CPEN 311 Lab 3 Submission

Daniel Li (51814762), Labib Kowsar (89102628)

June 15th, 2025

1. SOF file directory: **rtl\output_files\rc4.sof**

2. Status of lab (what works and what doesn't):

- TLDR: everything including the bonus works, this includes simulations and SignalTap for all FSMs written in this lab as required
    - Task 1 complete
        - S (working) memory instance initializes array according to for loop.
    - Task 2 complete
        - 2a: Working memory is modified according to for loop and matches the expected result from the lab pdf.
        - 2b: Algorithm is implemented and we were able to decrypt all sample messages using the secret_keys provided.
    - Task 3 complete
        - Brute force algorithm implemented to cycle through the key space to crack the message
    - Bonus: complete
        - Instantiated 4 cores, each core's current key is displayed on HEX displays (alternates between displaying each key of each core)
        - When a correct key is found, correct key is displayed on HEX displays, all cores terminate operation
        - Everything works, let's go!

---

3. Annotated simulation screenshots as required by the lab

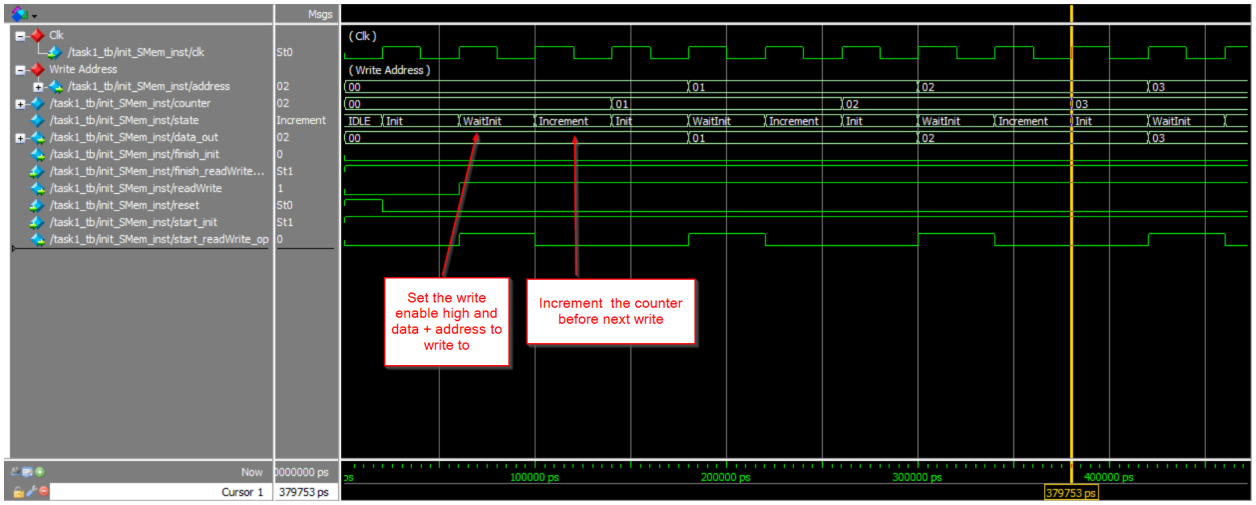
For this Lab, we utilized the Start-Finish protocol for controlling the datapath. All of our modules except for the top module (the module connecting the signals between the FSMs) utilize this control protocol to report when its task is complete. This helps increase the modularity while ensuring the FSMs are correctly coordinated.

Task 1

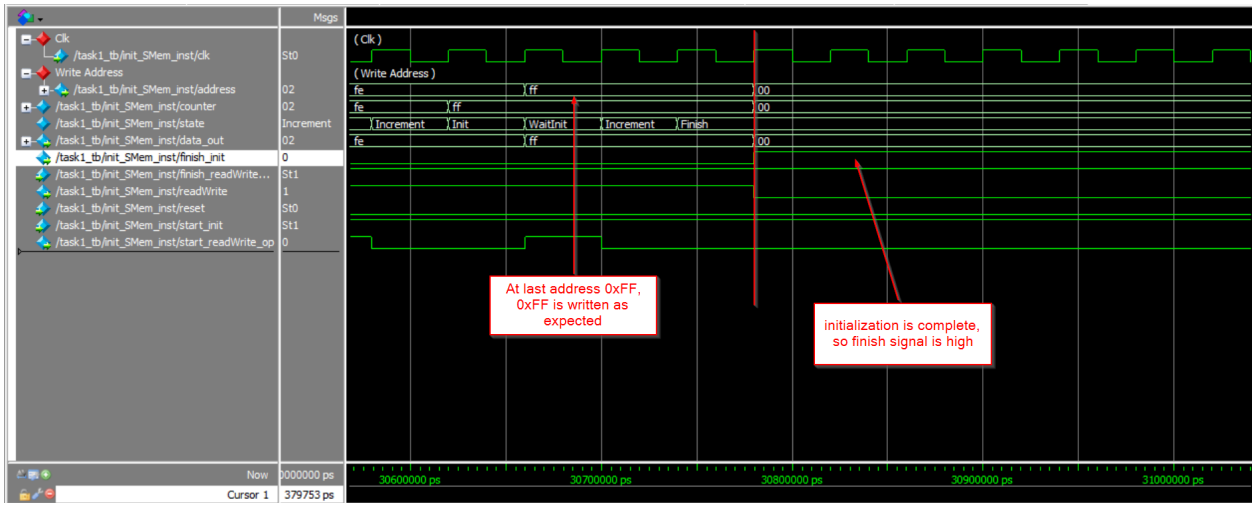
init_SMem

Module that initializes the working memory to its starting values.

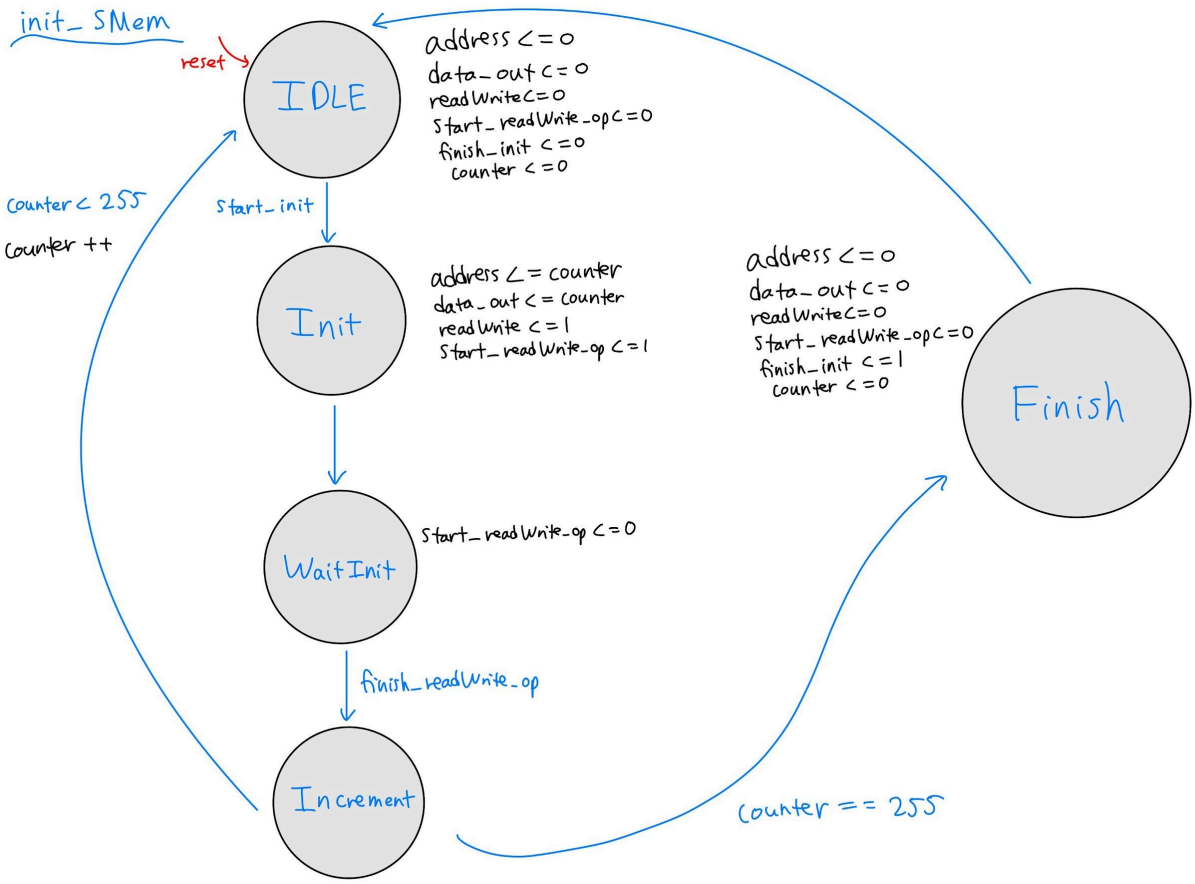
→ First few iterations: module writes to the address with its own value. (S[adr] = adr)



→ Init module finish writing and sets finish to high



FSM Flow Chart - init_SMem



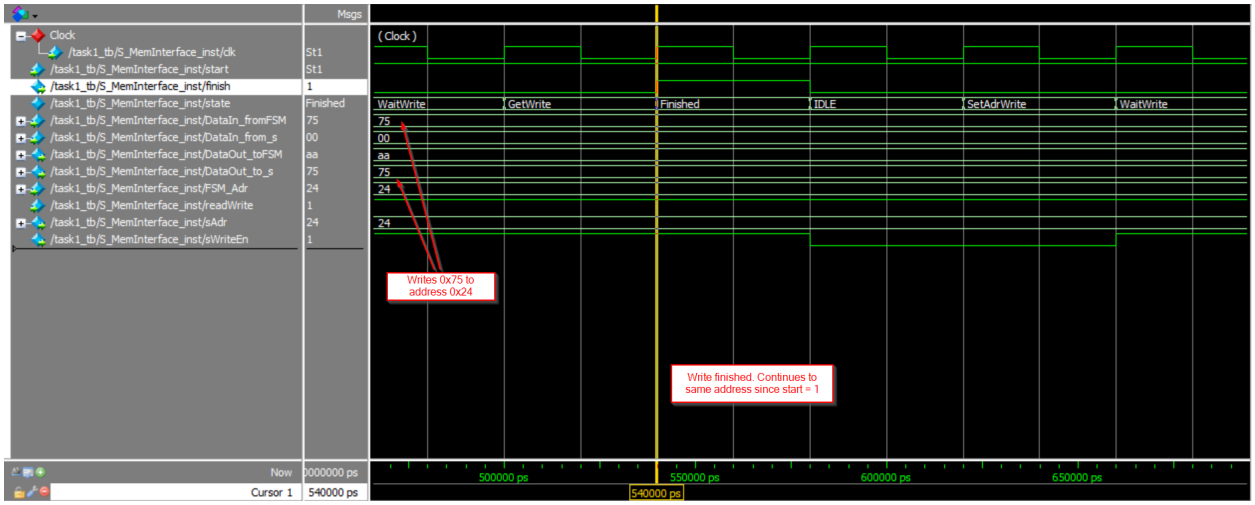
S_MemInterface

This module is a helper function that facilitates read/write operations to the working memory module. When it's parameters are passed, the module will cycle through its states and return a finish flag for when the read/write operation is complete.

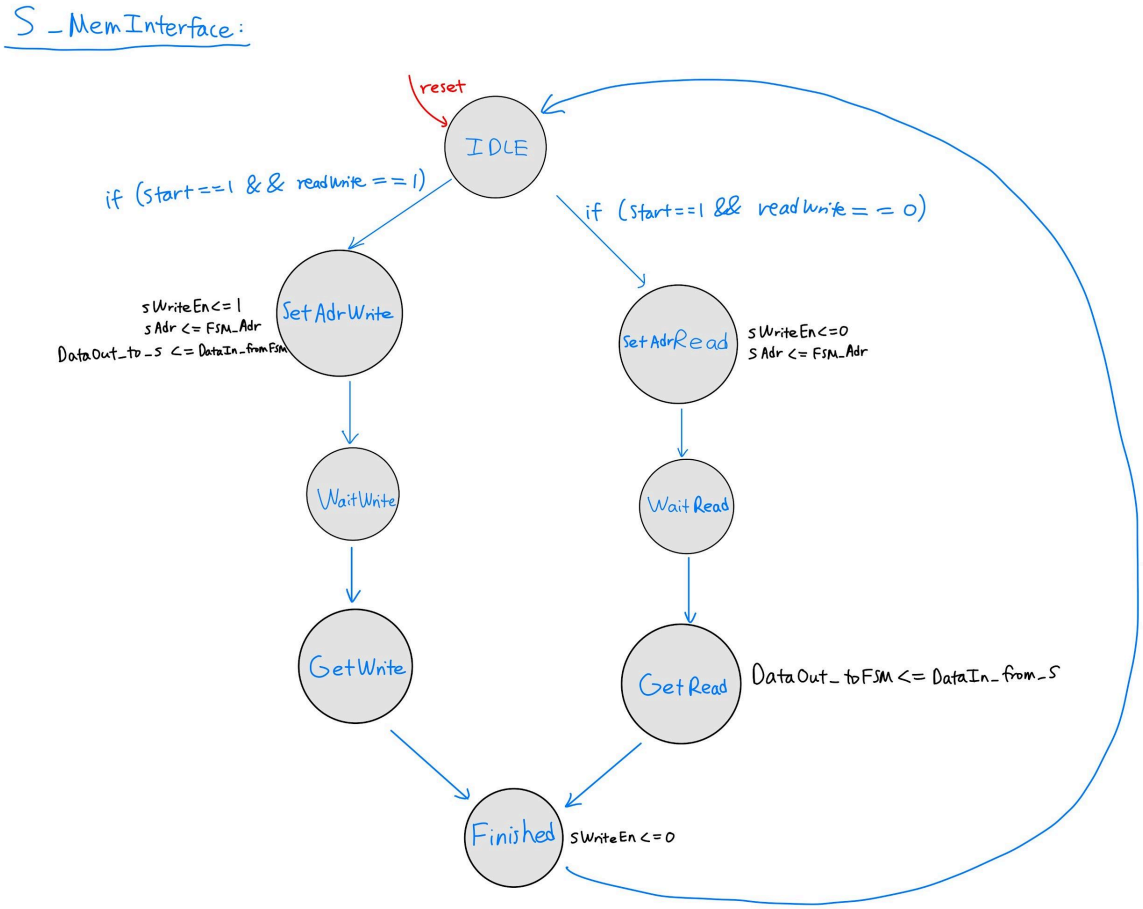
For instance, in init_SMem, the readWrite (1 = read, 0 = write) command is used to write data. This module is used repeatedly for all of our read/write operations in this lab.

→ Read cycle. Return value of 0xAA as expected from testbench.

→ Write Operation. Writes 0x75 to address 0x24



FSM Flow Chart - S_MemInterface



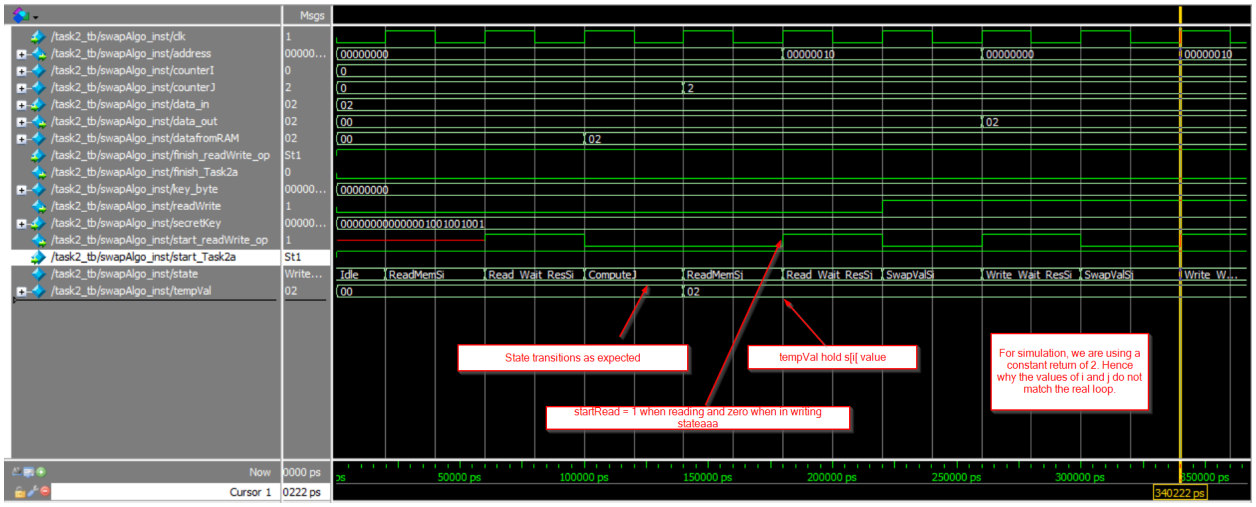
Task 2

swapAlgo

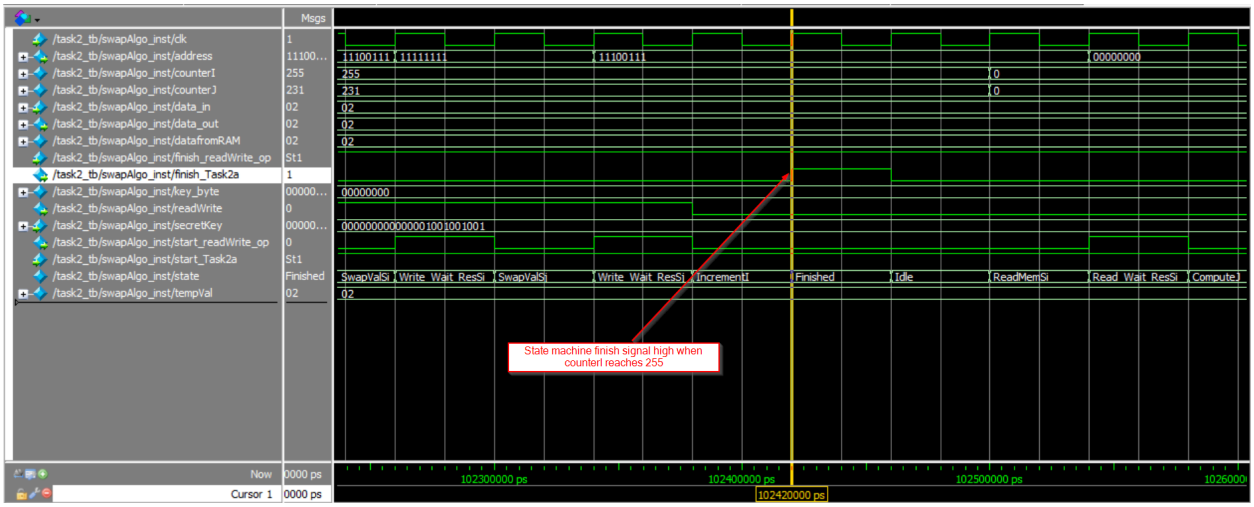
Module for task2a algorithm. Named SwapAlgo since after computing the value j, memory positions of s[i] and s[j[are swapped.

```
// shuffle the array based on the secret key.  You will build this in Task 2
j = 0
for i = 0 to 255 {
        j = (j + s[i] + secret_key[i mod keylength] )  //keylength is 3 in our impl.
        swap values of s[i] and s[j]
}
```

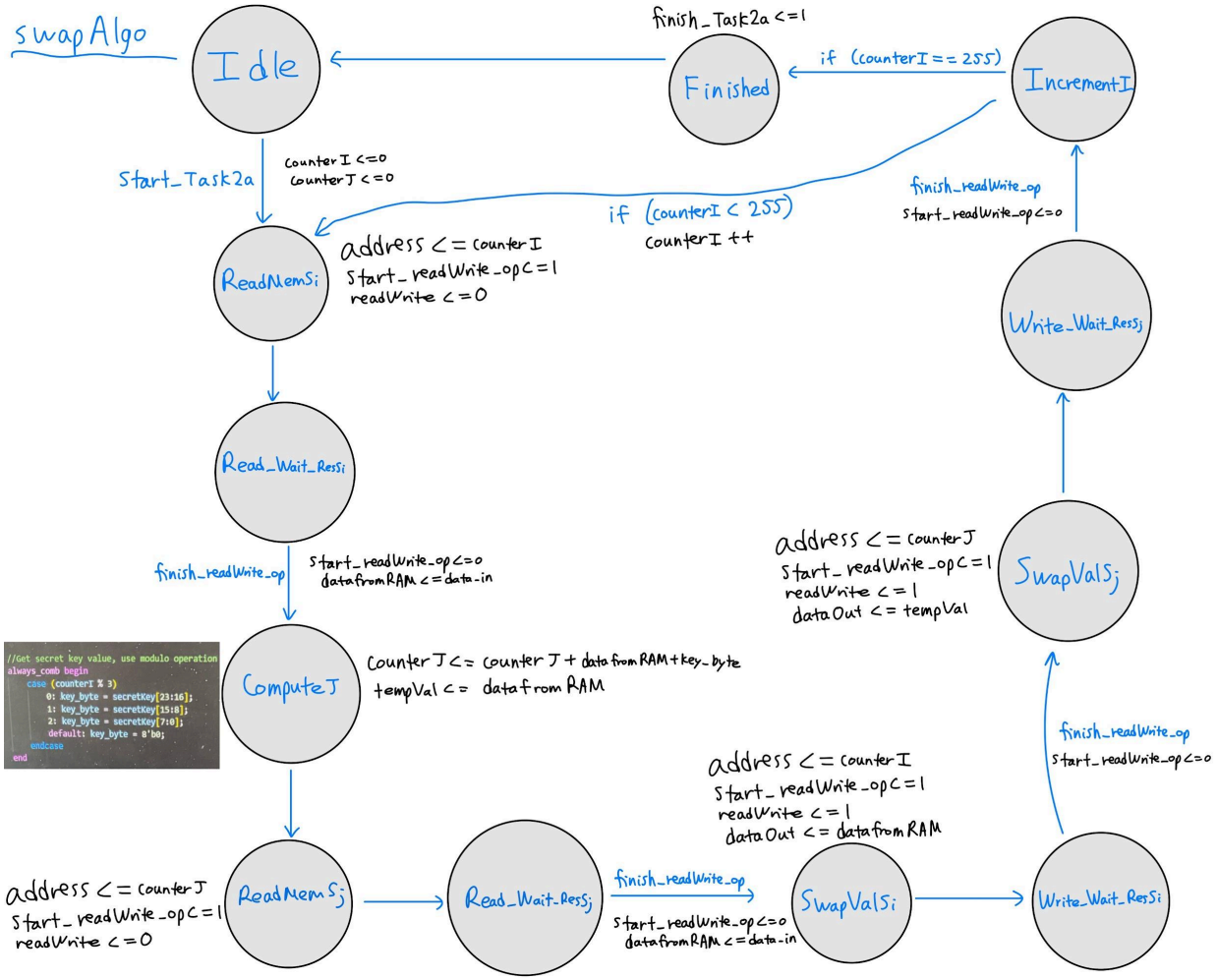
→ SwapAlgo starts. CounterI increments from 0 to 255, and CounterJ is computed.



→ SwapAlgo finished



FSM Flow Chart - SwapAlgo

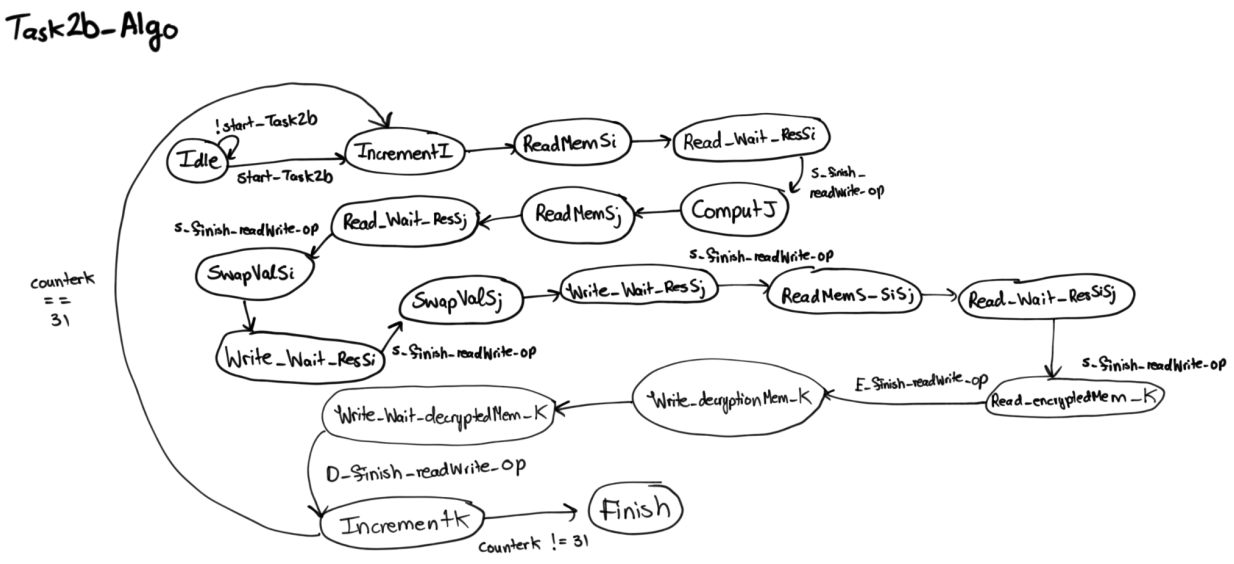


Task2b_Algo

Module that implements for loop in task 2b.

```
// compute one byte per character in the encrypted message.  You will build this in Task 2
i = 0, j=0
for k = 0 to message_length-1 {   // message_length is 32 in our implementation
        i = i+1
        j = j+s[i]
        swap values of s[i] and s[j]
        f = s[ (s[i]+s[j]) ]
        decrypted_output[k] = f xor encrypted_input[k]   // 8 bit wide XOR function
}
```

FSM Flow chart



Idle	counterI <= 0;
	counterJ <= 0;
	counterK <= 0;
	datafromRAM <= 0;
	f <= 0;
	S_i <= 0;
	S_j <= 0;
	
	S_address <= 0;
	S_data_out <= 0;

	E_address <= 0; D_address <= 0; D_data_out <= 0; S_start_readWrite_op <= 0; E_start_readWrite_op <= 0; D_start_readWrite_op <= 0; state <= IncrementI;
IncrementI	counterI <= counterI + 8'b1; state <= ReadMemSi;
ReadMemSi	S_start_readWrite_op <= 1; S_address <= counterI; state <= Read_Wait_ResSi;

Read_Wait_ResSi	if (S_finish_readWrite_op) begin
	S_start_readWrite_op <= 0;
	S_i <= S_data_in;
	
	state <= ComputeJ;
	end

ComputeJ	counterJ <= counterJ + S_i; //j = j + S[i]
	
	state <= ReadMemSj;

ReadMemSj	S_start_readWrite_op <= 1; S_address <= counterJ; state <= Read_Wait_ResSj;

Read_Wait_ResSj	if (S_finish_readWrite_op) begin
	S_start_readWrite_op <= 0;
	S_j <= S_data_in;
	
	state <= SwapValSi;
	end

SwapValSi	S_start_readWrite_op <= 1; S_address <= counterI; S_data_out <= S_j; state <= Write_Wait_ResSi;

Write_Wait_ResSi	if(S_finish_readWrite_op) begin
	S_start_readWrite_op <= 0;
	
	state <= SwapValSj;
	end

SwapValSj	S_start_readWrite_op <= 1; S_address <= counterJ; S_data_out <= S_i; state <= Write_Wait_ResSj;

Write_Wait_ResSj	if(S_finish_readWrite_op) begin
	S_start_readWrite_op <= 0;
	
	state <= ReadMemS_SiSj;
	end

ReadMemS_SiSj	S_start_readWrite_op <= 1; S_address <= S_i + S_j; state <= Read_Wait_ResSiSj;

Read_Wait_ResSiSj	if (S_finish_readWrite_op) begin
	S_start_readWrite_op <= 0;
	f <= S_data_in;
	
	state <= Read_encryptedMem_K;
	end

Read_encryptMem_K	E_start_readWrite_op <= 1; E_address <= counterK; state <= Read_Wait_encryptedMem_K;

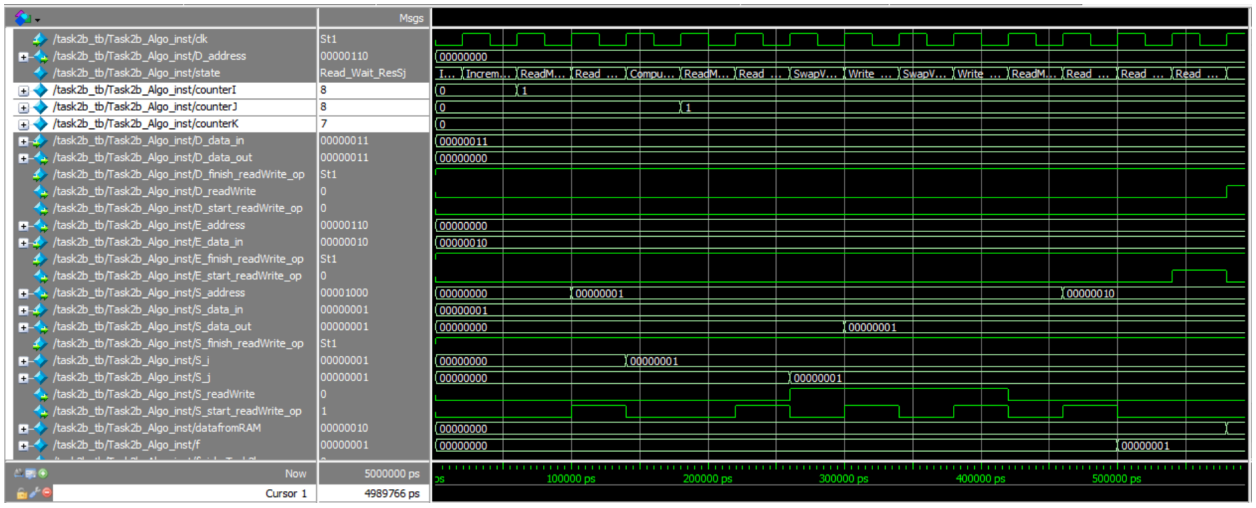
Write_decryptedMem_K	if (E_finish_readWrite_op) begin
	E_start_readWrite_op <= 0;
	datafromRAM <= E_data_in;
	
	state <= Write_decryptedMem_K;
	end

Wait_decryptedMem_K	D_start_readWrite_op <= 1; D_address <= counterK; D_data_out <= datafromRAM ^ f; state <= Write_Wait_decryptedMem_K;

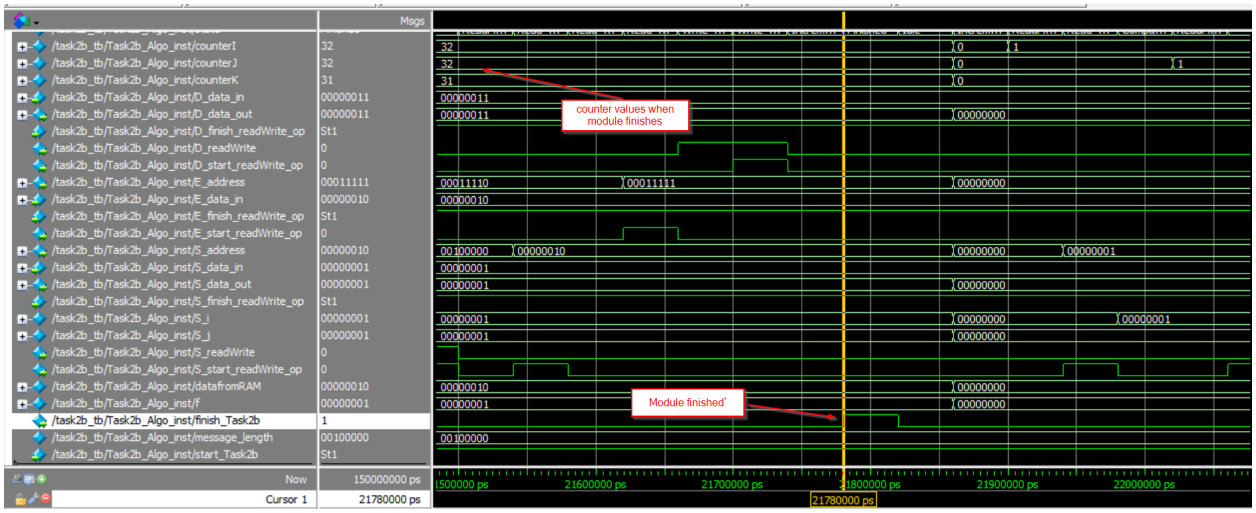
IncrementK	if (counterK == message_length-1) begin //for k = 0 to message_length-1
	state <= Finished;
	end else begin
	counterK <= counterK + 8'b1;
	state <= IncrementI;
	end

Finish	state <= Idle;

→ Picture below shows how the signals interact to implement the algorithm. It uses multiple interfacing modules such as Decrypted_Meminterface and Encrypted_Meminterface below.



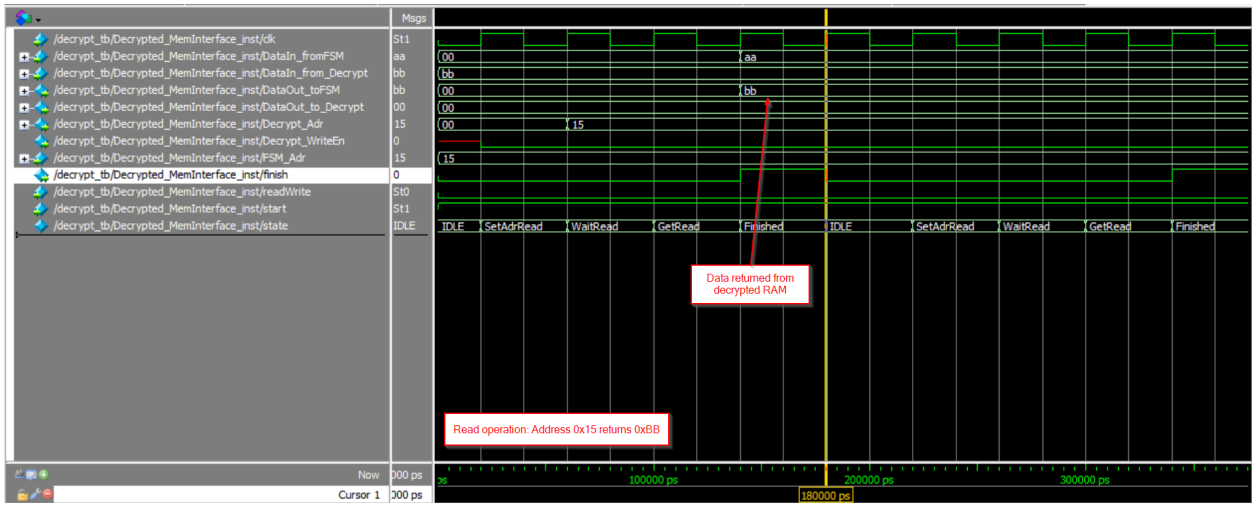
→ Module terminates



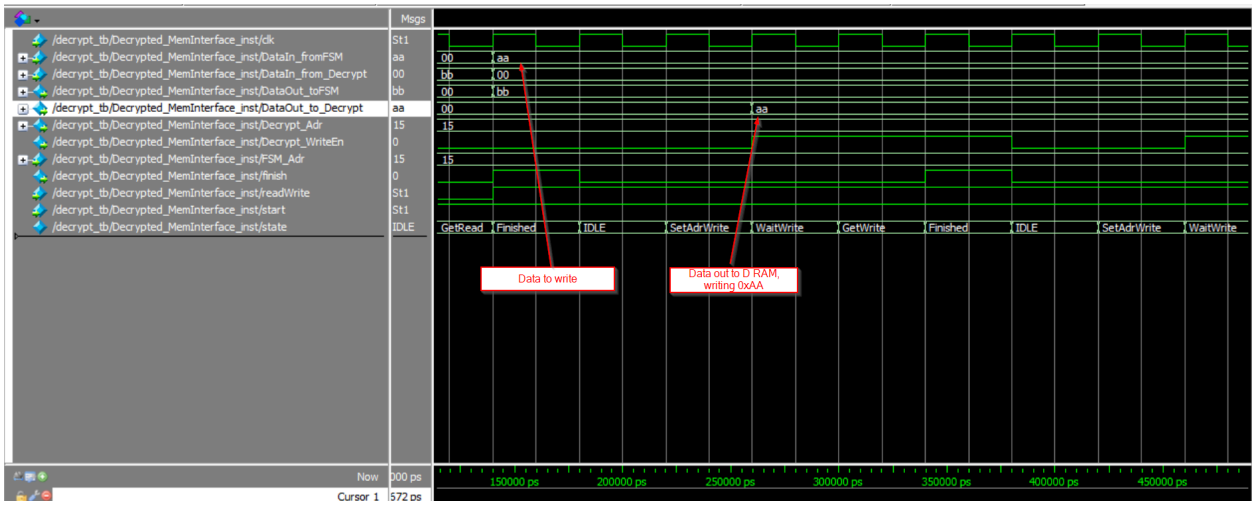
Decrypted_MemInterface

Module similar to S_MemInterface, and is used to interface with the 32 word x 8 bit Decrypted RAM memory. Tells the calling FSM when writing to the memory is finished.

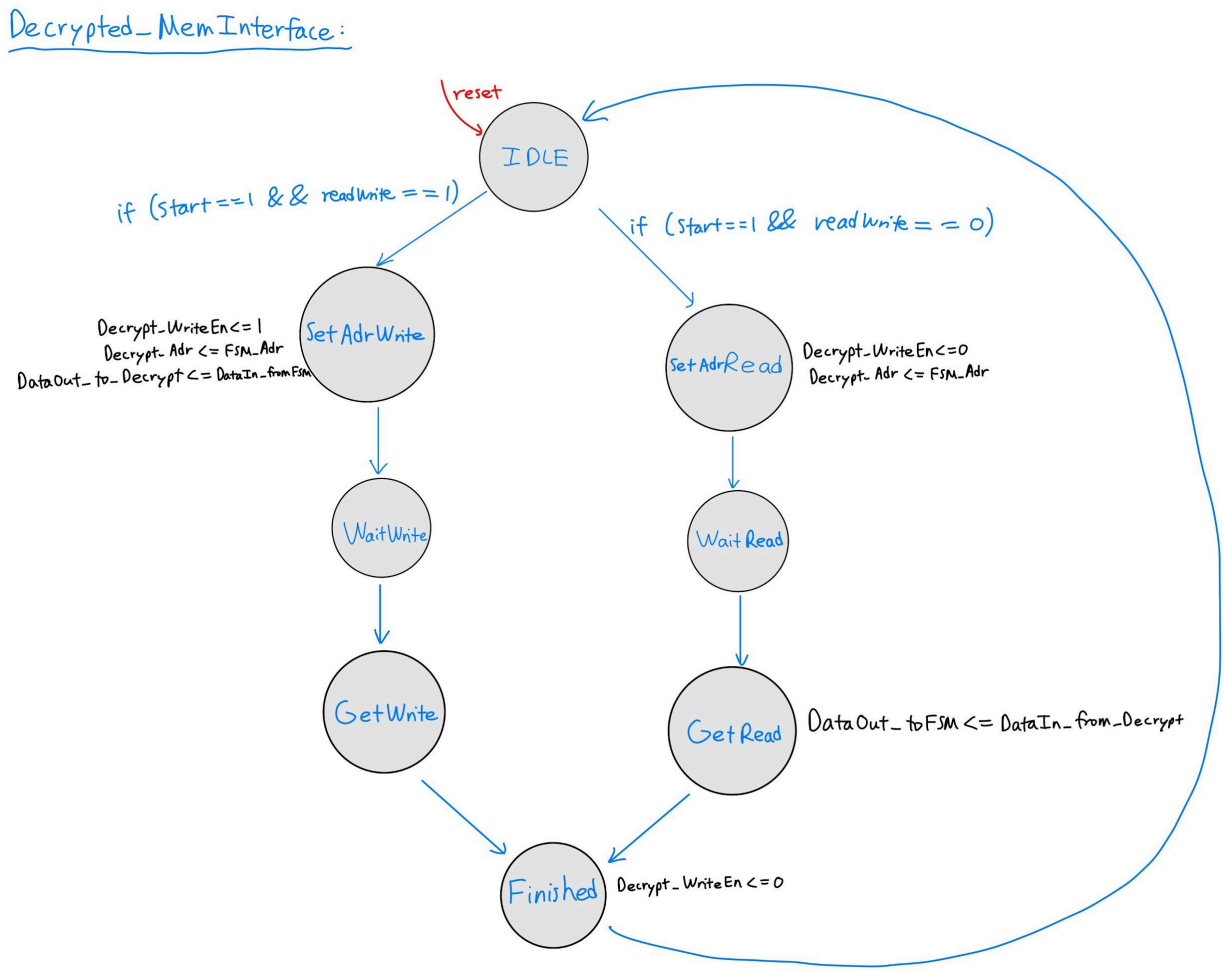
→ Read operation. This is not used in the algorithm but we kept it in (if we want to expand on its functionality in the future).



→ Write operation

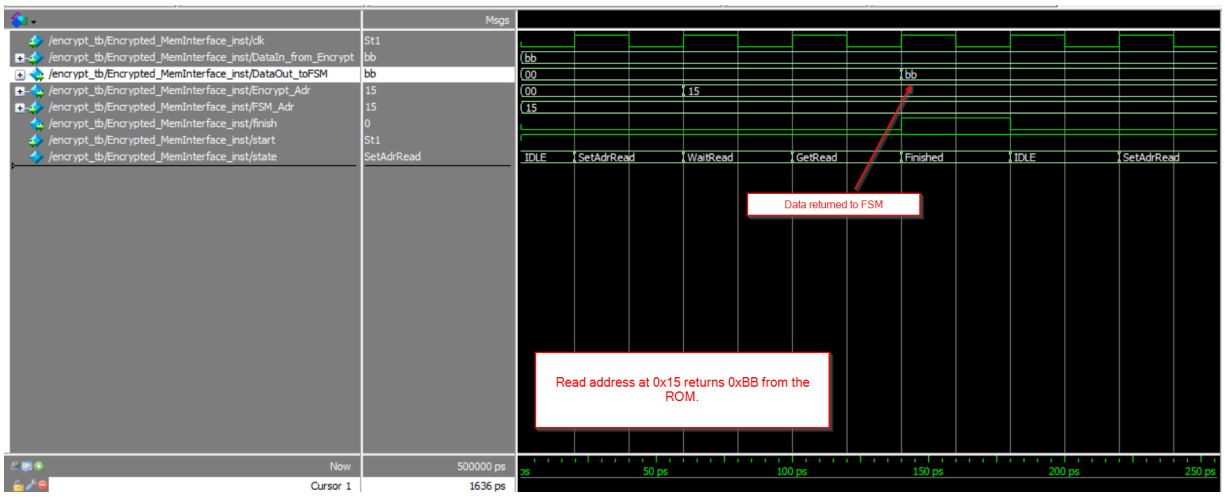


FSM Flow Chart - Decrypted_MemInterface

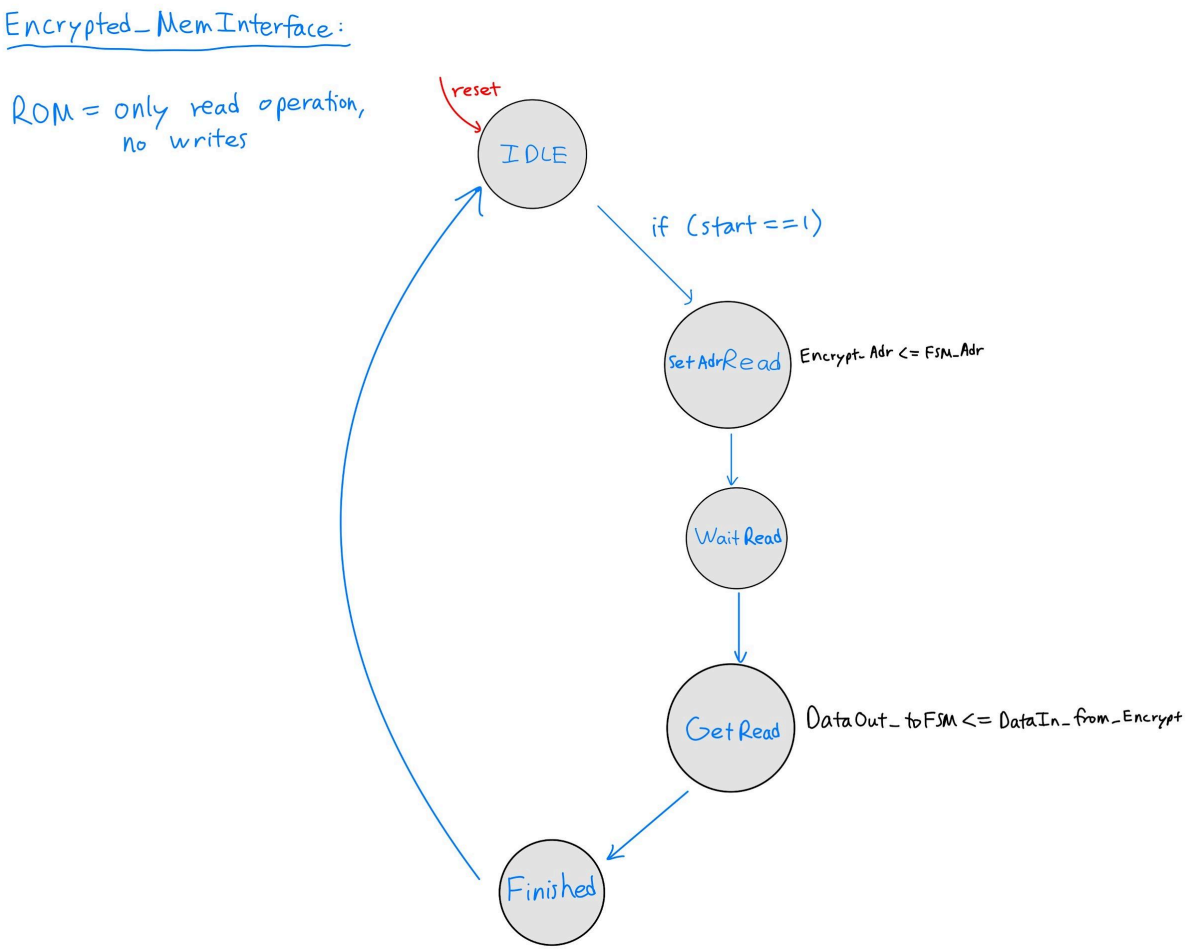


Encrypted_MemInterface

Module similar to S_MemInterface, and is used to interface with the 32 word x 8 bit Encrypted ROM memory. Tells the calling FSM when reading from the ROM is finished.



FSM Flow Chart - Encrypted_MemInterface

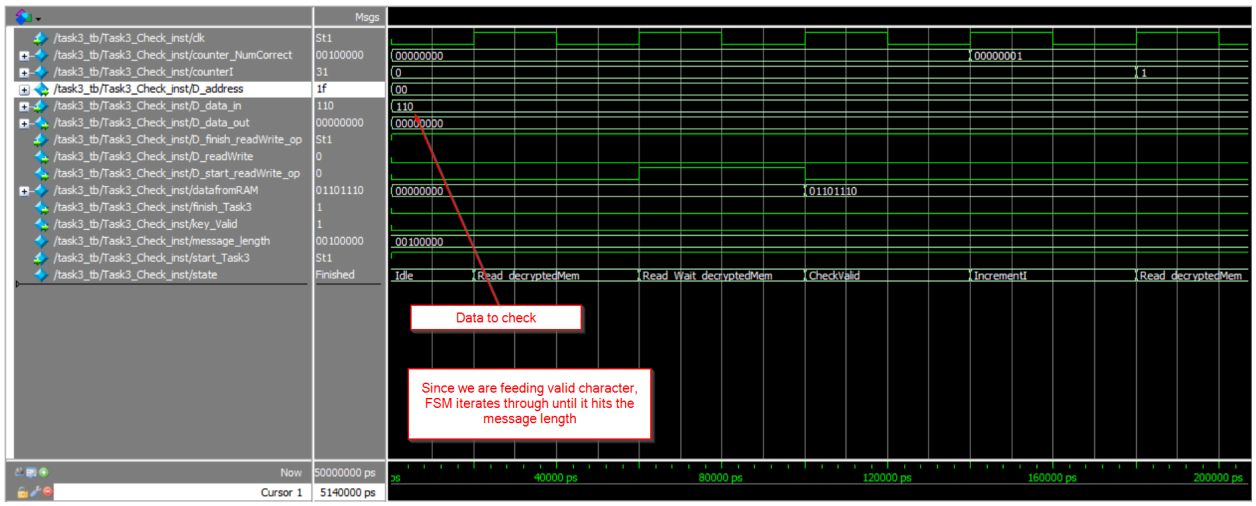


Task 3

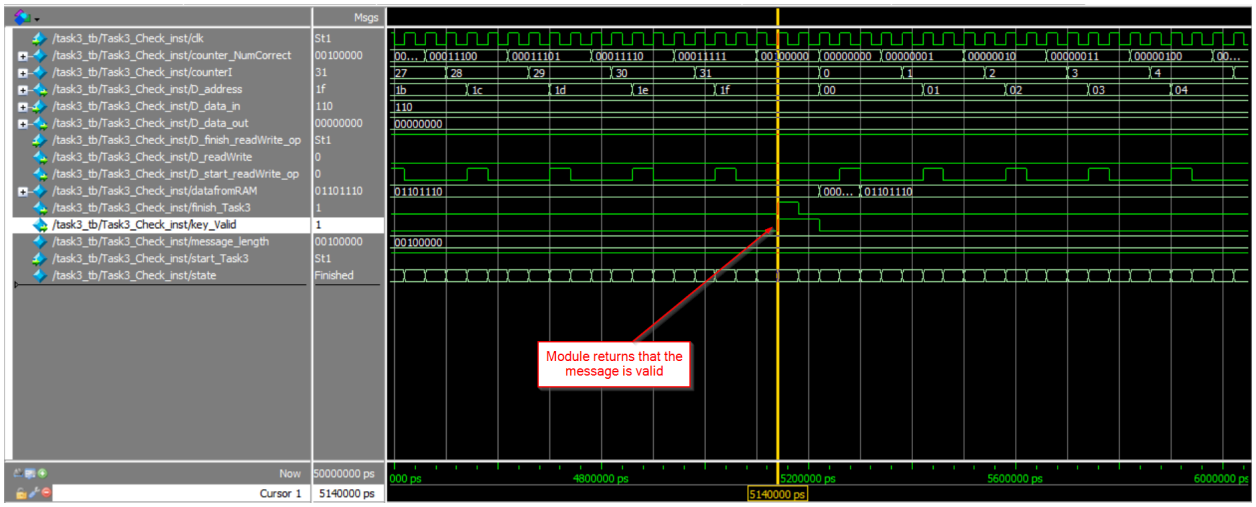
Task3_Check

This module checks whether the result in D instance RAM is valid (each character in the message is from lower a-z or a space). Returns keyValid if the message is valid.

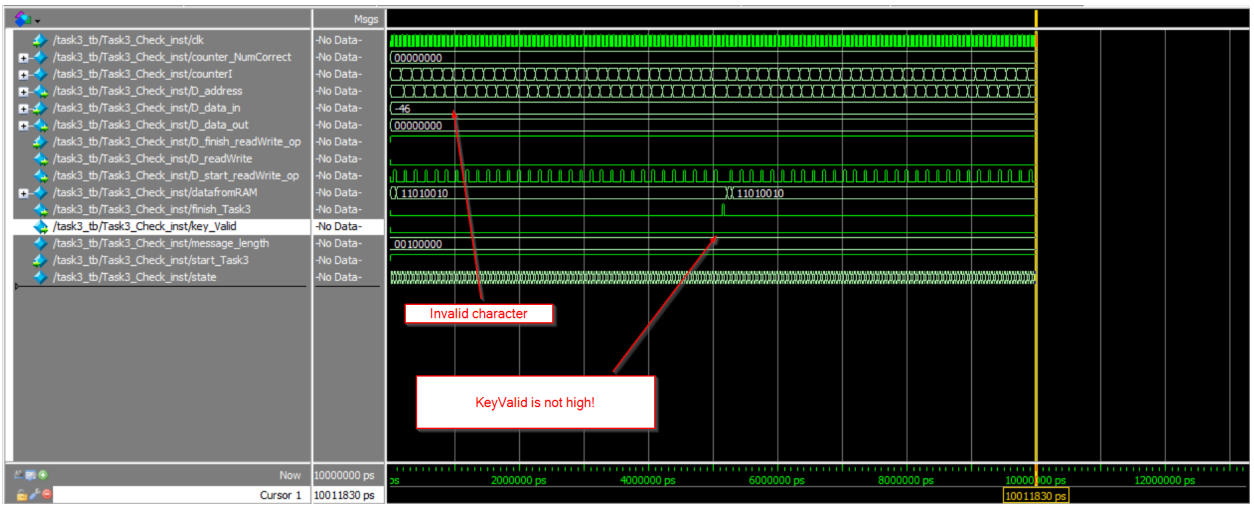
→ FSM cycling through given a constant valid character.



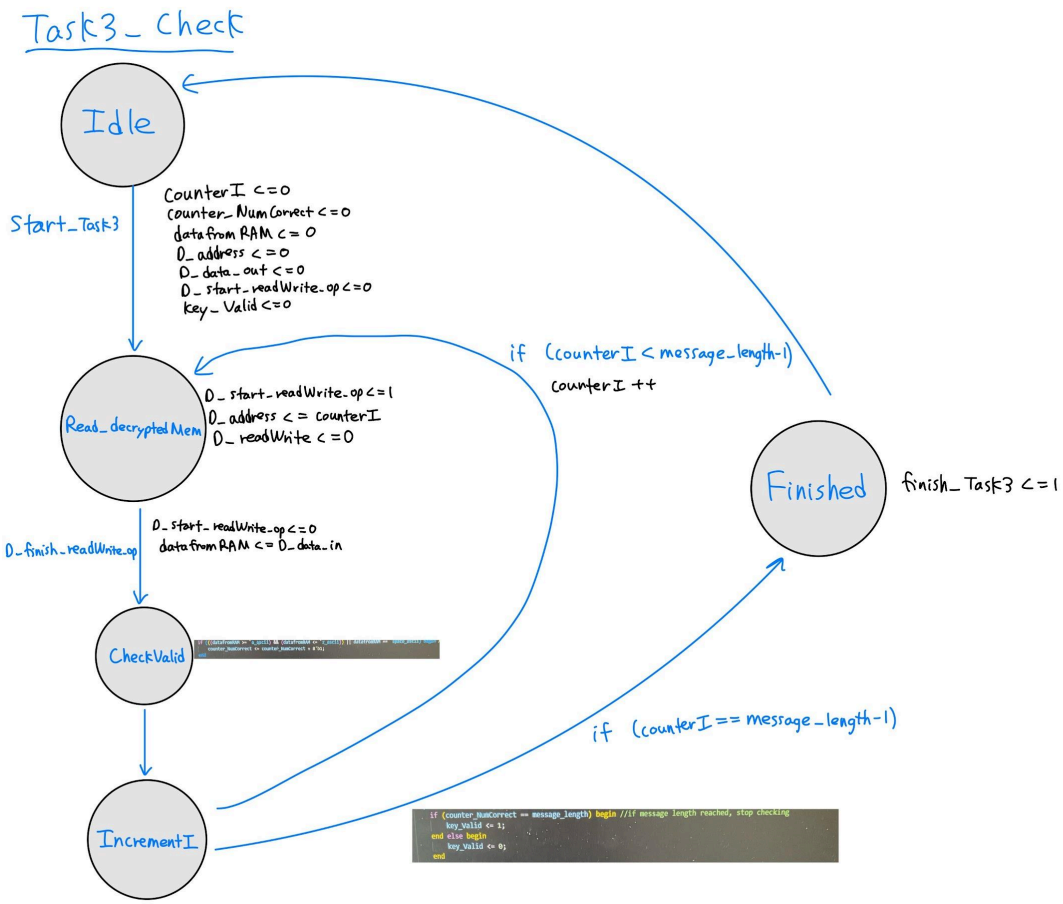
→ KeyValid reached and returned



→ Not valid character, keyValid is never high



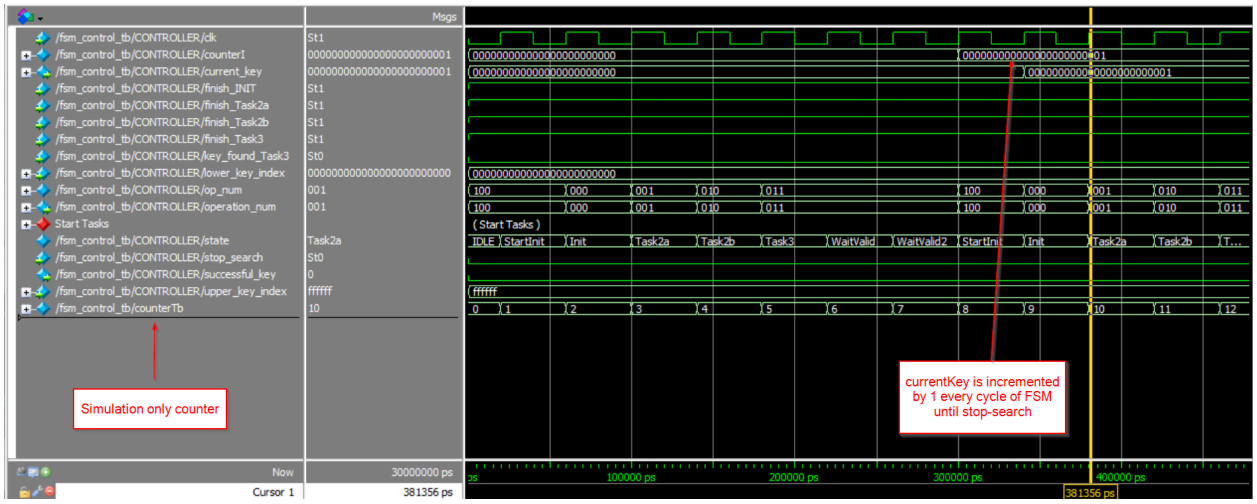
FSM Flow Chart - Task3_Check

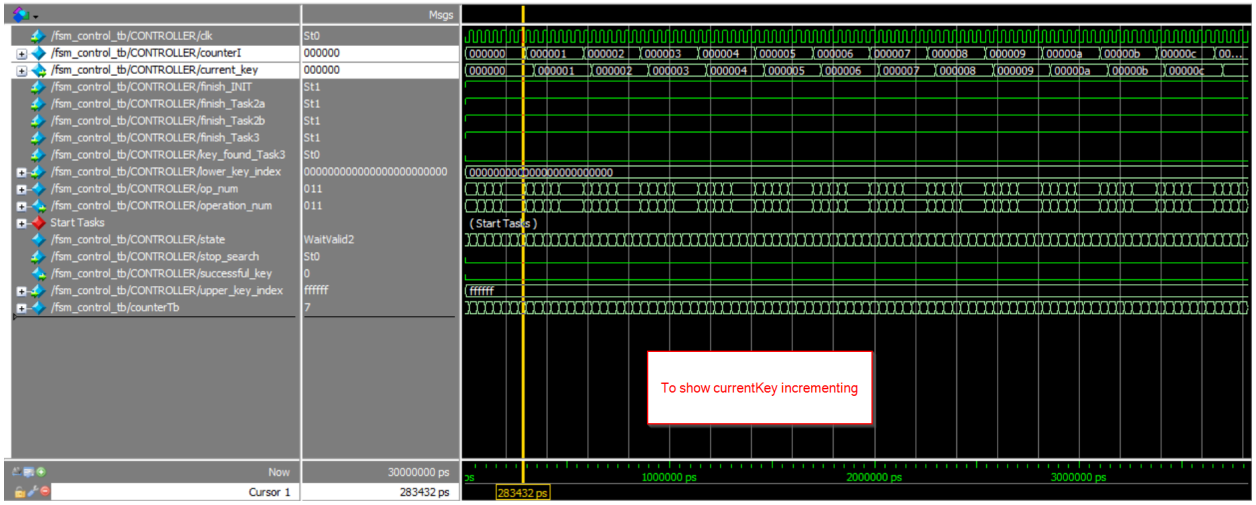


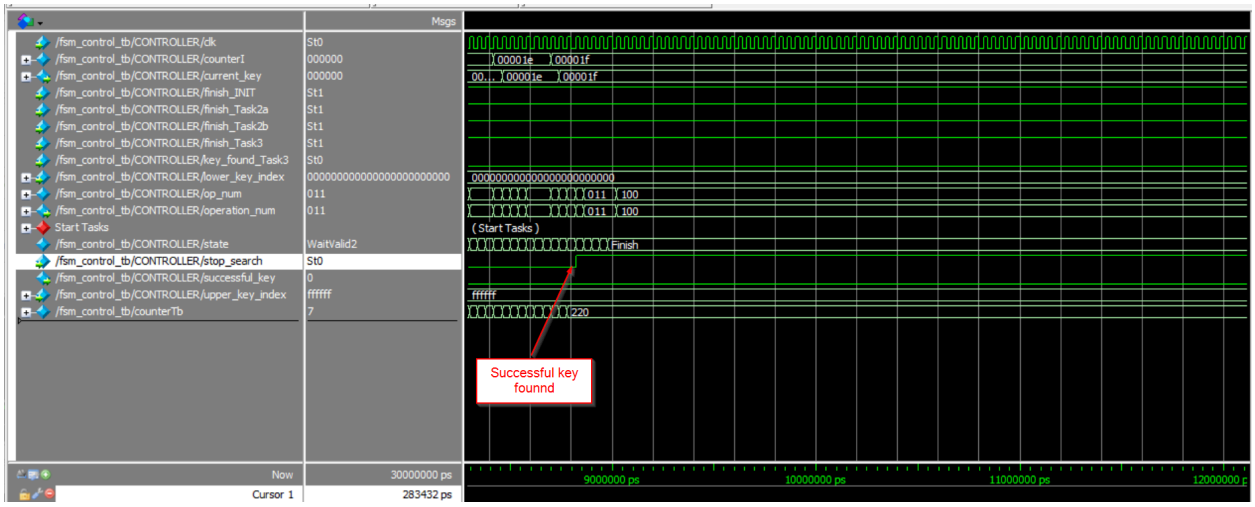
FSM_Controller

This module controls the datapath of calling modules to decrypt the message. It starts with the lower bound possible secretKey, checks if a valid message has been formed. If not, then it increments the key until the upper bound is reached.

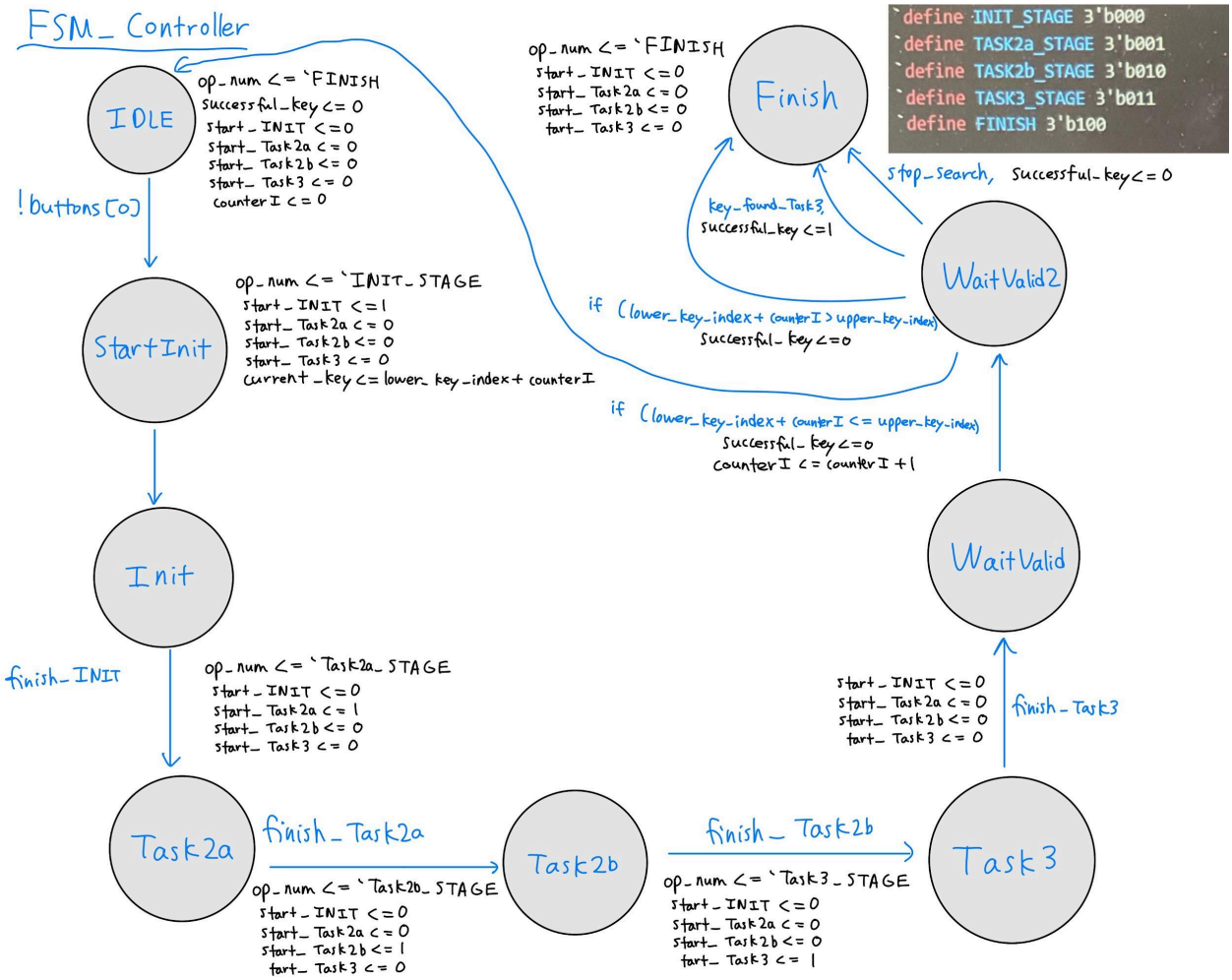
→ Using a simulation only counter, we increment it until 220, upon which, we set stop_search to high. This is to simulate the normal process of finding a successful key match.





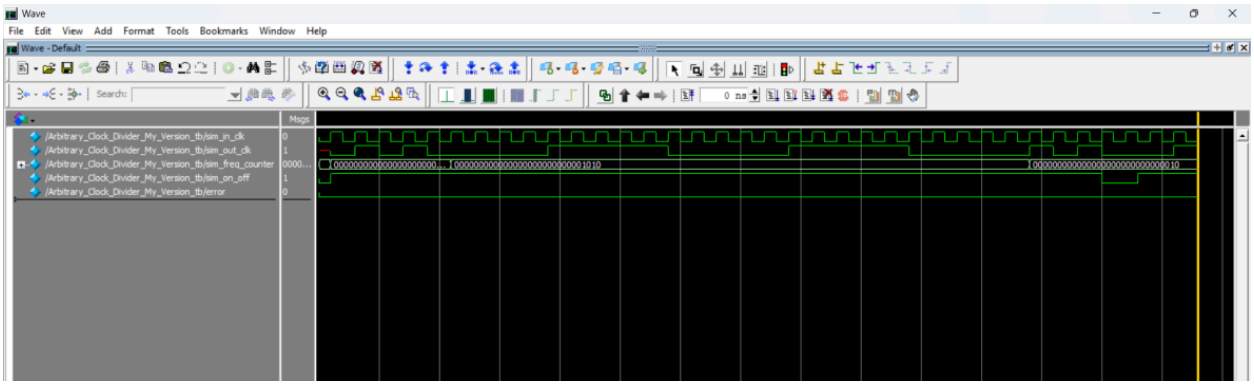


FSM Flow Chart - FSM_Controller



Clock_divider (used in bonus, not an FSM)

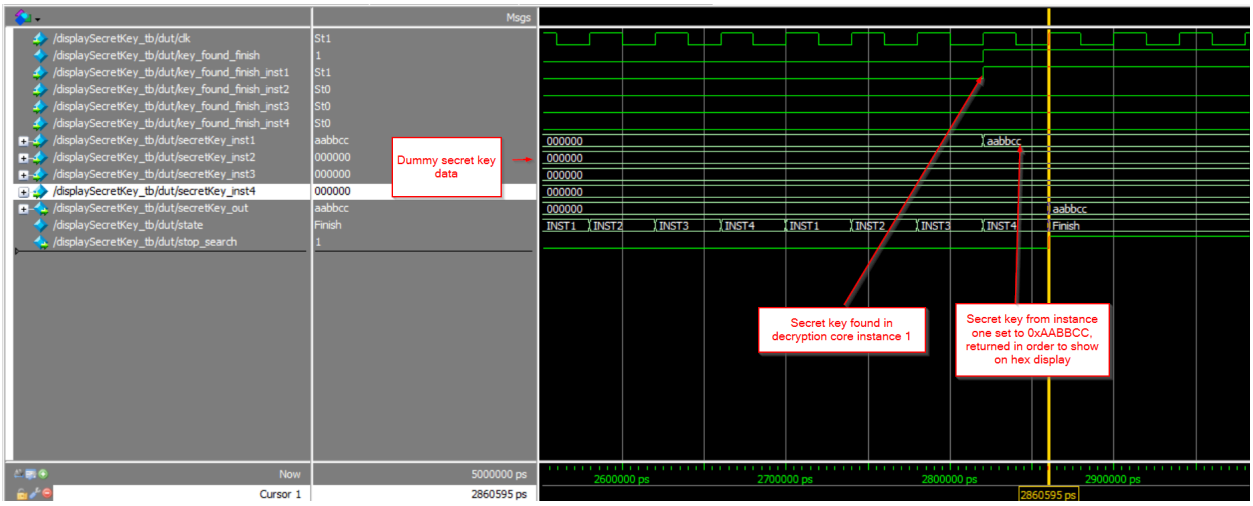
Module from Lab 1. Used to divide the input clock into a slower output clock.



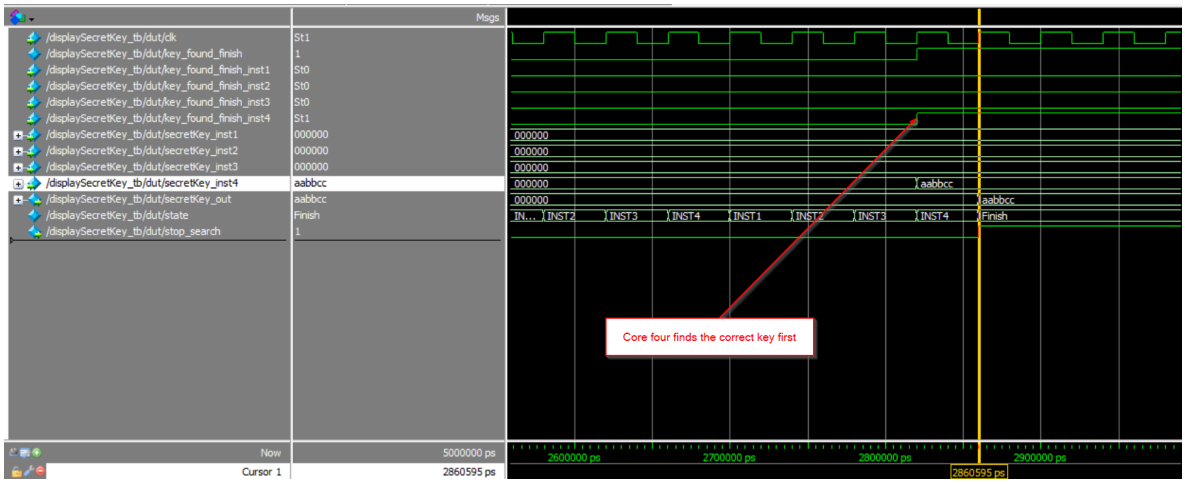
Display_SecretKey

This module displays the correct secret key once its been cracked by one of the four cores (HEX display).

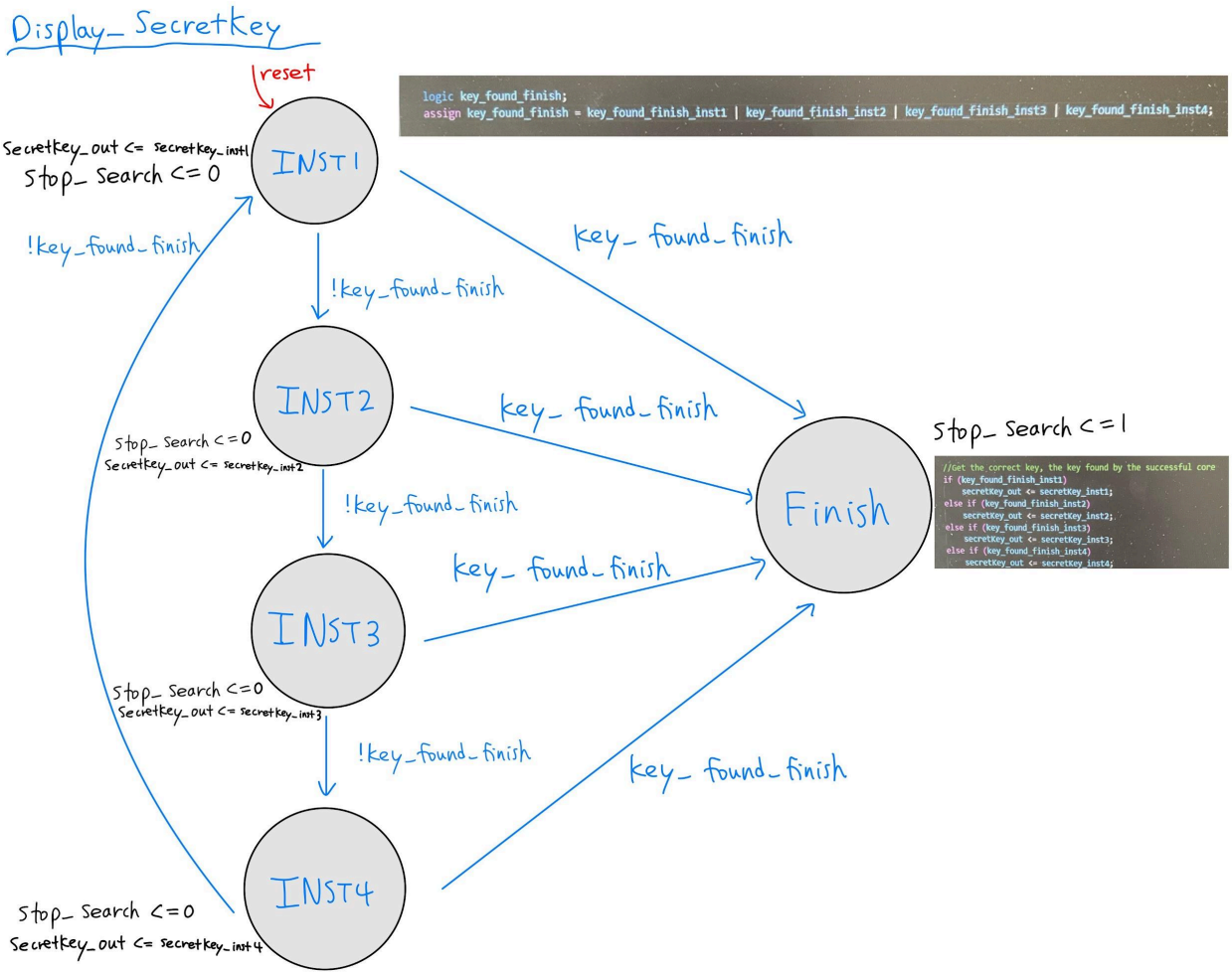
→ Example of core 1 finding the key first



→ Example of core four finding the key first



FSM Flow Chart - Display_SecretKey

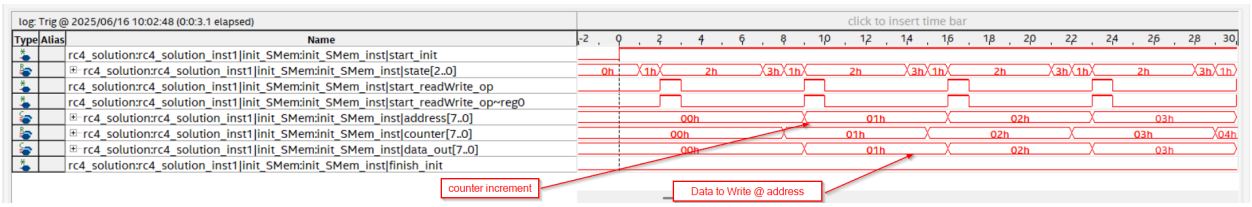


4. Annotated SignalTap screenshots as required by the lab

init_SMem

```
/*
| **State Name** | **Hex (2-digit)** |
| -------------- | ----------------- |
| `IDLE`         | 0x00              |
| `Init`         | 0x01              |
| `WaitInit`     | 0x02              |
| `Increment`    | 0x03              |
| `Finish`       | 0x04              |
*/
```

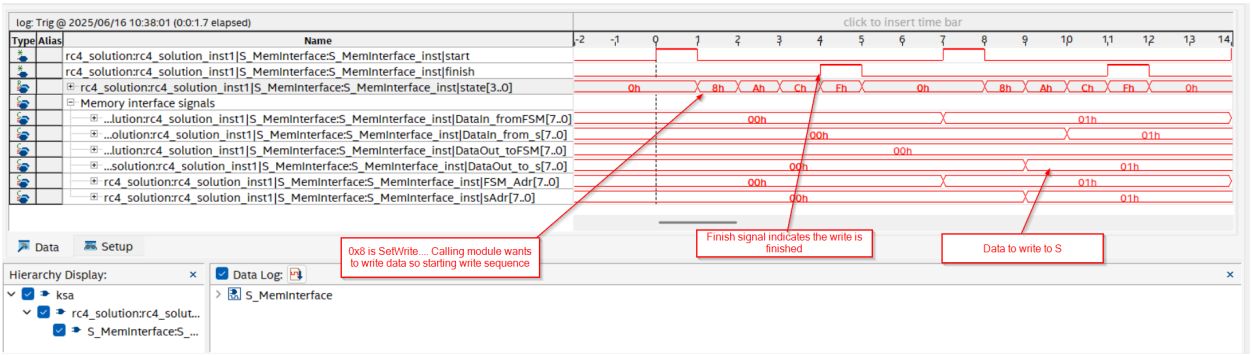
→ Write sequence



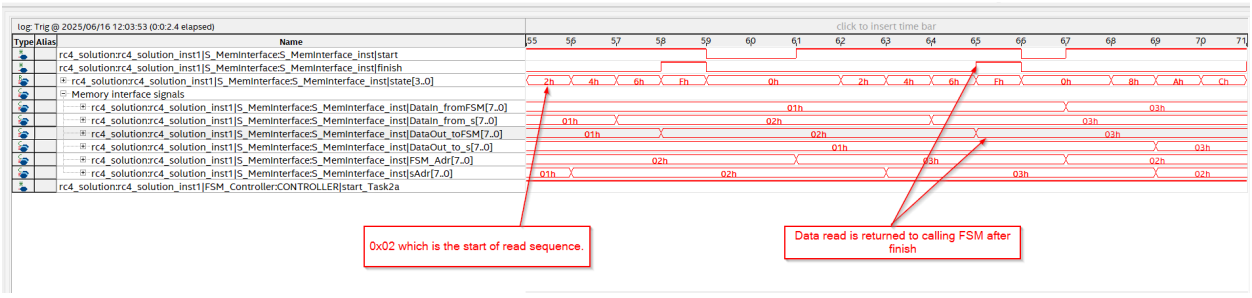
S_MemInterface (when init_SMem is the caller)

```
/*
| **State Name** | **Hex (2-digit)** |
| -------------- | ----------------- |
| `IDLE`         | 0x00              |
| `SetAdrRead`   | 0x02              |
| `WaitRead`     | 0x04              |
| `GetRead`      | 0x06              |
| `SetAdrWrite`  | 0x08              |
| `WaitWrite`    | 0x0A              |
| `GetWrite`     | 0x0C              |
| `Finished`     | 0x0F              |
*/
```

→ Write sequence

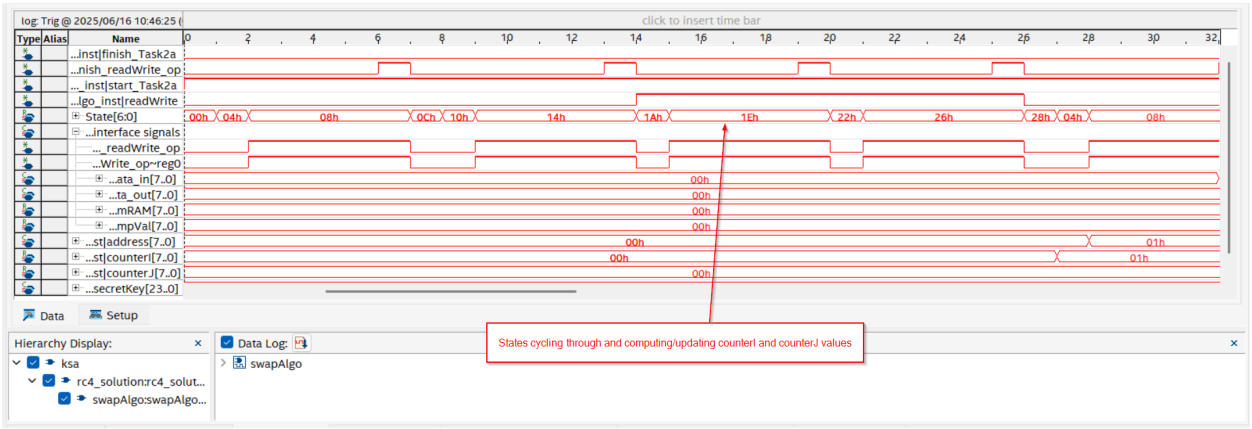


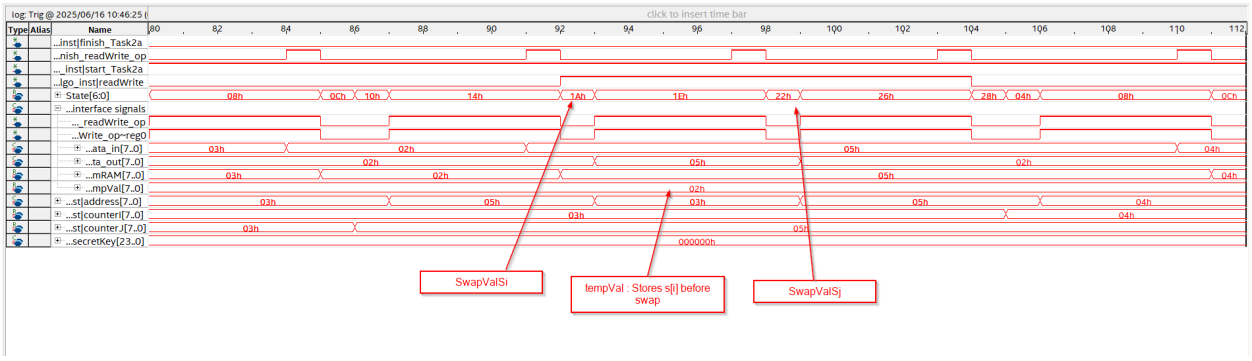
→ Read sequence (in swapAlgo)



swapAlgo

```
/*
| State Name | Hex Value |
|------------|-----------|
| Idle | 0x00 |
| ReadMemSi | 0x04 |
| Read_Wait_ResSi | 0x08 |
| ComputeJ | 0x0C |
| ReadMemSj | 0x10 |
| Read_Wait_ResSj | 0x14 |
| SwapValSi | 0x1A |
| Write_Wait_ResSi | 0x1E |
| SwapValSj | 0x22 |
| Write_Wait_ResSj | 0x26 |
| IncrementI | 0x28 |
| Finished | 0x2D |
*/
```

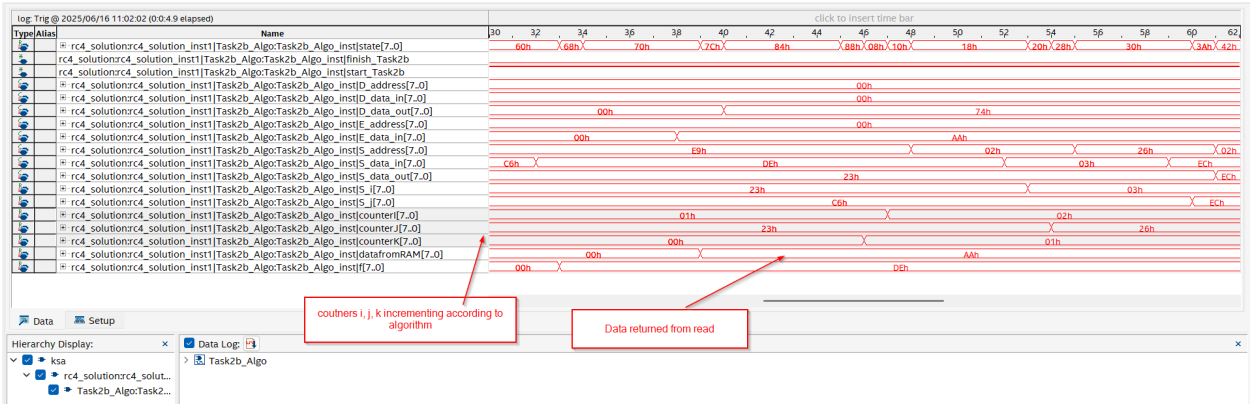




→ Longer timeframe snapshot

Task2b_Algo

```
/*
| **State Name**              | **Hex (2-digit)** |
| --------------------------- | ----------------- |
| `Idle`                      | 0x00              |
| `IncrementI`                | 0x08              |
| `ReadMemSi`                 | 0x10              |
| `Read_Wait_ResSi`           | 0x18              |
| `ComputeJ`                  | 0x20              |
| `ReadMemSj`                 | 0x28              |
| `Read_Wait_ResSj`           | 0x30              |
| `SwapValSi`                 | 0x3A              |
| `Write_Wait_ResSi`          | 0x42              |
| `SwapValSj`                 | 0x4A              |
| `Write_Wait_ResSj`          | 0x52              |
| `ReadMemS_SiSj`             | 0x58              |
| `Read_Wait_ResSiSj`         | 0x60              |
| `Read_encryptedMem_K`       | 0x68              |
| `Read_Wait_encryptedMem_K`  | 0x70              |
| `Write_decryptedMem_K`      | 0x7C              |
| `Write_Wait_decryptedMem_K` | 0x84              |
| `IncrementK`                | 0x88              |
| `Finished`                  | 0x91              |
*/
```

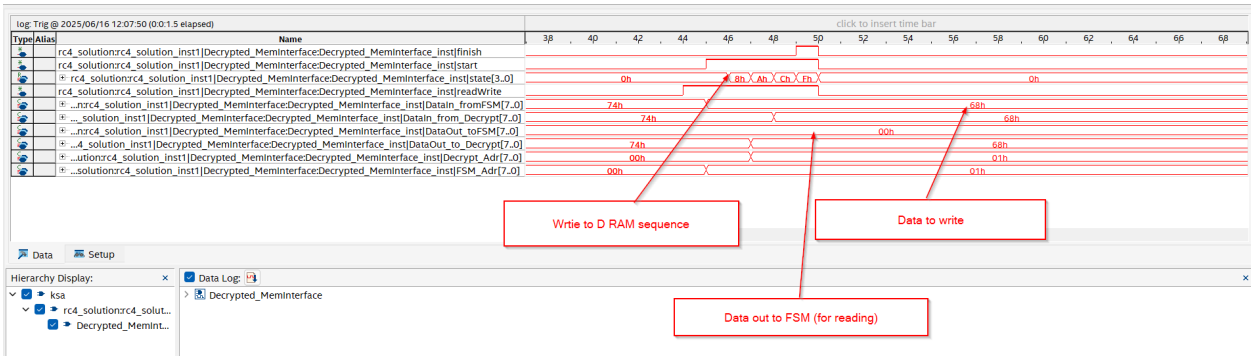


→ Longer timeframe snapshot

Decrypted_MemInterface

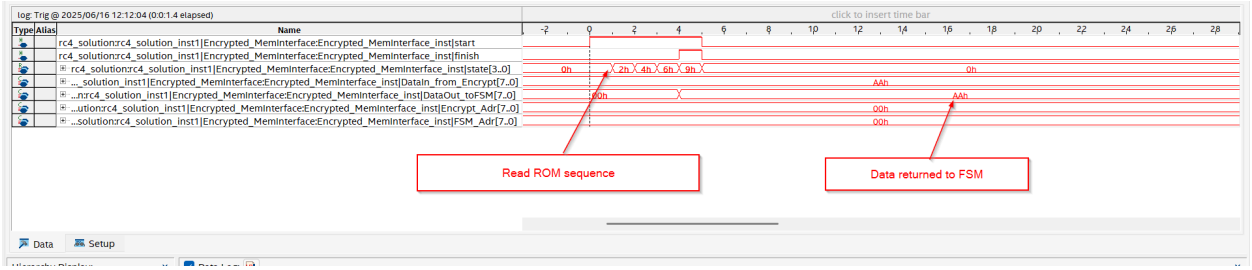
```
/*
| **State Name** | **Hex (2-digit)** |
| -------------- | ----------------- |
| `IDLE`         | 0x00              |
| `SetAdrRead`   | 0x02              |
| `WaitRead`     | 0x04              |
| `GetRead`      | 0x06              |
| `SetAdrWrite`  | 0x08              |
| `WaitWrite`    | 0x0A              |
| `GetWrite`     | 0x0C              |
| `Finished`     | 0x0F              |
*/
```

→ Calling from Task2b_mod



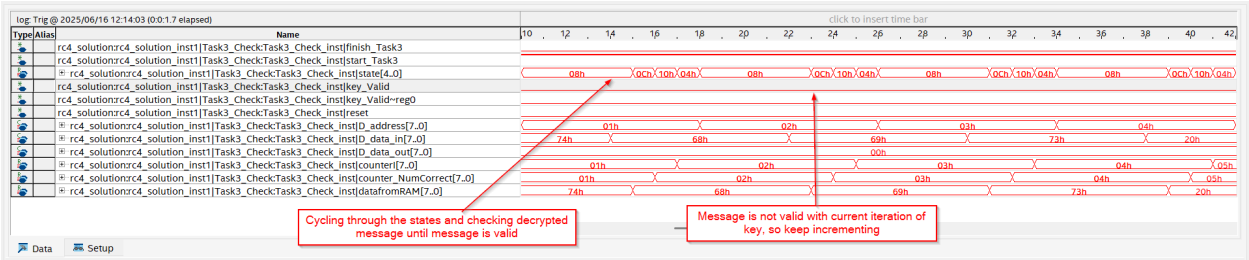
Encrypted_MemInterface

```
/*
| **State Name** | **Hex (2-digit)** |
| -------------- | ----------------- |
| `IDLE`         | 0x00              |
| `SetAdrRead`   | 0x02              |
| `WaitRead`     | 0x04              |
| `GetRead`      | 0x06              |
| `Finished`     | 0x09              |
*/
```



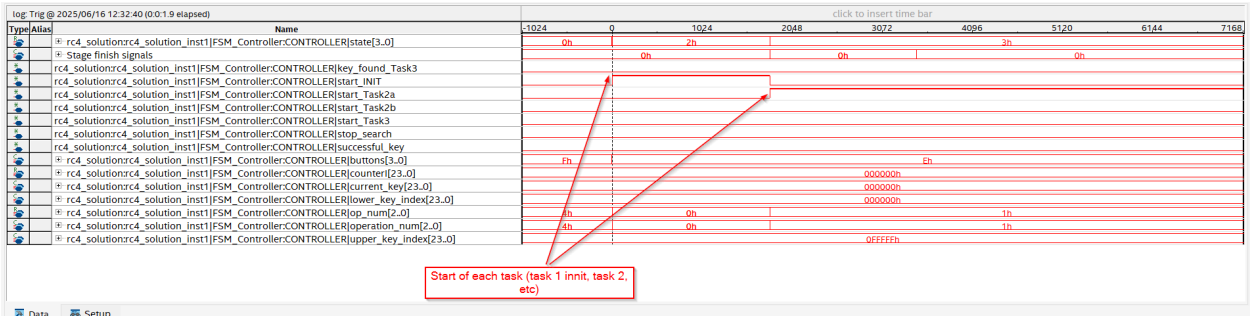
Task3_Check

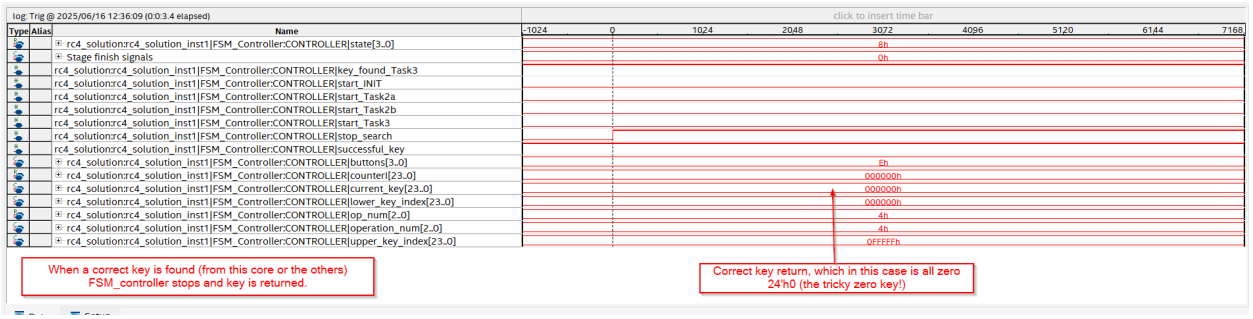
```
/*
| State                    | Hex  |
| ------------------------ | ---- |
| Idle                     | 0x00 |
| Read\_decryptedMem       | 0x04 |
| Read\_Wait\_decryptedMem | 0x08 |
| CheckValid               | 0x0C |
| IncrementI               | 0x10 |
| Finished                 | 0x15 |
*/
```



FSM_Controller

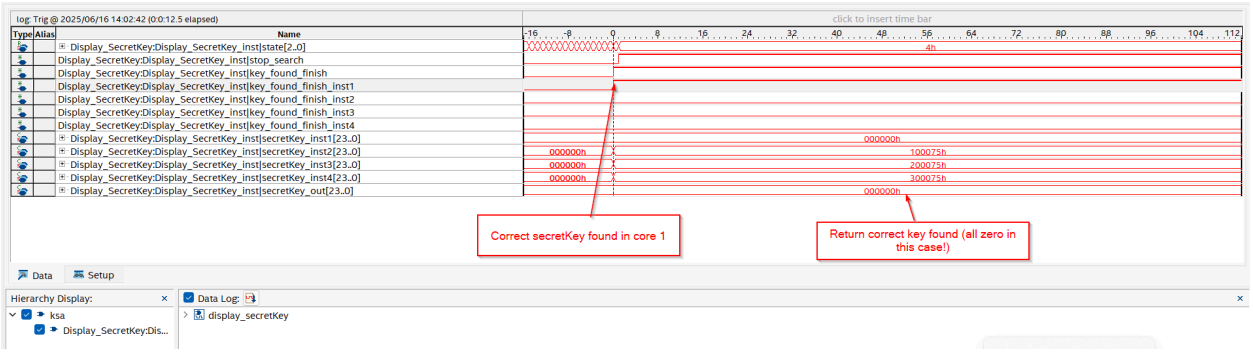
```
/*
| State      | Hex  |
| ---------- | ---- |
| IDLE       | 0x00 |
| StartInit  | 0x01 |
| Init       | 0x02 |
| Task2a     | 0x03 |
| Task2b     | 0x04 |
| Task3      | 0x05 |
| WaitValid  | 0x06 |
| WaitValid2 | 0x07 |
| Finish     | 0x08 |
*/
```





Display_SecretKey

```
/*
 | State Name | Hex Value |
 |------------|-----------|
 | INST1 | 0x0 |
 | INST2 | 0x1 |
 | INST3 | 0x2 |
 | INST4 | 0x3 |
 | Finish | 0x4 |
*/
```



5. Information on how to run the simulations (i.e. where the files are located, which program you used to run your simulation).

- Software: ModelSim.
- Testbench files are located in sim/ and sim/savedWaveForms directory. The actual module .sv files are located in rtl/
- Open ModelSim project lab1Sim.mpf file.
- Open the .do files included in the folder.
- To run then, type ‘do [waveformName].do’ in the Transcript terminal

6. Any additional information that would be relevant for the TA marking your project.

If there is any additional information required for the lab, please don’t hesitate to reach out and let us know. Thank you for marking our lab!