Lab 4 README.pdf

Lab 4 Partners

Mark Lee, 36183168

Seyed-Ebrahim Mir-Mohammadsadeghi, 84477686

Break Down

1. Directory path of SOF and solution:

rtl/output_files/rc4.sof
rtl/output_files/Chain1.cdf

2. Status

Everything in the lab is working, and according to the provided instructions. There are some caveats, however, as we will discuss below.

Task 1:

- Write 00 to FF in a RAM block – Check

Task 2:

- Write a single decryption core – Check
- Decrypt messages based on switches for key – Check

Note: the code looks for an MIF file under “rtl/message.mif”, and not “rtl/secret_messages/...”

Task 3:

- Cycle from key 24’h000_000 to 24’hFFF_FFF or until code is cracked – Check
- Display key on hex display – Check
- Success/Failure LEDS – Check
- Check for valid characters – Check
  - The code immediately exits the loop upon an invalid character to save time
  - This is a superiour method time-wise compared to completing the decryption and comparing each index at the end.

Task 3 With Bonus:

- Use 4-cores simultaneously – Unfortunately not functional

3. Annotated simulation screenshots as required by the lab

The screenshots are organized from top to bottom and in chronological order. These can be found lower in the document.

4. Information on how to run the simulations

The simulations use imitation RAM and ROM blocks written in Verilog, since the original RAM and ROM do not compile in ModelSim. The code for these imitation blocks were provided with permission by Justin Chang. This code is not used anywhere in the actual project.

For testbenching we used one massive testbench called task3_TB.sv.

5. Any additional information

Modules were created, and each has their own respective file. These files were called upon by the main “ksa.sv” file.

- “task1_fsm.sv” for task 1, the code is recycled in later stages, but not used directly
- “task2a_fsm_ebi_ver.sv” for tasks 2, 3, and 3 with bonus
- “task2b_fsm_crack.sv” for tasks 2, 3, and 3 with bonus
- “task2_fsm” which handles the task2a and task2b
- “decrypted_msg.v” for the ROM containing the output information
- “encrypted_msg.v” for the RAM containing the input information
- “task3_fsm.sv” which handles the task2 with an fsm of its own.
- “task3_bonus_fsm.sv” which handles the task2 with an fsm of its own.

There are rigorous comments in each code module that may be helpful for understanding the “what” & “why”s of the code.

Please note that comments containing “//&?&” is just a shorthand that I use to CTRL-F to areas in the professor’s code that requires adjustment.

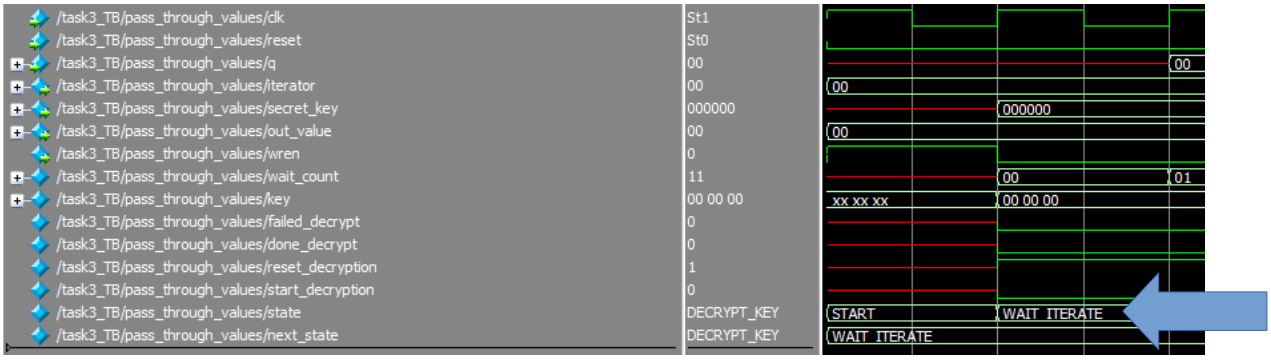
Annotated Screenshots

Each FSM here is shown.

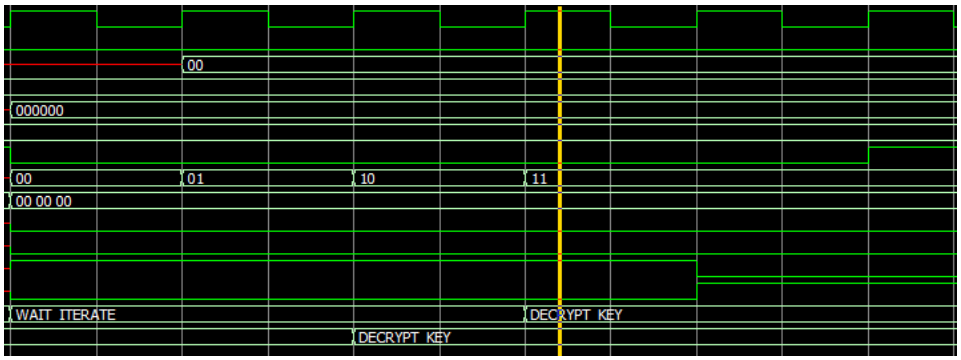
Most of these images are from the simulation of brute force on message 1, it has a small key, which makes simulating it being cracked on ModelSim far more tolerable on my computer.

task3_fsm

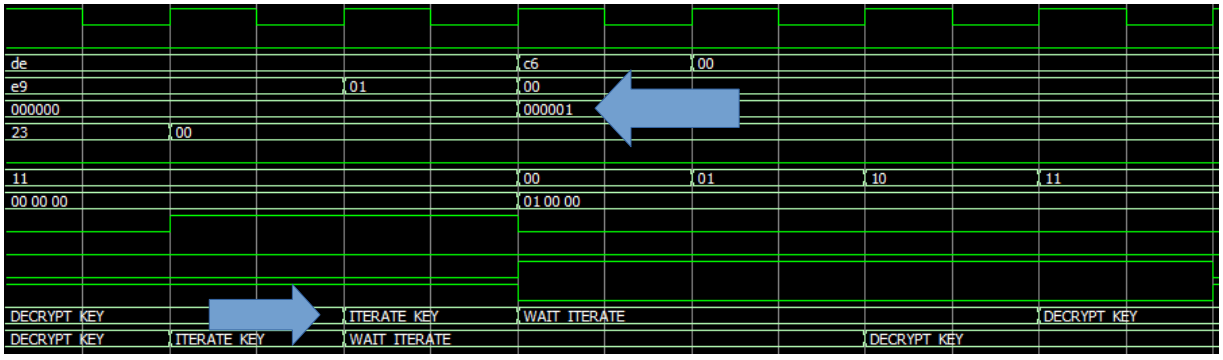
This FSM's order is "ITERATE_KEY", then "WAIT_ITERATE", "DECRYPT_KEY", and loop around until the process has been completed. However the first "ITERATE_KEY" state is skipped, so that the all zeros key is not ignored as a possibility.



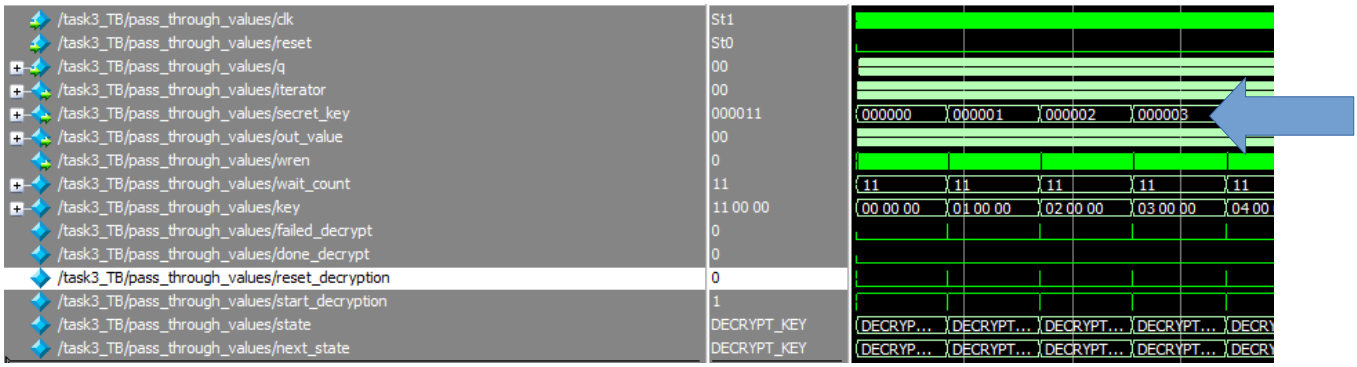
After "WAIT_ITERATE" comes the "DECRYPT_KEY" state, where all the subsequent FSMs get called. Naturally, this is the state that this FSM stays in the longest.



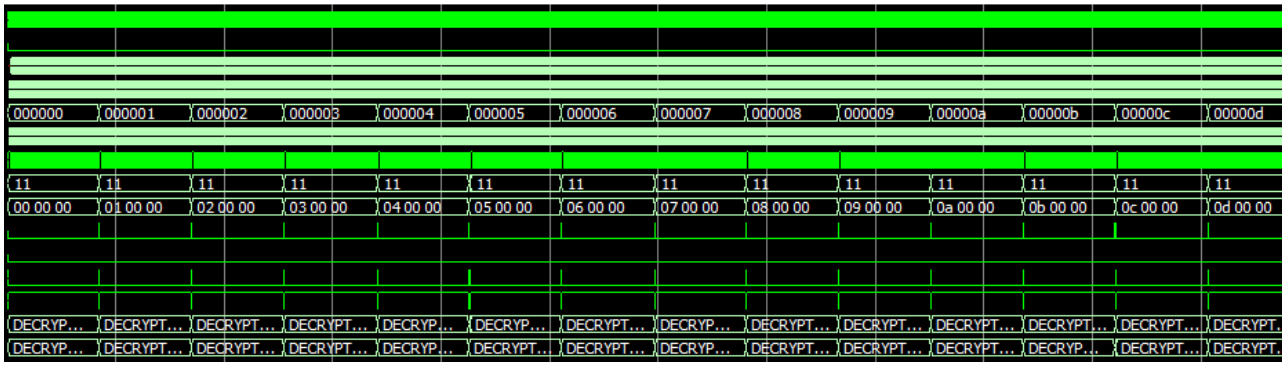
If the decryption fails for that particular key, we then iterate the key.



This overall process is continually repeated.

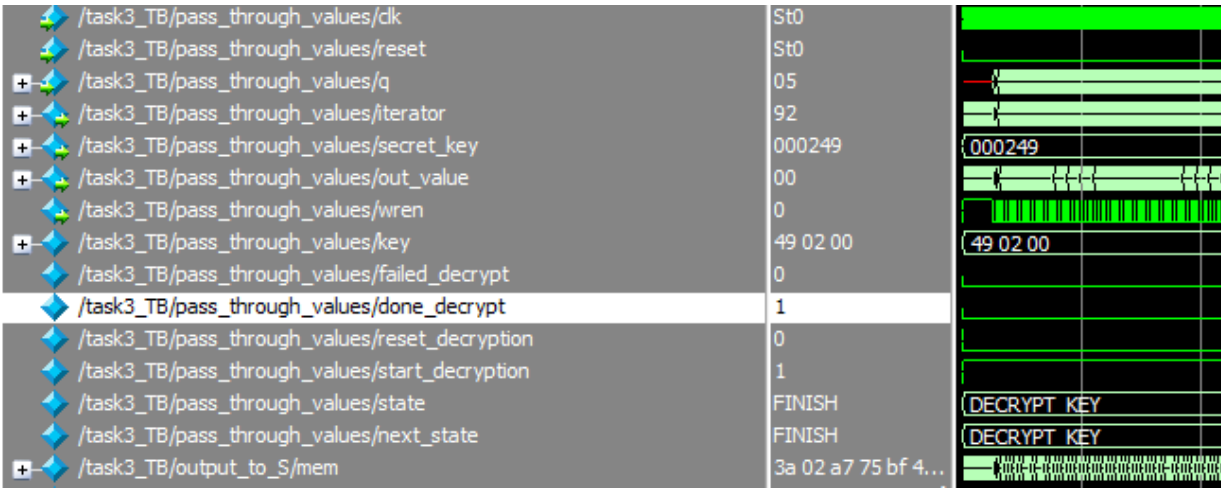


Over and over again.

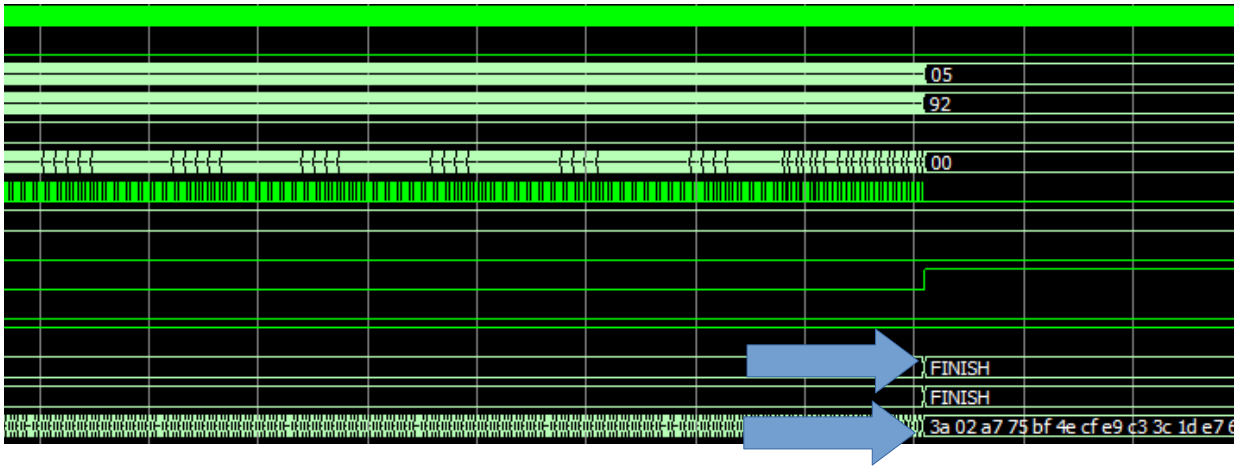


Let's now go back to the fact that this FSM simply manages others.

This FSM is a very simple one, immediately after starting, it calls the decryption core and makes it start with the key it provides (and iterates).



Once the decryption module (task2_fsm) completes its work, it sends a "done_decrypt" signal, and the task3_fsm enters its "FINISH" state, where RAM is suspended and unchanged.

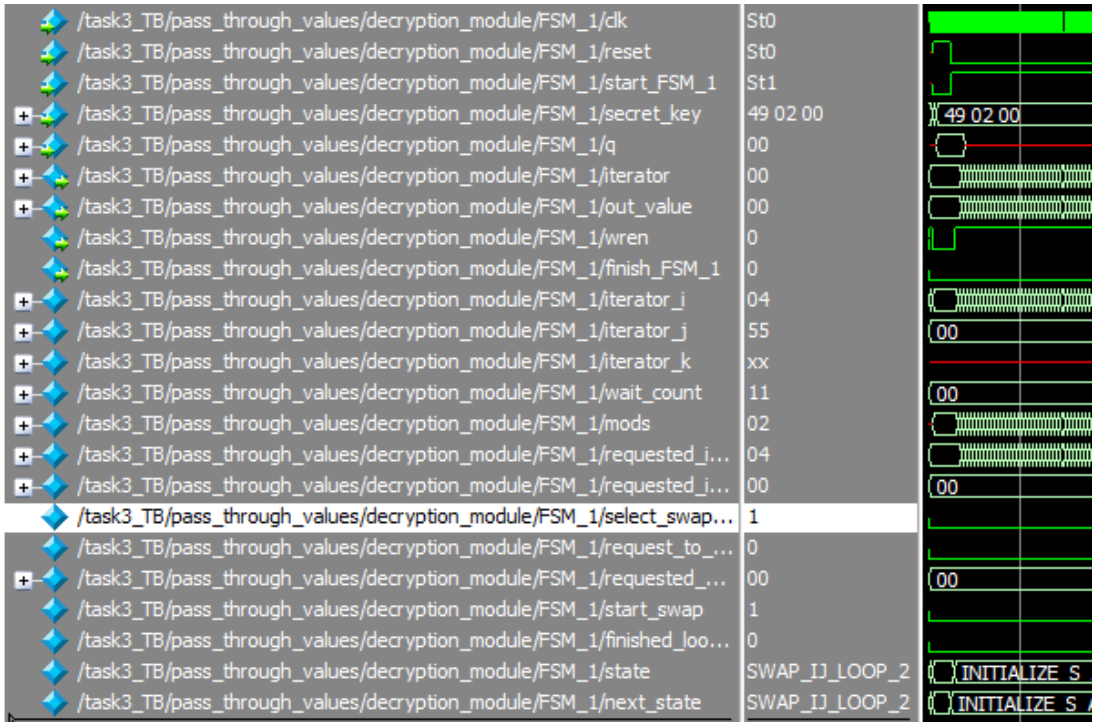


task2_fsm

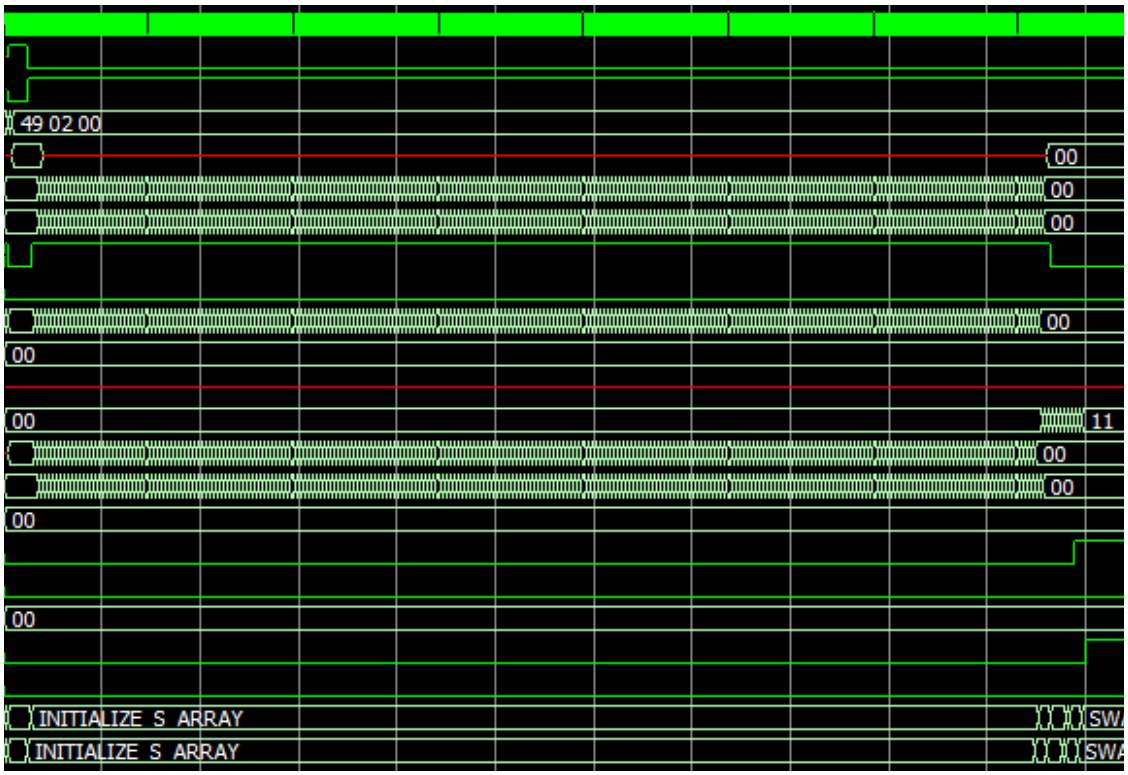
This is nominally labelled “fsm” but it is simply a collection of wires, regs, and muxes to handle the combination of task2a_fsm and the task2b_fsm. And does not have any states.

task2a_fsm

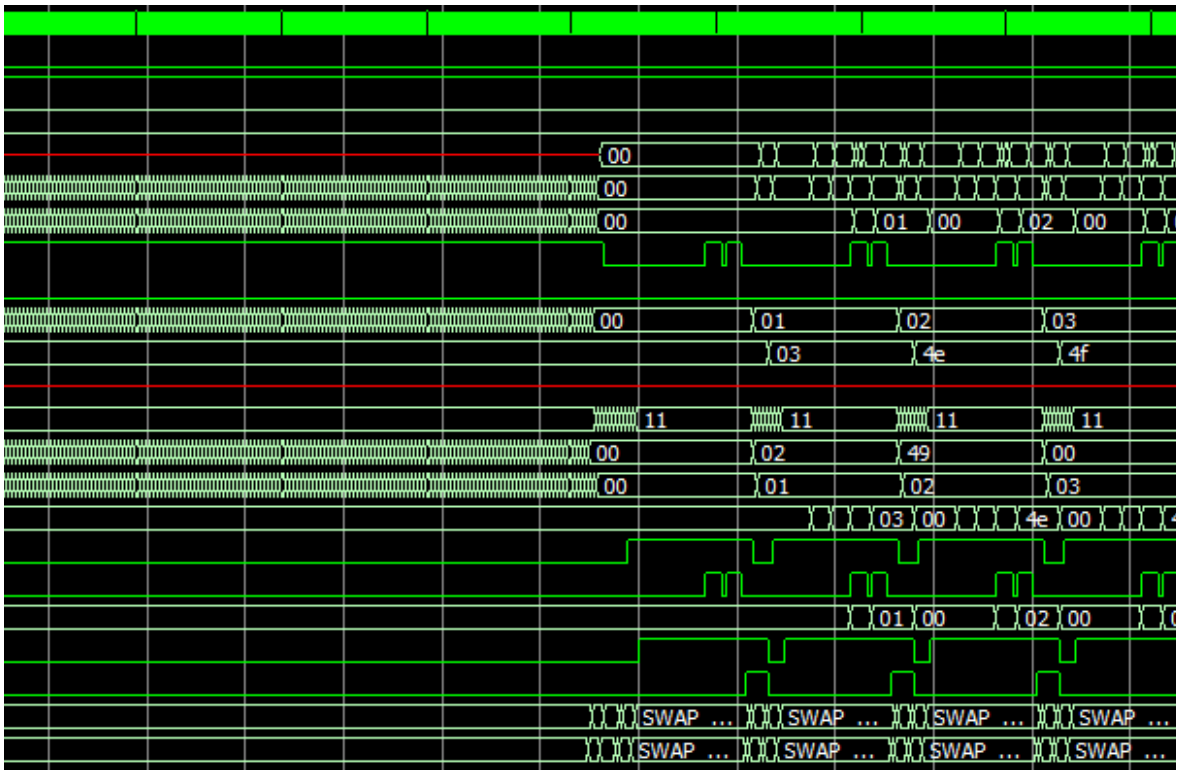
The task2a fsm does not use iterator_k, the wire is from a slightly previous version of the code, which has been removed. The code has been tweaked for extra wires such as these.



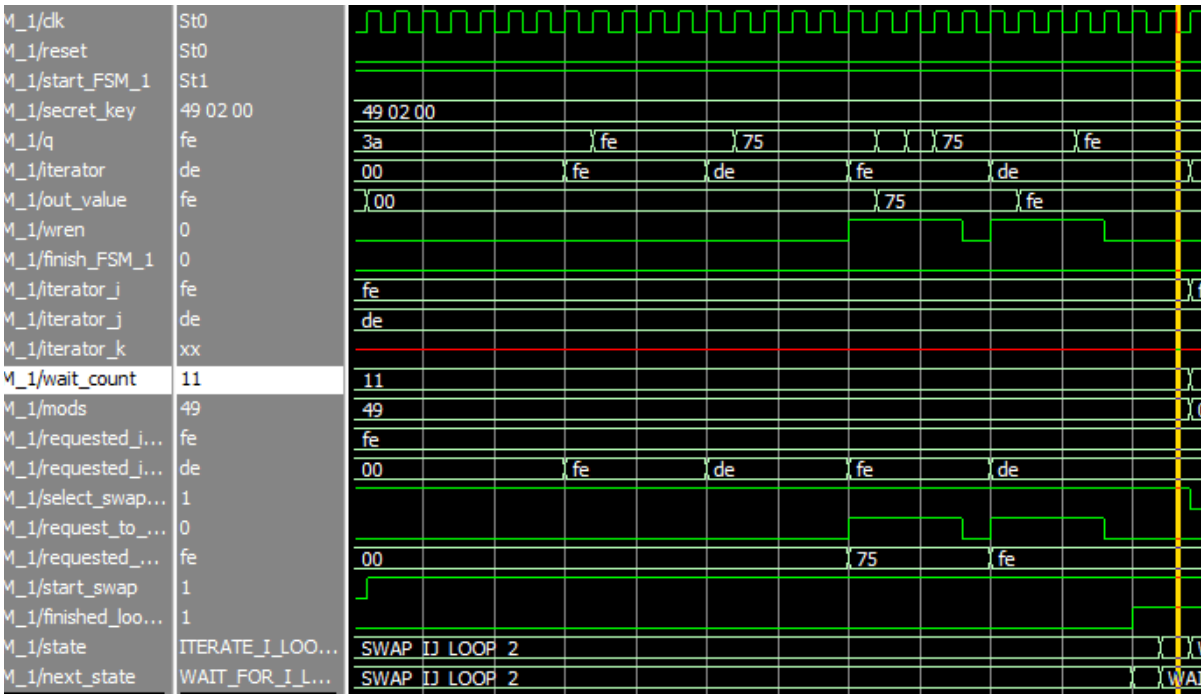
The task2a_fsm starts by doing the first loop, and by placing the values into S_memory. During this process, the value of “q” is irrelevant to us so it is seen as don’t cares.



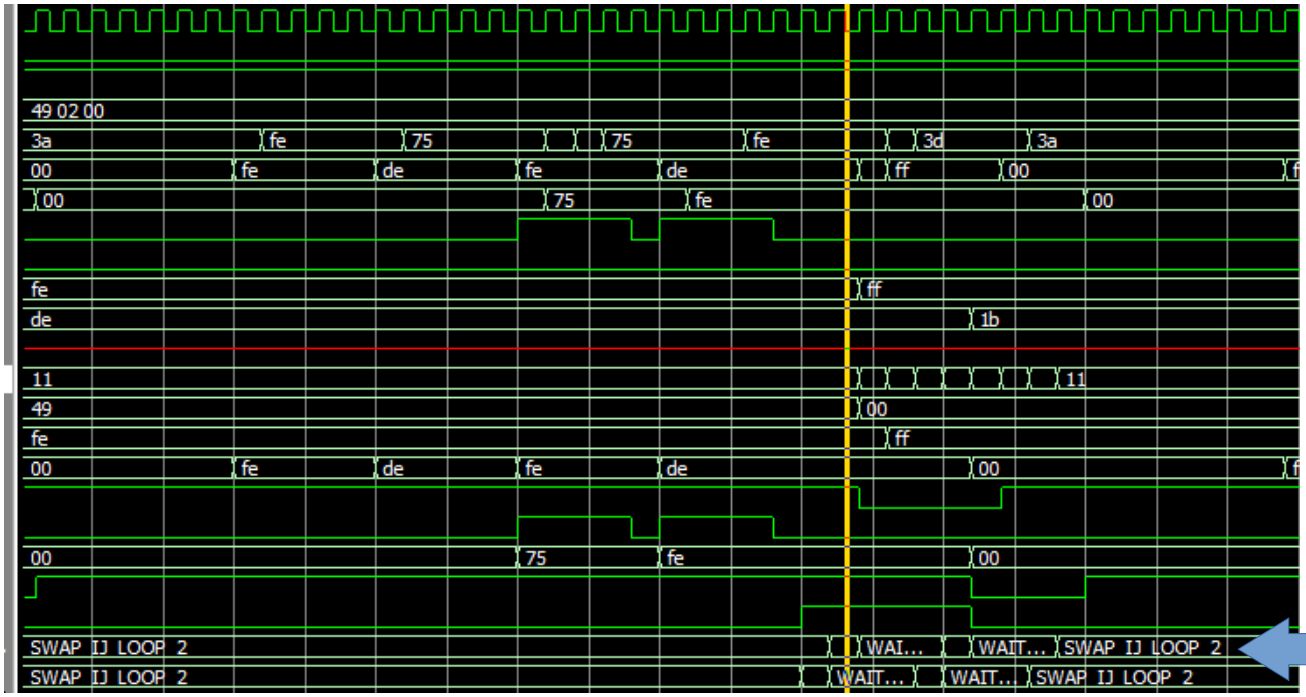
Once the S_memory has been filled from 00 to FF, it continues onto the second loop. Here the values of iterator_i and iterator_j are incremented, and then the swap_ij_fsm is called.



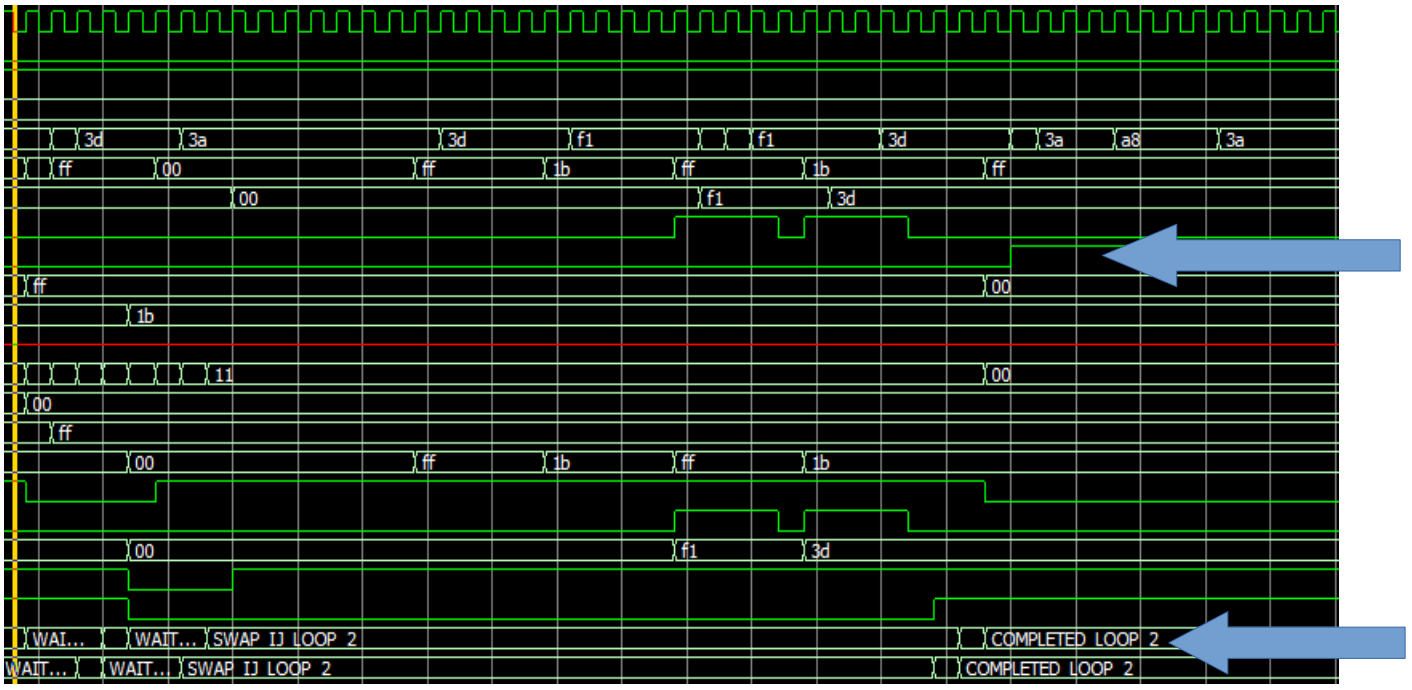
From a more zoomed in perspective we can see that while this FSM is in the SWAP_IJ_LOOP_2 state, the swap_fsm is changing values.



After the swap_ij_fsm finishes, the task2a_fsm iterates iterator_i, waits, iterates iterator_j, waits again, and then calls the swap_ij_fsm once more.

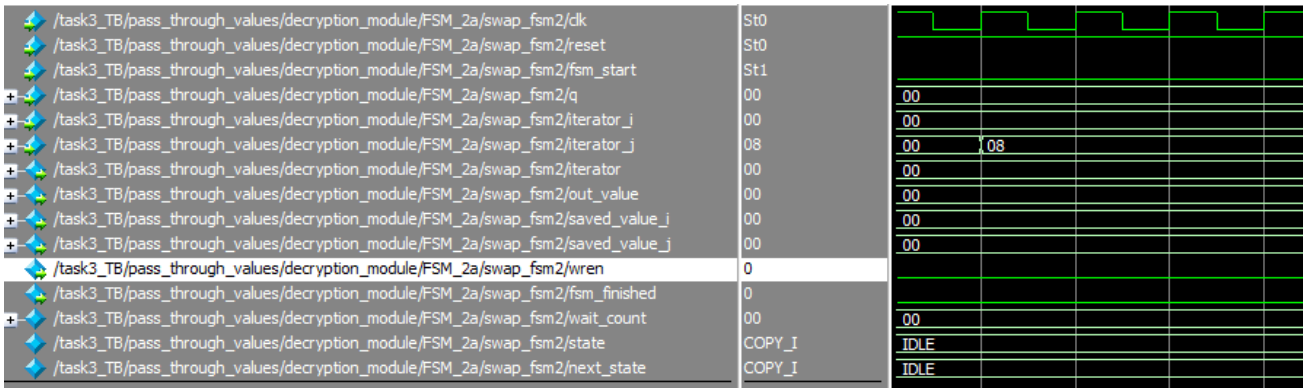


This process is repeated until loop_2 is completed. Where we can see the finish_FSM_1 wire is set high.

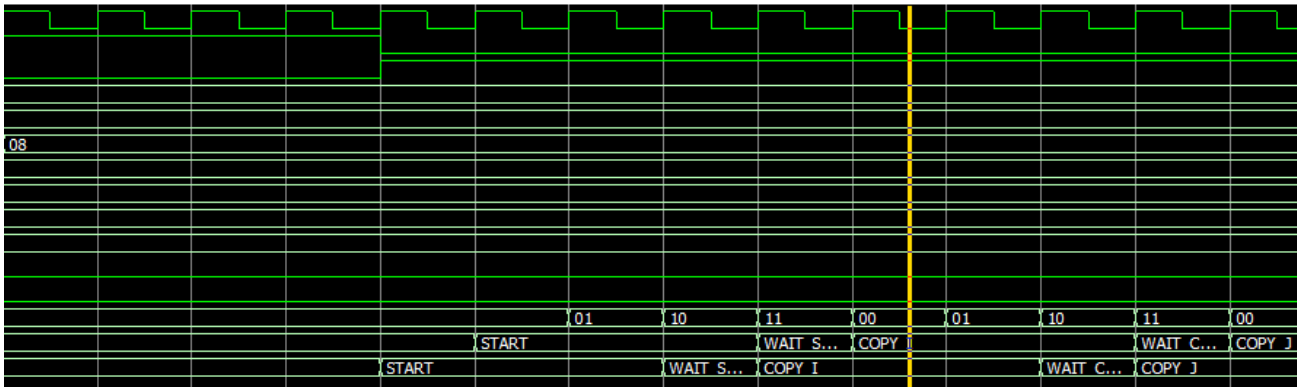


task2_swap_ij_fsm

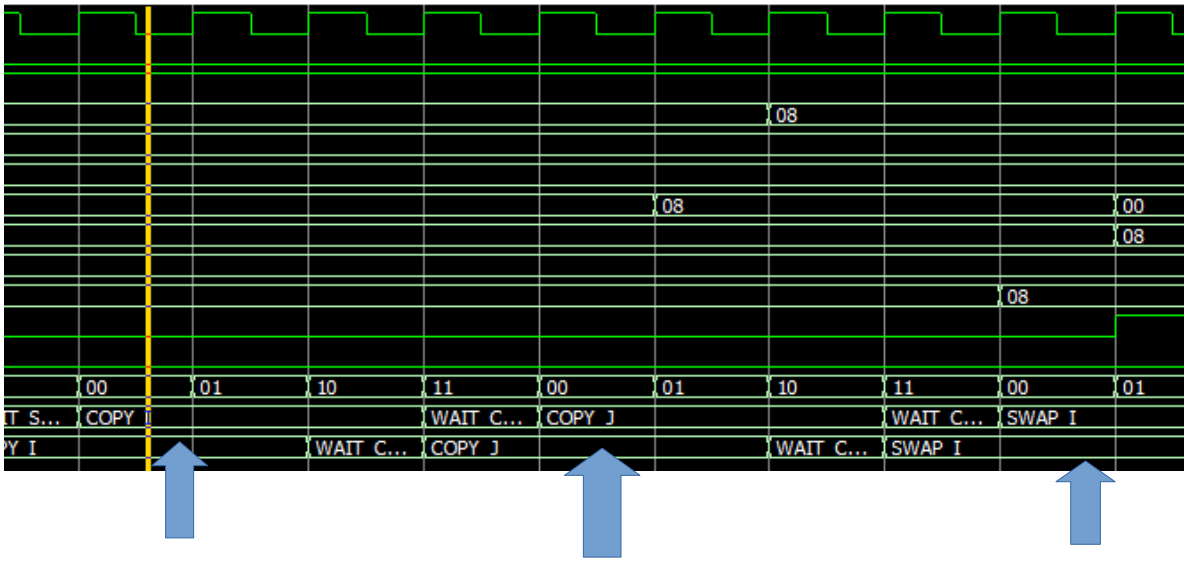
As mentioned earlier this fsm is used in both task2a and task2b. Here it is held in a suspended state until it is told to start.



Here we can see that “fsm_reset” was disabled, and “fsm_start” was enabled. Immediately we exit the “IDLE” state and give the S_memory time to gather the necessary information while in the “START” and “WAIT_START” states. “IDLE” initializes values, “START” increments a counter for number of cycles to wait, and “WAIT_START” restarts that counter, so that the next state can use the same reg for its own counter.

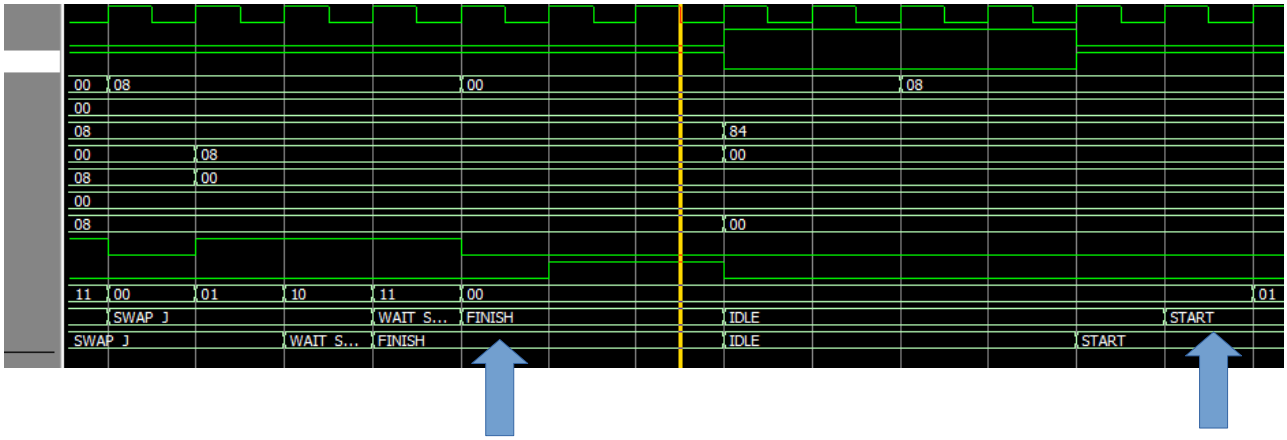


After the value from location s[i] is saved, we go to the location for s[j] and saved that value as well. We then place the value from s[j] into the location at s[i] in the “SWAP_I” state.



Then the earlier value from s[i] is placed in the location at s[j] in the “SWAP_J” state. After a wait state, the swap_ij_fsm is finished, and is ready to be reset and called again.

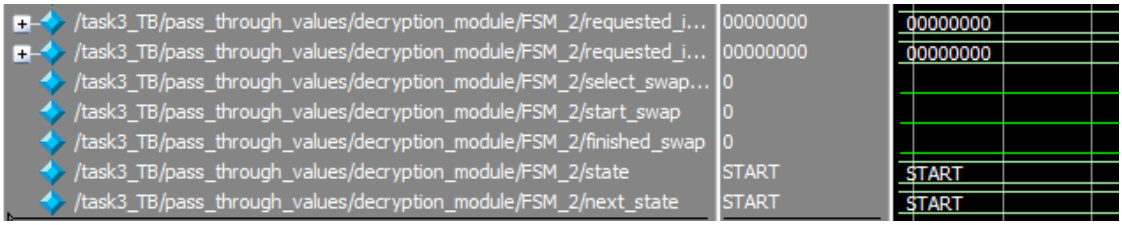
The below image is spliced down the middle (at the yellow bar) for easier viewing.



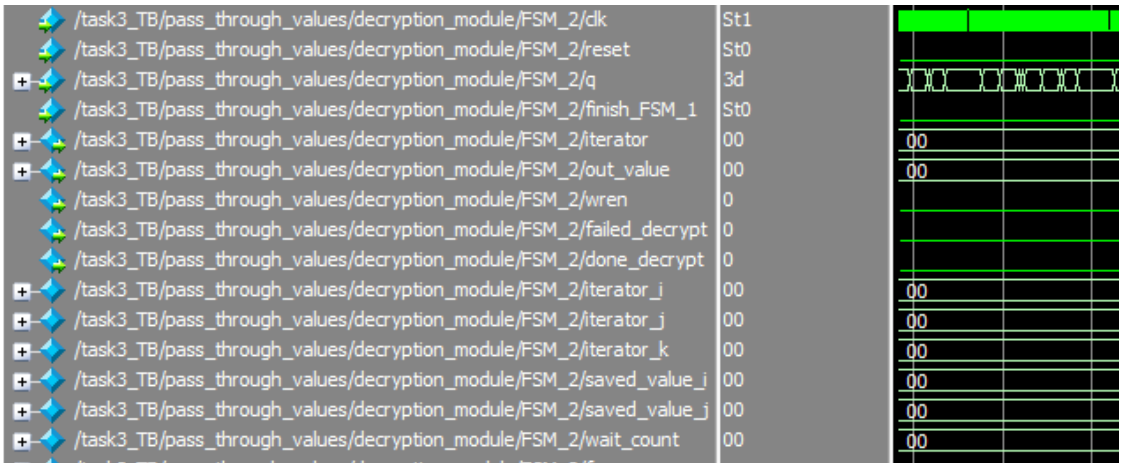
An FSM diagram has been included at the end of the document for the swap_ij_fsm.

task2b_fsm

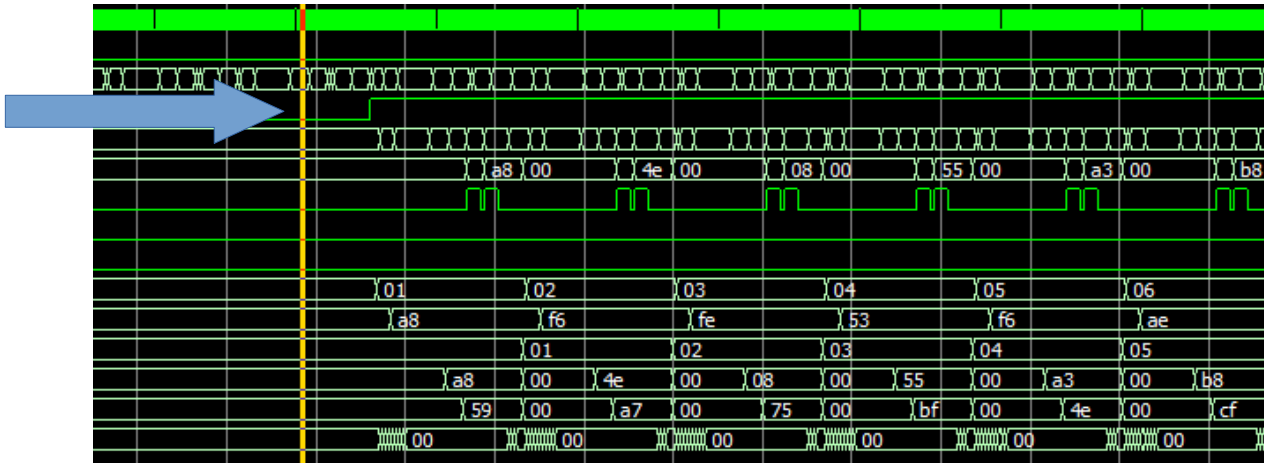
We start in the “START” state,



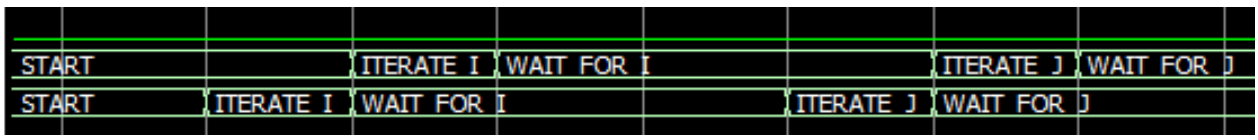
Here all our values are initialized to be zeros.



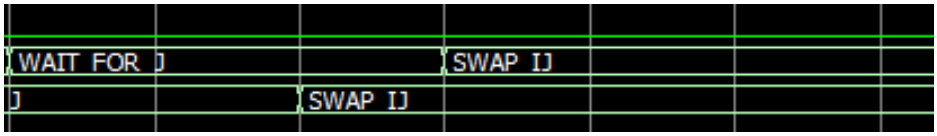
momentarily the finish_FSM_1 signal comes in, and the FSM starts iterating through loop 3.



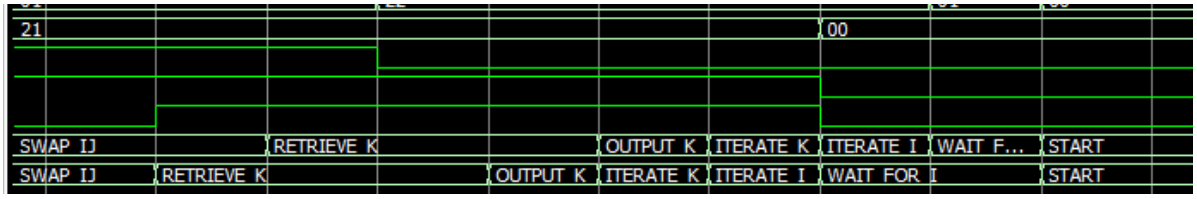
Here the FSM goes into ITERATE_I, WAIT_FOR_I, ITERATE_J, WAIT_FOR_J,



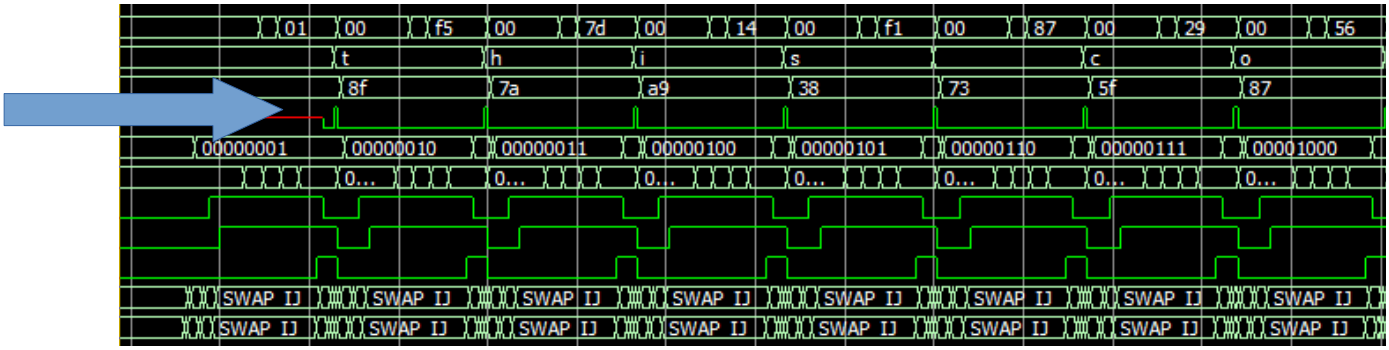
and then SWAP_IJ.



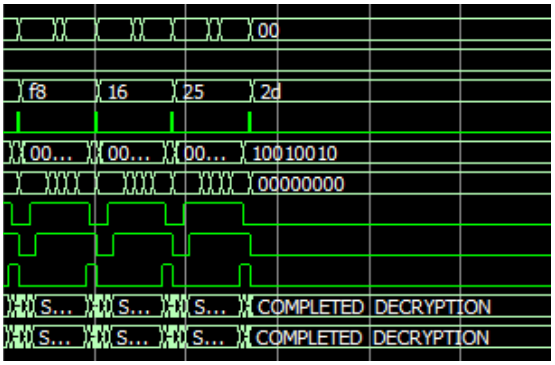
We then retrieve the information from the encrypted RAM in the “RETRIEVE_K” state.



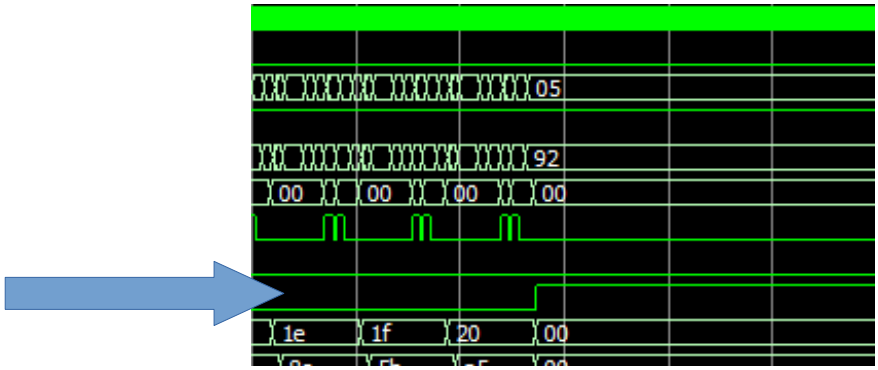
From there the value for k is used to access a location in memory in both the RAM and ROM. The moment where the encrypted memory is first called and q_m becomes valid is apparent.



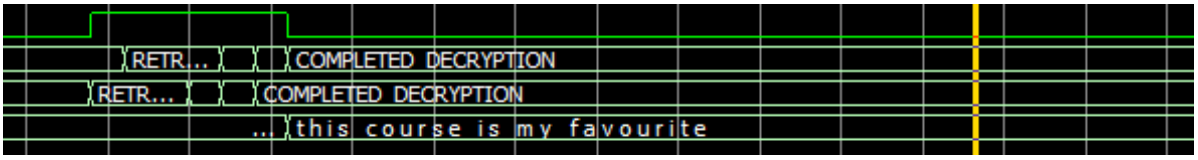
Eventually the decryption is completed.



and we can see the “done_decrypt” signal is set high.



As well as the result.



Messages

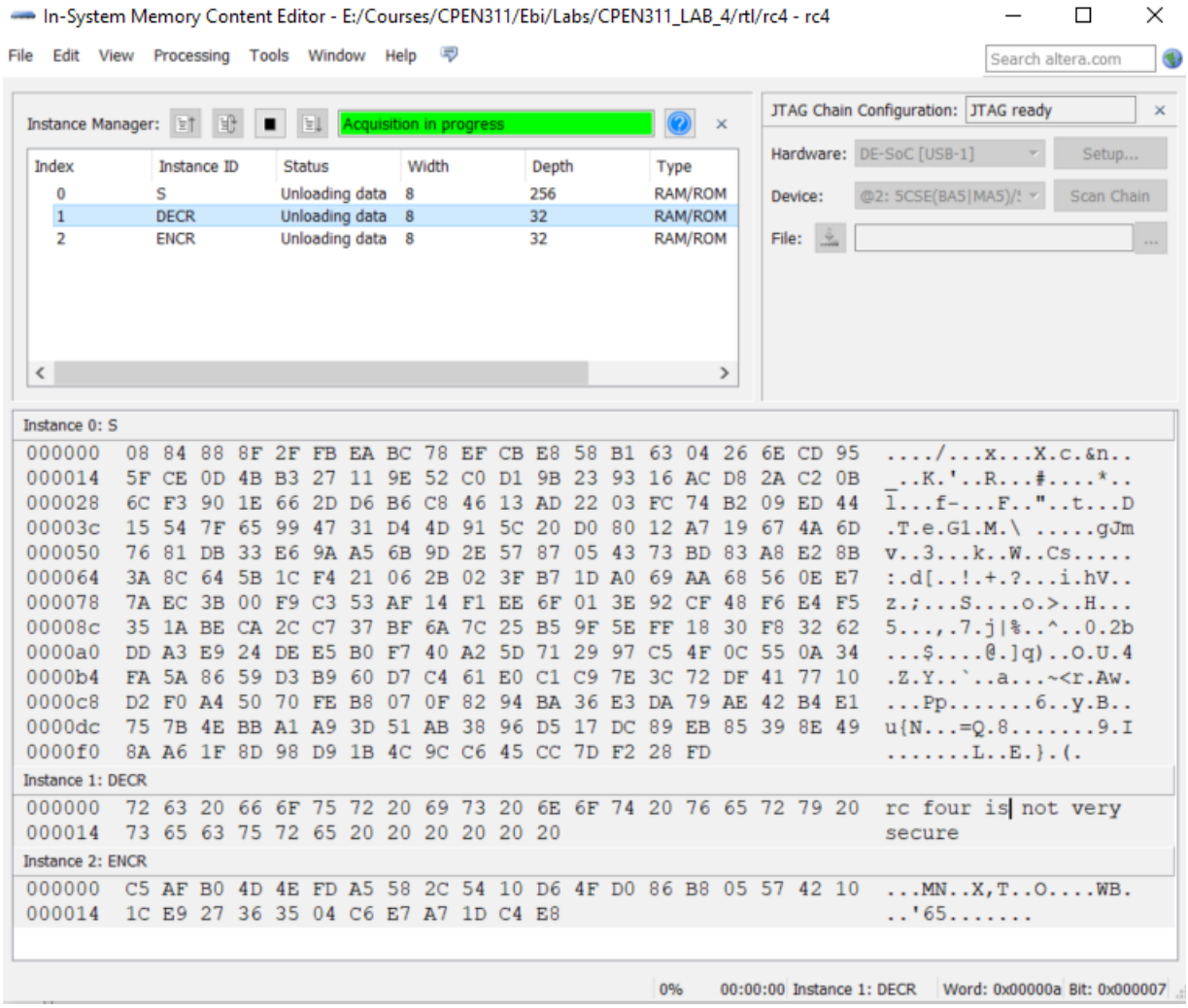
Messages 1-3,

Key = 249: "this course is my favourite "

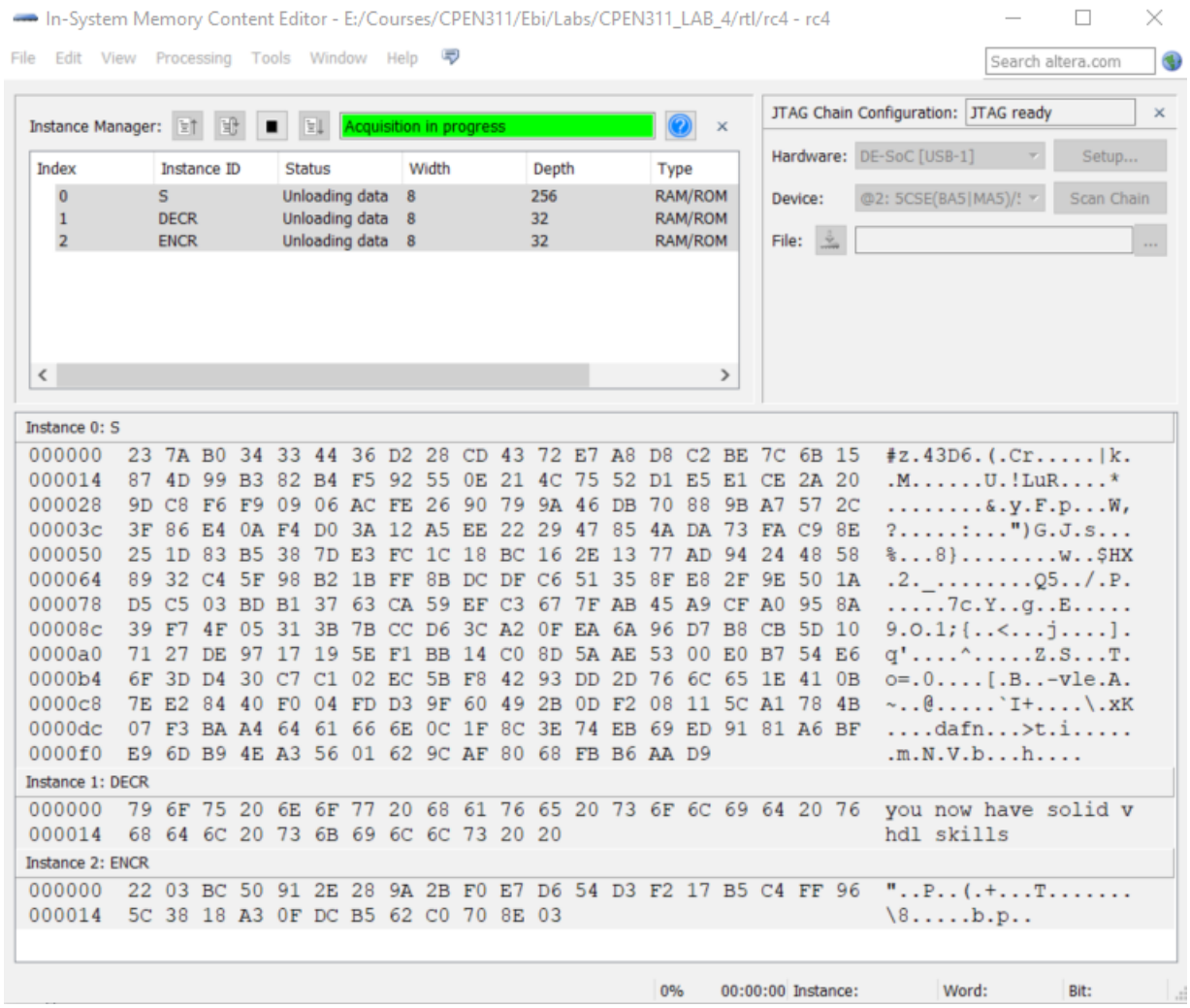
Key = 3FF: "congratulations on task two "

Key = 2AA: "ubc elec and comp engineering "

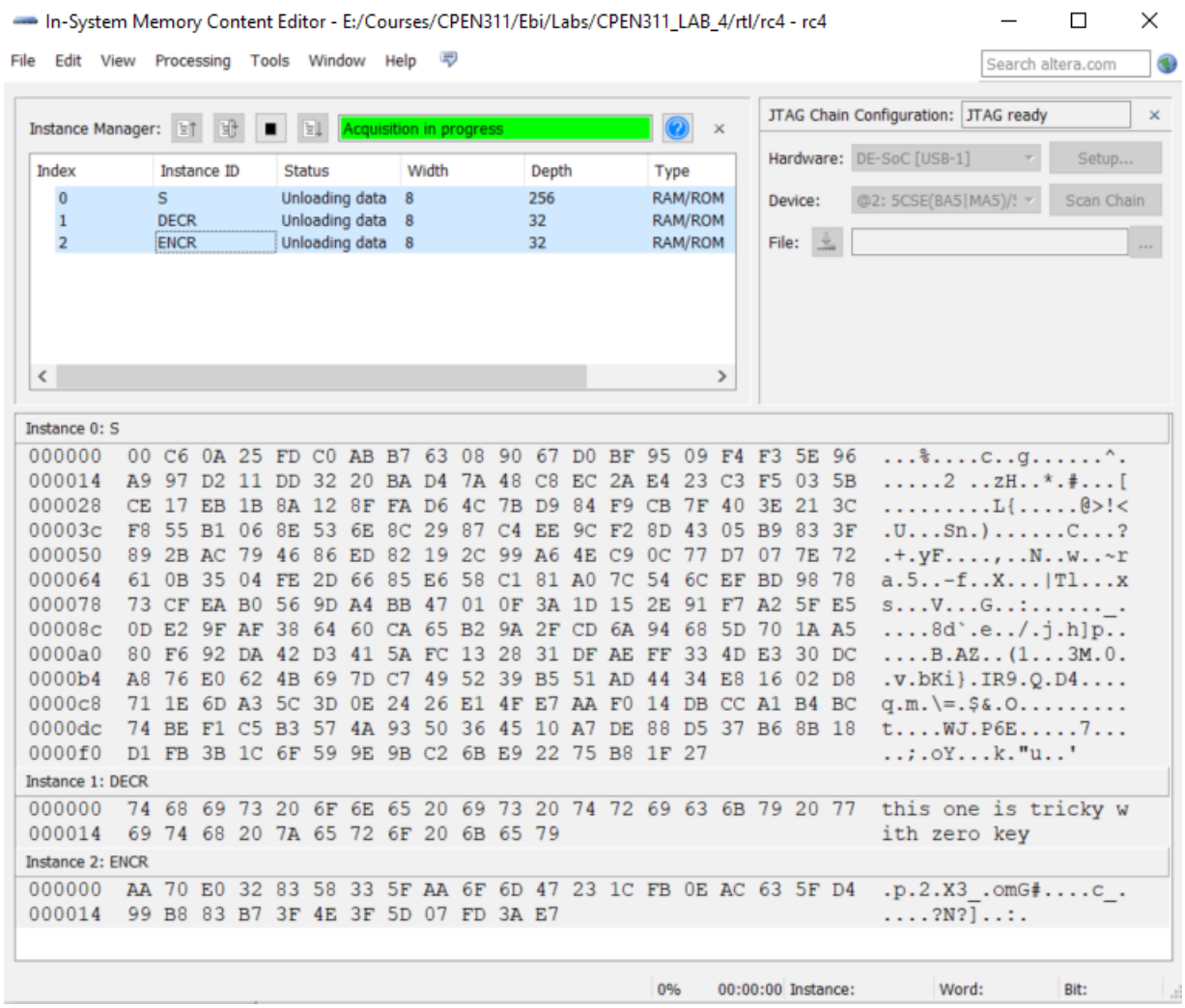
Message 4, Key: 087B2D



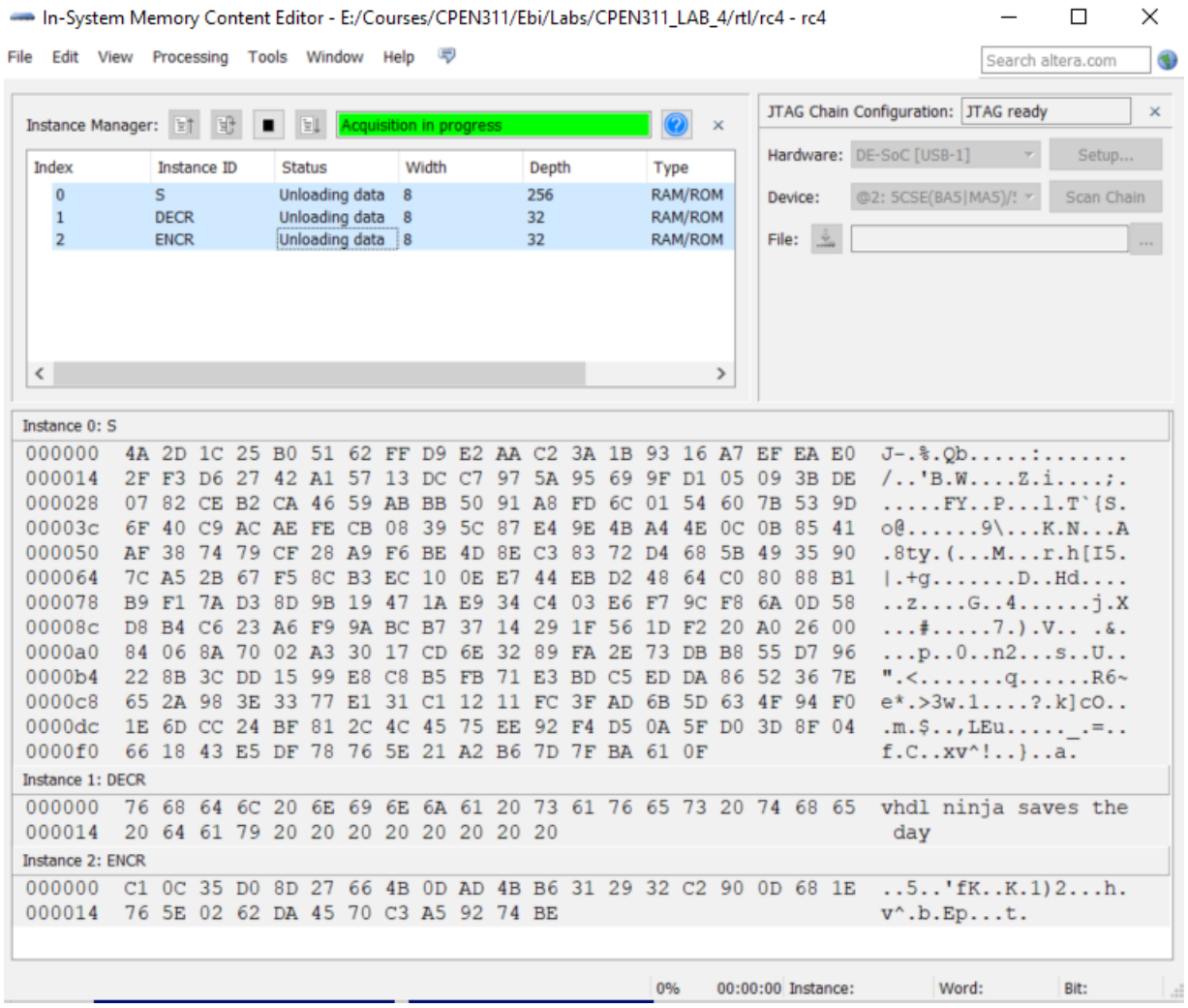
Message 5, Key: 04000C



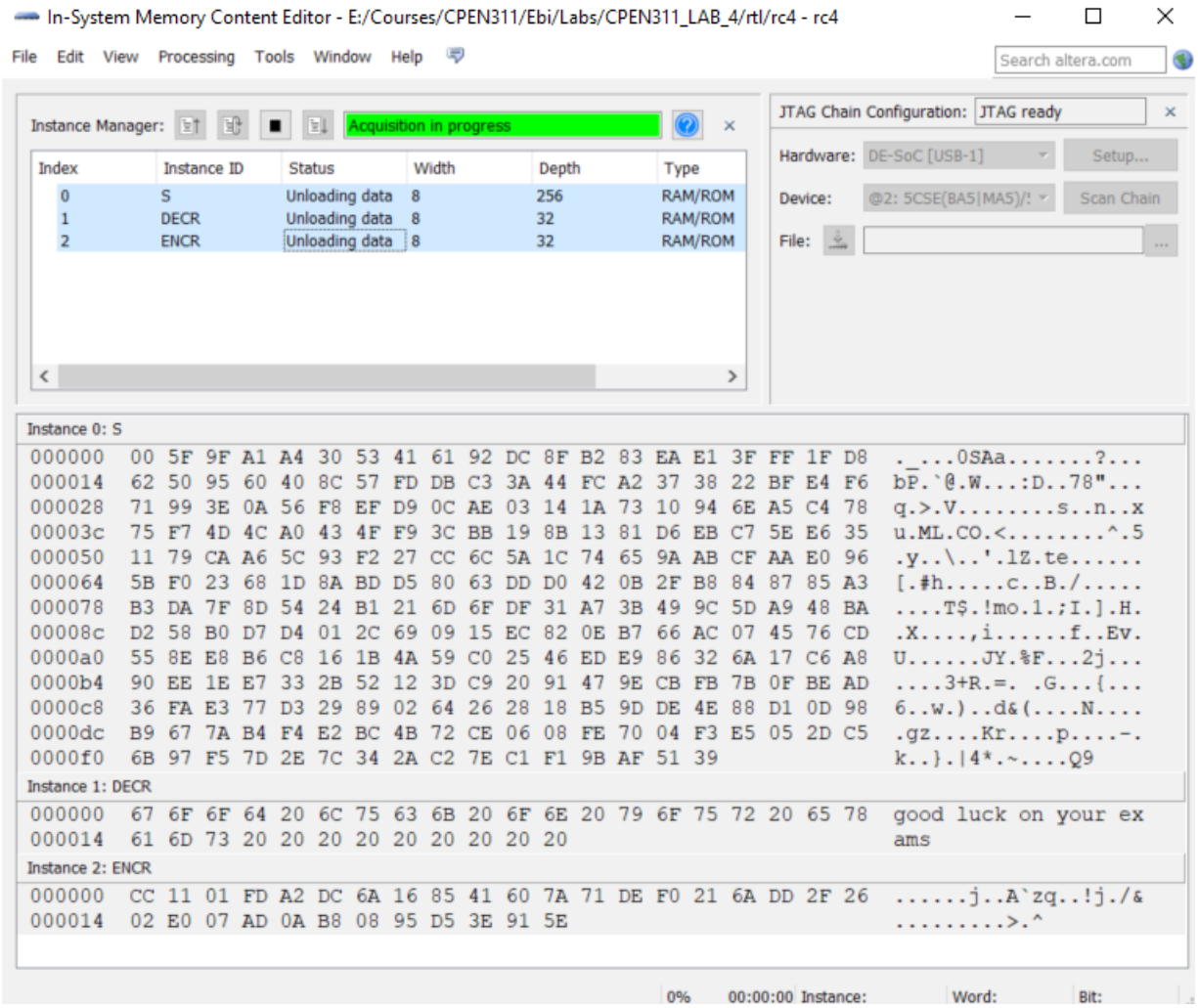
Message 6, Key: 000000



Message 7, Key: 02640F


In-System Memory Content Editor - E:/Courses/CPEN311/Ebi/Labs/CPEN311_LAB_4/rtl/rc4 - rc4
File Edit View Processing Tools Window Help
Search altera.com
Instance Manager:
Acquisition in progress
Index Instance ID Status Width Depth Type
0 S Unloading data 8 256 RAM/ROM
1 DECR Unloading data 8 32 RAM/ROM
2 ENCR Unloading data 8 32 RAM/ROM
JTAG Chain Configuration: JTAG ready
Hardware: DE-SoC [USB-1] Setup...
Device: @2: 5CSE(BA5|MA5)/ Scan Chain
File:
Instance 0: S
000000 4A 2D 1C 25 B0 51 62 FF D9 E2 AA C2 3A 1B 93 16 A7 EF EA E0 J-.%.Qb.....:.....
000014 2F F3 D6 27 42 A1 57 13 DC C7 97 5A 95 69 9F D1 05 09 3B DE /..'B.W....Z.i....;.
000028 07 82 CE B2 CA 46 59 AB BB 50 91 A8 FD 6C 01 54 60 7B 53 9D .....FY..P...l.T`{S.
00003c 6F 40 C9 AC AE FE CB 08 39 5C 87 E4 9E 4B A4 4E 0C 0B 85 41 o@......9\...K.N...A
000050 AF 38 74 79 CF 28 A9 F6 BE 4D 8E C3 83 72 D4 68 5B 49 35 90 .8ty.(...M...r.h[I5.
000064 7C A5 2B 67 F5 8C B3 EC 10 0E E7 44 EB D2 48 64 C0 80 88 B1 |.+g.......D..Hd....
000078 B9 F1 7A D3 8D 9B 19 47 1A E9 34 C4 03 E6 F7 9C F8 6A 0D 58 ..z....G..4......j.X
00008c D8 B4 C6 23 A6 F9 9A BC B7 37 14 29 1F 56 1D F2 20 A0 26 00 ...#.....7.).V.. .&.
0000a0 84 06 8A 70 02 A3 30 17 CD 6E 32 89 FA 2E 73 DB B8 55 D7 96 ...p..0..n2...s..U..
0000b4 22 8B 3C DD 15 99 E8 C8 B5 FB 71 E3 BD C5 ED DA 86 52 36 7E ".<.......q......R6~
0000c8 65 2A 98 3E 33 77 E1 31 C1 12 11 FC 3F AD 6B 5D 63 4F 94 F0 e*.>3w.1....?.k]cO..
0000dc 1E 6D CC 24 BF 81 2C 4C 45 75 EE 92 F4 D5 0A 5F D0 3D 8F 04 .m.$..,LEu....._.=..
0000f0 66 18 43 E5 DF 78 76 5E 21 A2 B6 7D 7F BA 61 0F f.C..xv^!..}..a.
Instance 1: DECR
000000 76 68 64 6C 20 6E 69 6E 6A 61 20 73 61 76 65 73 20 74 68 65 vhdl ninja saves the
000014 20 64 61 79 20 20 20 20 20 20 20 20 day
Instance 2: ENCR
000000 C1 0C 35 D0 8D 27 66 4B 0D AD 4B B6 31 29 32 C2 90 0D 68 1E ..5..'fK..K.1)2...h.
000014 76 5E 02 62 DA 45 70 C3 A5 92 74 BE v^.b.Ep...t.
0% 00:00:00 Instance: Word: Bit:

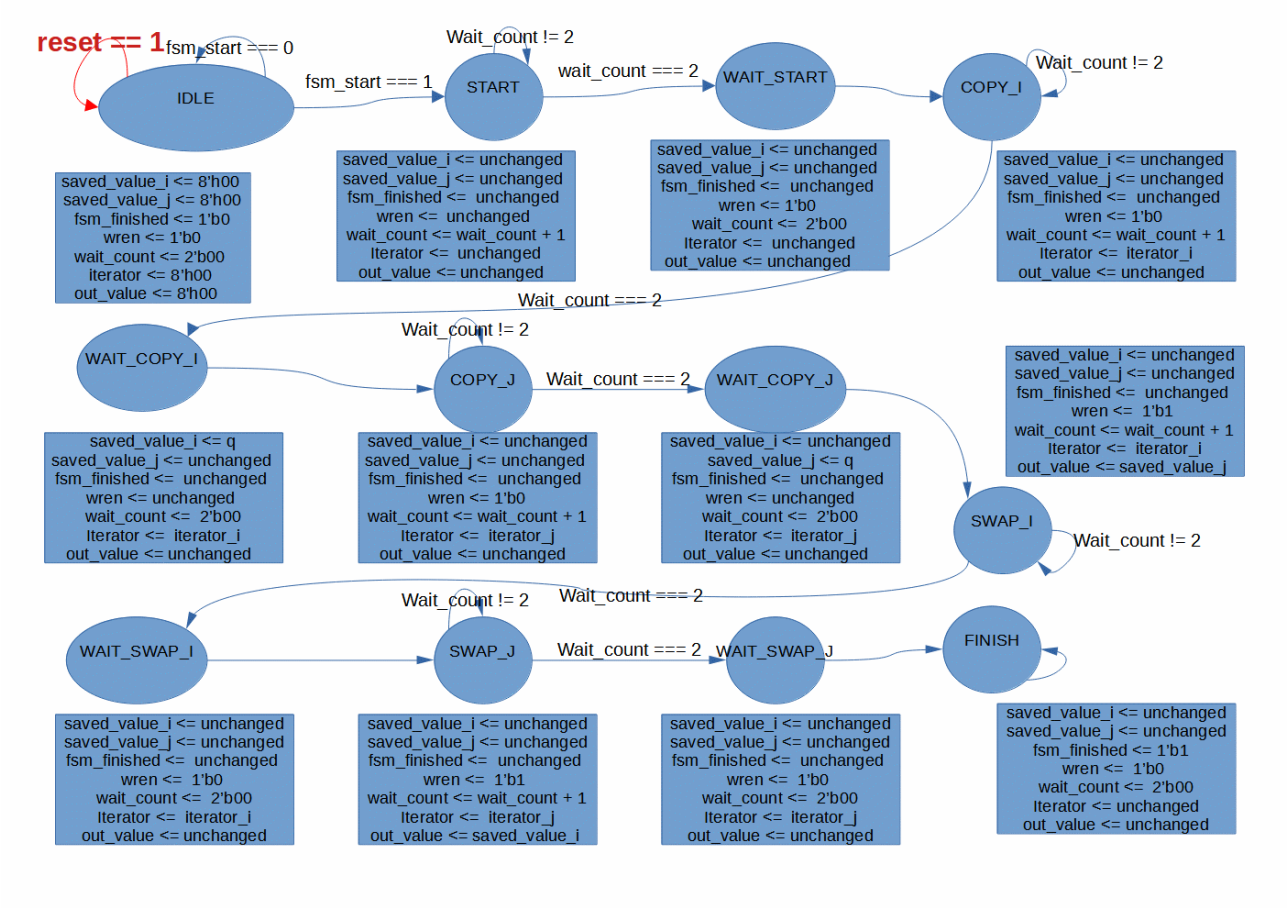
Message 8, Key: 3FFFFF



Reference

SWAP_IJ_FSM Diagram

Reset has not been drawn on all states for the sake of visual simplicity. In the actual desing, the FSM can be reset at any moment.


reset == 1
fsm_start === 0
IDLE
fsm_start === 1
START
Wait_count != 2
wait_count === 2
WAIT_START
COPY_I
Wait_count != 2
saved_value_i <= 8'h00
saved_value_j <= 8'h00
fsm_finished <= 1'b0
wren <= 1'b0
wait_count <= 2'b00
iterator <= 8'h00
out_value <= 8'h00
saved_value_i <= unchanged
saved_value_j <= unchanged
fsm_finished <= unchanged
wren <= unchanged
wait_count <= wait_count + 1
Iterator <= unchanged
out_value <= unchanged
saved_value_i <= unchanged
saved_value_j <= unchanged
fsm_finished <= unchanged
wren <= 1'b0
wait_count <= 2'b00
Iterator <= unchanged
out_value <= unchanged
saved_value_i <= unchanged
saved_value_j <= unchanged
fsm_finished <= unchanged
wren <= 1'b0
wait_count <= wait_count + 1
Iterator <= iterator_i
out_value <= unchanged
Wait_count === 2
WAIT_COPY_I
Wait_count != 2
COPY_J
Wait_count === 2
WAIT_COPY_J
saved_value_i <= unchanged
saved_value_j <= unchanged
fsm_finished <= unchanged
wren <= 1'b1
wait_count <= wait_count + 1
Iterator <= iterator_i
out_value <= saved_value_j
saved_value_i <= q
saved_value_j <= unchanged
fsm_finished <= unchanged
wren <= unchanged
wait_count <= 2'b00
Iterator <= iterator_i
out_value <= unchanged
saved_value_i <= unchanged
saved_value_j <= unchanged
fsm_finished <= unchanged
wren <= 1'b0
wait_count <= wait_count + 1
Iterator <= iterator_j
out_value <= unchanged
saved_value_i <= unchanged
saved_value_j <= q
fsm_finished <= unchanged
wren <= unchanged
wait_count <= 2'b00
Iterator <= iterator_j
out_value <= unchanged
SWAP_I
Wait_count != 2
Wait_count === 2
Wait_count != 2
WAIT_SWAP_I
SWAP_J
Wait_count === 2
WAIT_SWAP_J
FINISH
saved_value_i <= unchanged
saved_value_j <= unchanged
fsm_finished <= unchanged
wren <= 1'b0
wait_count <= 2'b00
Iterator <= iterator_i
out_value <= unchanged
saved_value_i <= unchanged
saved_value_j <= unchanged
fsm_finished <= unchanged
wren <= 1'b1
wait_count <= wait_count + 1
Iterator <= iterator_j
out_value <= saved_value_i
saved_value_i <= unchanged
saved_value_j <= unchanged
fsm_finished <= unchanged
wren <= 1'b0
wait_count <= 2'b00
Iterator <= iterator_j
out_value <= unchanged
saved_value_i <= unchanged
saved_value_j <= unchanged
fsm_finished <= 1'b1
wren <= 1'b0
wait_count <= 2'b00
Iterator <= unchanged
out_value <= unchanged