to support one functionality at a time (however, the CPU might incorrectly ask for multiple operations to be executed before they have finished).

All code written must respect assignment rules for sequential and combinational logic. In short, always @(posedge) blocks should make assignments on *reg* types using the <= operator. On the other hand, the always@(*) blocks should use the = operator. For a very easy to follow template of what is allowed, check https://ati.ttu.ee/~spagliar/teaching/ias0630/labs/lab3.v. You are also not allowed to have multiple assignments to the same variable coming from different always blocks. Failing to follow these rules/templates will prevent you from obtaining full grades, even if the security features appear to be functionally correct.

In general, you are expected to write behavioral Verilog code, except for the oscillators that have to be manually put together with structural Verilog.

You may use as many Verilog modules as you deem necessary. However, keep only one module per .v file. You are also encouraged to use the $display function to help with debugging. For instance, you should probably write to the terminal when the RoT starts/finishes an operation.

You are expected to write (most of) your code during lab time.

Provided files

Check https://ati.ttu.ee/~spagliar/teaching/ias0630/project/ for a list of files that you will be given. In particular, look at inv.sv for a description on an inverter that has a random delay between 9ps and 11ps. Similarly, a nand gate is provided that also has a random delay between 10ps and 12ps. These files will allow you to build the oscillators that are required for the TRNG/PUF.

A template for the RoT top-level module is also provided in rot.v.

Testbenches #1, #2, and #3 will be provided on the first day you have to present the project. They are not provided at this time as not to encourage you to write code that bypasses the testbench without implementing any security features.