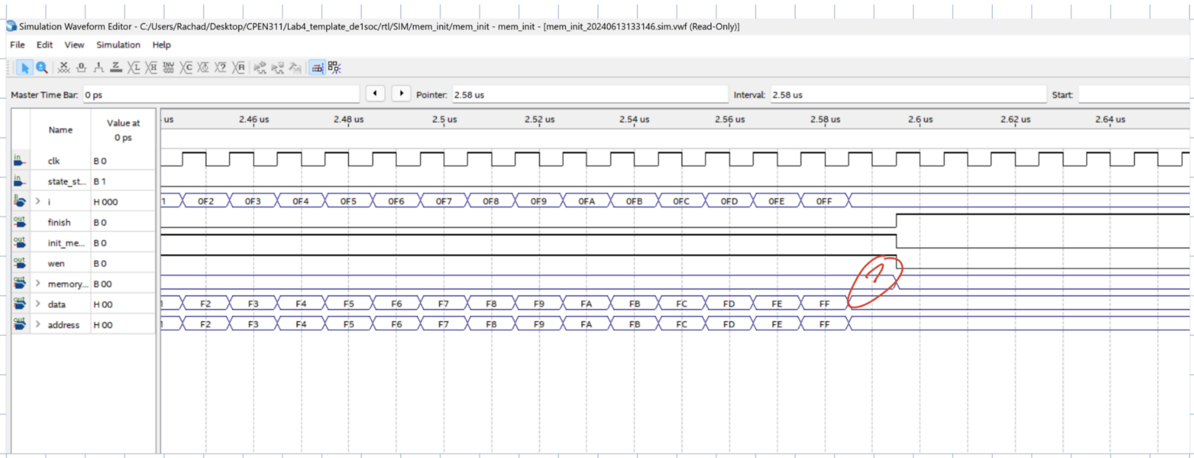
The SOF file is located int rtl/output_files directory and is called rc4.sof.

Everything works, including bonus.

Simulations are comprised of both quartus waveform and mpf files. The mem_shuffle module is simulated using quartus, while everything else is simulated using modelsim.

Simulations

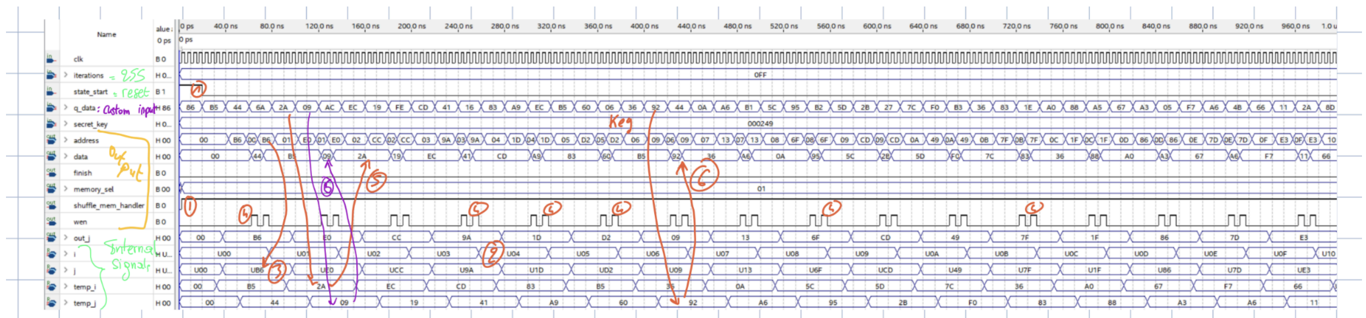
TASK1 MODULE


Simulation Waveform Editor - C:/Users/Rachad/Desktop/CPEN311/Lab4_template_de1soc/rtl/SIM/mem_init/mem_init - mem_init - [mem_init_20240613133146.sim.vwf (Read-Only)]
File Edit View Simulation Help
Master Time Bar: 0 ps
Pointer: 2.58 us
Interval: 2.58 us
Start:
Name
Value at 0 ps
clk B 0
state_st... B 1
i H 000
finish B 0
init_me... B 0
wen B 0
memory... B 00
data H 00
address H 00
2.46 us
2.48 us
2.5 us
2.52 us
2.54 us
2.56 us
2.58 us
2.6 us
2.62 us
2.64 us
0F2 0F3 0F4 0F5 0F6 0F7 0F8 0F9 0FA 0FB 0FC 0FD 0FE 0FF
F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
①

Here we can see data and address increasing : S[I] = i

①: When we reach FF we : Disable wen and send finish signal to start other FSM

TASK2A MODULE



① After the state start goes high and down, we init the signals "shuffle_mem_handler" and "mem_select" used to indicate what RAM we communicating with

② We can see i increasing i<=i+1 every time we finish laping the states

③ for j it's supposed to get Data from RAM but here we just use inputs

④ these two pulses indicates write enable for i and for j

⑤/⑥ Demonstrating swap operation ⑤ showing getting data from input putting it in tempi and then as output to go to j and the same thing for ⑥ for the j signal

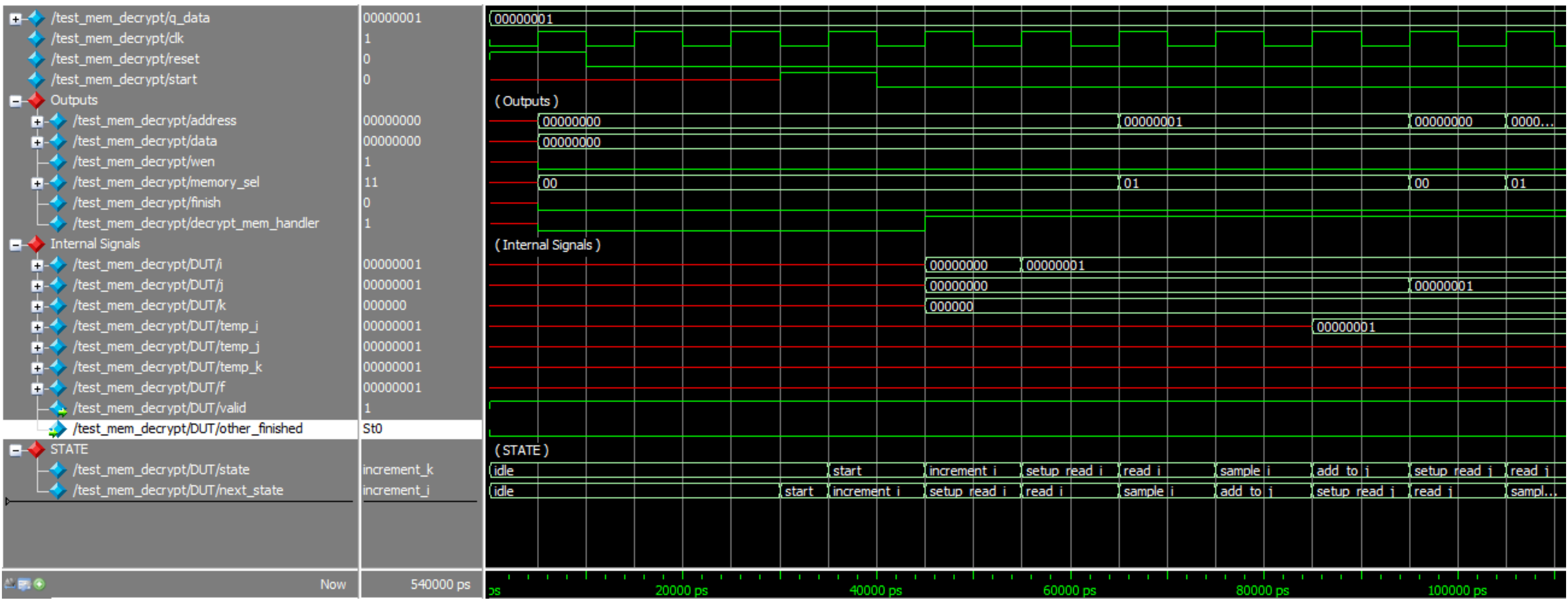
TASK2B MODULE

Memory_sel is used to select which memory block to communicate to:

- *1 is the working S RAM*
- *2 is the ROM*
- *3 is the decrypted output RAM*

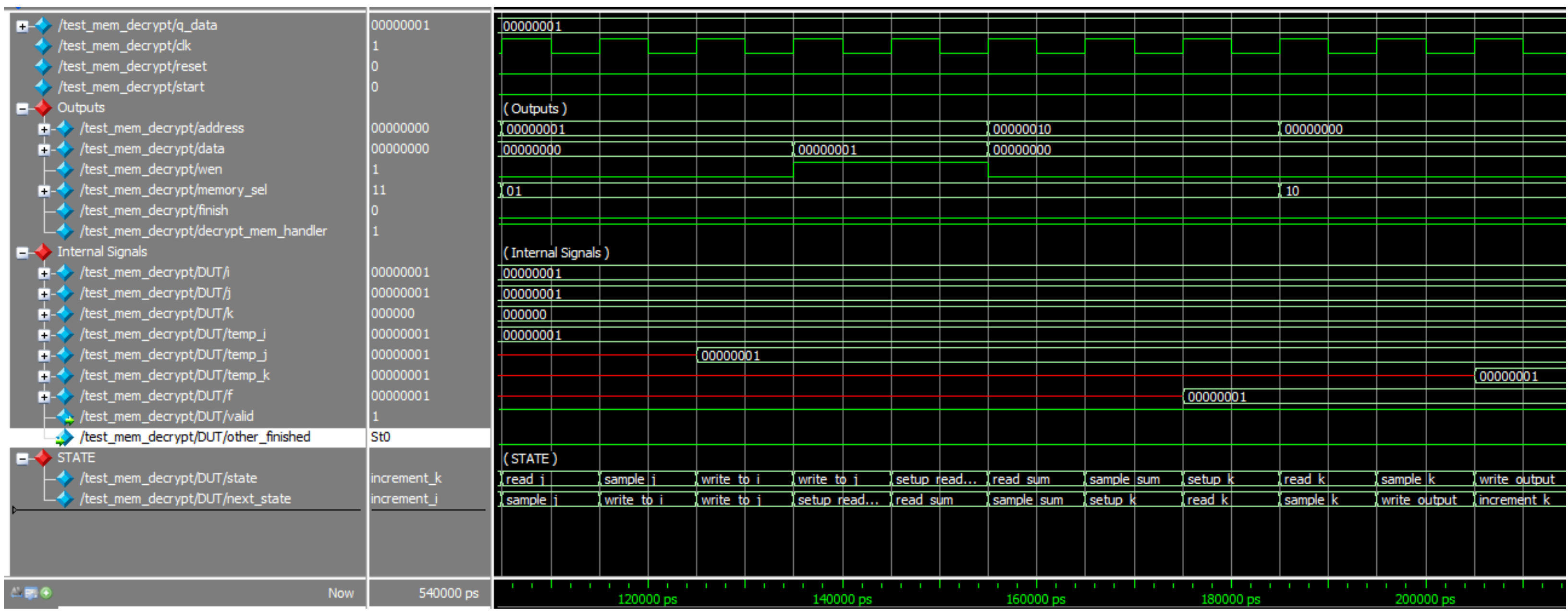
- Idle is the state where the FSM is checking where or not to start
- Outputs are one cycle behind the states
- Once the start signal is high, the state goes to start
  - Decrypt_mem_handler becomes high until the FSM goes to the finished state
- At the next clock edge, the state goes to increment i
  - i = i + 1, everything else is off
- At the next clock edges, the state goes to setup_read_i, read_i, and sample_i respectively
  - These states are used for retrieving s[i] from memory
  - For these states, address is assigned to i and memory_sel is assigned to 1
  - temp_i is assigned to q_data at the sample_i state and stores the read value s[i]
- At the next clock edge, the state goes to add_to_j
  - j = j + s[i] (temp_i), everything else is off
- At the next clock edges, the clock goes to setup_read_j, read_j, and sample_j respectively
  - These states are used for retrieving s[j] from memory

- For these states, address is assigned to j and memory_sel is assigned to 1
- temp_j is assigned to a_data at the sample_j state and stores the read value s[j]


/test_mem_decrypt/q_data 00000001
/test_mem_decrypt/clk 1
/test_mem_decrypt/reset 0
/test_mem_decrypt/start 0
Outputs
/test_mem_decrypt/address 00000000
/test_mem_decrypt/data 00000000
/test_mem_decrypt/wen 1
/test_mem_decrypt/memory_sel 11
/test_mem_decrypt/finish 0
/test_mem_decrypt/decrypt_mem_handler 1
Internal Signals
/test_mem_decrypt/DUT/i 00000001
/test_mem_decrypt/DUT/j 00000001
/test_mem_decrypt/DUT/k 000000
/test_mem_decrypt/DUT/temp_i 00000001
/test_mem_decrypt/DUT/temp_j 00000001
/test_mem_decrypt/DUT/temp_k 00000001
/test_mem_decrypt/DUT/f 00000001
/test_mem_decrypt/DUT/valid 1
/test_mem_decrypt/DUT/other_finished St0
STATE
/test_mem_decrypt/DUT/state increment_k
/test_mem_decrypt/DUT/next_state increment_i
Now 540000 ps
(Outputs)
(Internal Signals)
(STATE)
idle
start
increment_i
setup_read_i
read_i
sample_i
add_to_j
setup_read_j
read_j
sampl...
00000000
00000001
0000...
00
01
000000
20000 ps
40000 ps
60000 ps
80000 ps
100000 ps

- At the next clock edge, the state goes to write_to_i
  - s[i] = temp_j
  - Memory_sel = 1
  - The address signal becomes i, the data signal becomes temp_j, and the wen signal (write enable) goes high
- At the next clock edge, the state goes to write_to_j
  - s[j] = temp_j
  - Memory_sel = 1
  - The address signal becomes j, the data signal becomes temp_i, and the wen signal remains high
- At the next clock edges, the state goes to setup_read_sum, read_sum, and sample_sum respectively
  - These states are used for retrieving s[(s[i]+s[j])] from memory
  - The address is temp_i + temp_j (s[i] + s[j]) and memory_sel = 1 for these states
  - f is assigned to q_data at sample_sum and holds the read value s[(s[i]+s[j])]
- At the next clock edges, the state goes to setup_k, read_k, and sample_k respectively
  - These states are used for retrieving encrypted_output[k] from memory
  - The address is k and the memory_sel is 2 for these states
  - temp_k is assigned to q_data at sample_k and holds the read value encrypted_output[k]
- At the next clock edge, the state goes to write_output
  - Writes the decoded byte to decrypted_output[k]

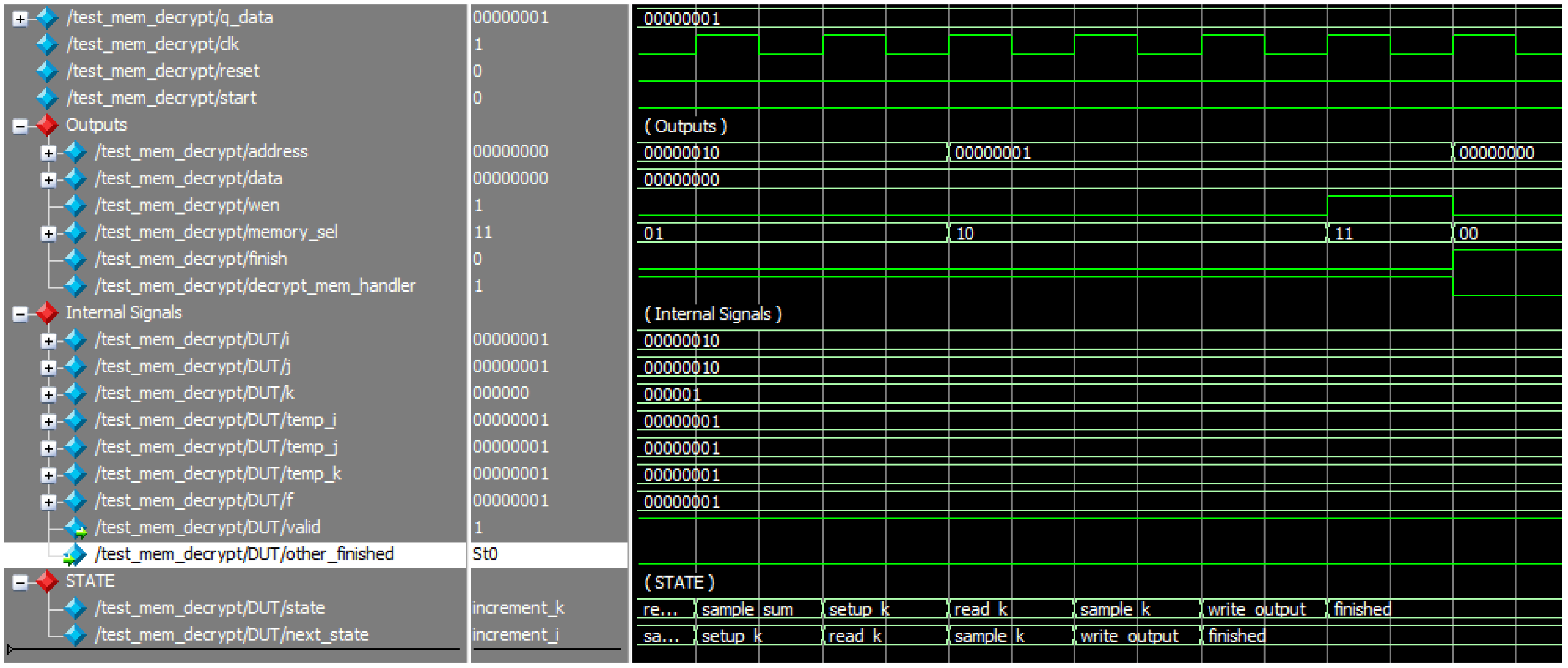
    - The address is k, the memory_sel is 3, the data is f XOR temp_k, and the wen is high for this state
- At the next clock edge, the state goes to finished if k has finished all its iterations. Otherwise, the state goes to increment_k
- At increment_k, the state goes to increment_i since (valid & ~others_finished) is high
    - k = k + 1



/test_mem_decrypt/q_data	00000001
/test_mem_decrypt/clk	1
/test_mem_decrypt/reset	0
/test_mem_decrypt/start	0
Outputs	
/test_mem_decrypt/address	00000000
/test_mem_decrypt/data	00000000
/test_mem_decrypt/wen	1
/test_mem_decrypt/memory_sel	11
/test_mem_decrypt/finish	0
/test_mem_decrypt/decrypt_mem_handler	1
Internal Signals	
/test_mem_decrypt/DUT/i	00000001
/test_mem_decrypt/DUT/j	00000001
/test_mem_decrypt/DUT/k	000000
/test_mem_decrypt/DUT/temp_i	00000001
/test_mem_decrypt/DUT/temp_j	00000001
/test_mem_decrypt/DUT/temp_k	00000001
/test_mem_decrypt/DUT/f	00000001
/test_mem_decrypt/DUT/valid	1
/test_mem_decrypt/DUT/other_finished	St0
STATE	
/test_mem_decrypt/DUT/state	increment_k
/test_mem_decrypt/DUT/next_state	increment_i
Now	540000 ps

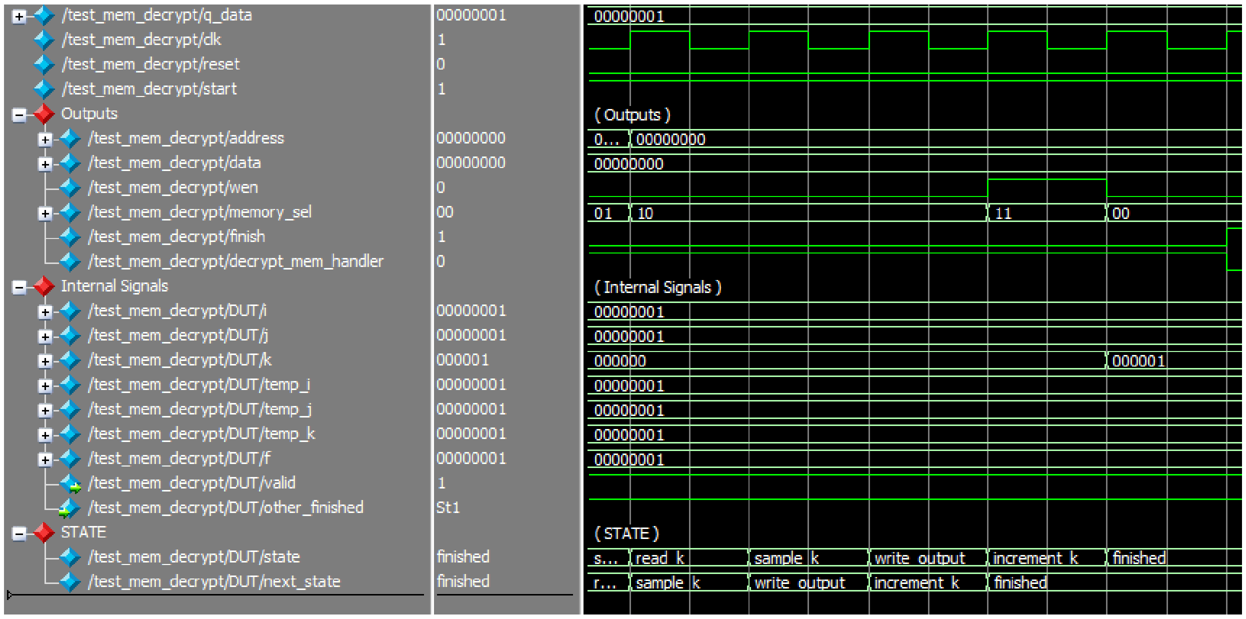
00000001
(Outputs)
00000001 00000010 00000000
00000000 00000001 00000000
01 10
(Internal Signals)
00000001
00000001
000000
00000001
00000001
00000001
00000001
(STATE)
read_j sample_j write_to_i write_to_j setup_read... read_sum sample_sum setup_k read_k sample_k write_output
sample_j write_to_i write_to_j setup_read... read_sum sample_sum setup_k read_k sample_k write_output increment_k
120000 ps
140000 ps
160000 ps
180000 ps
200000 ps

- When the state is write_output and we have traversed through all iterations (iterations = 1), the state becomes finished and stays at finished



/test_mem_decrypt/q_data	00000001
/test_mem_decrypt/clk	1
/test_mem_decrypt/reset	0
/test_mem_decrypt/start	0
Outputs	
/test_mem_decrypt/address	00000000
/test_mem_decrypt/data	00000000
/test_mem_decrypt/wen	1
/test_mem_decrypt/memory_sel	11
/test_mem_decrypt/finish	0
/test_mem_decrypt/decrypt_mem_handler	1
Internal Signals	
/test_mem_decrypt/DUT/i	00000001
/test_mem_decrypt/DUT/j	00000001
/test_mem_decrypt/DUT/k	000000
/test_mem_decrypt/DUT/temp_i	00000001
/test_mem_decrypt/DUT/temp_j	00000001
/test_mem_decrypt/DUT/temp_k	00000001
/test_mem_decrypt/DUT/f	00000001
/test_mem_decrypt/DUT/valid	1
/test_mem_decrypt/DUT/other_finished	St0
STATE	
/test_mem_decrypt/DUT/state	increment_k
/test_mem_decrypt/DUT/next_state	increment_i

- At increment_k, the state goes to finished since (valid & ~others_finished) is low

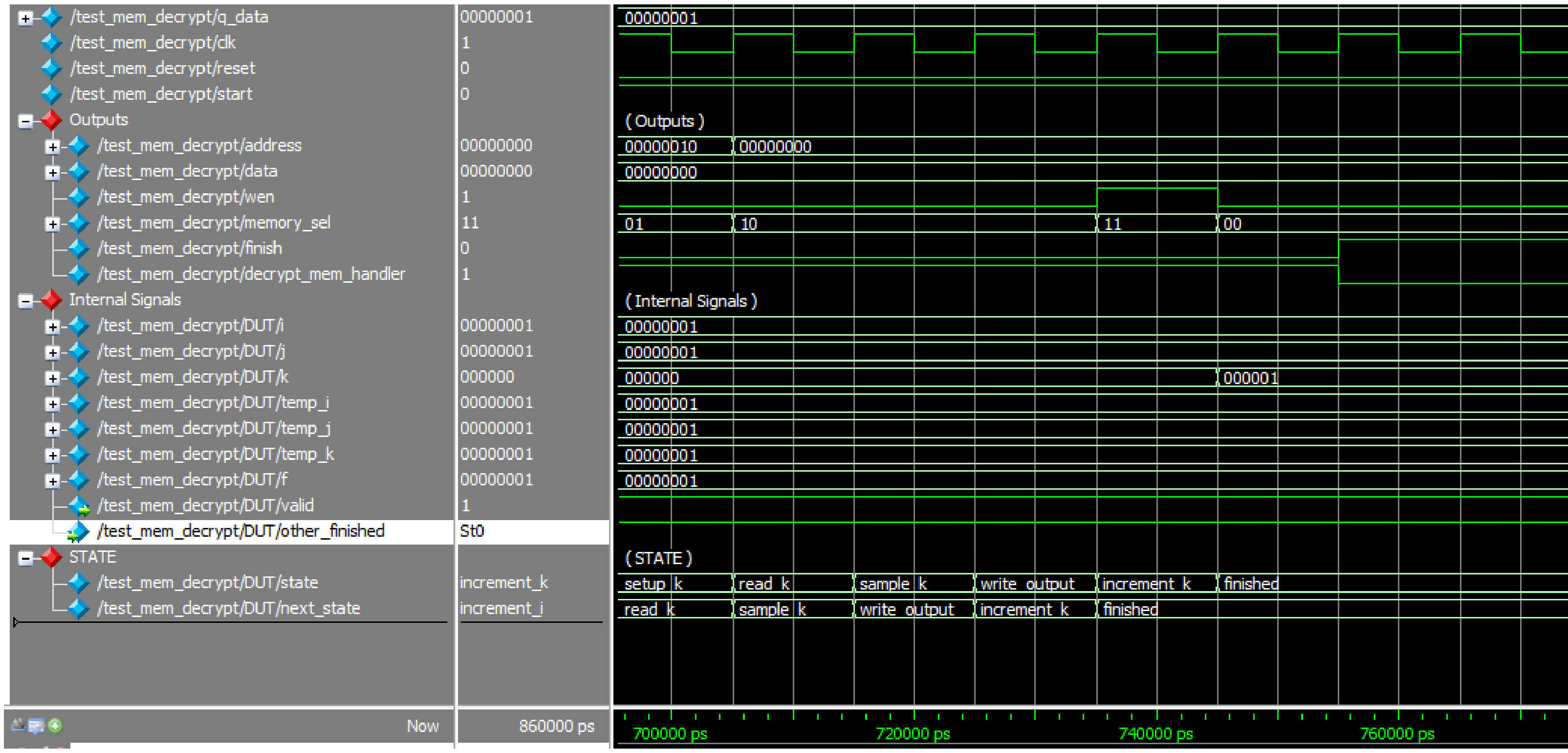


```
/test_mem_decrypt/q_data                    00000001
/test_mem_decrypt/clk                       1
/test_mem_decrypt/reset                     0
/test_mem_decrypt/start                     1
Outputs
  /test_mem_decrypt/address                 00000000
  /test_mem_decrypt/data                    00000000
  /test_mem_decrypt/wen                     0
  /test_mem_decrypt/memory_sel              00
  /test_mem_decrypt/finish                  1
  /test_mem_decrypt/decrypt_mem_handler     0
Internal Signals
  /test_mem_decrypt/DUT/i                   00000001
  /test_mem_decrypt/DUT/j                   00000001
  /test_mem_decrypt/DUT/k                   000001
  /test_mem_decrypt/DUT/temp_i              00000001
  /test_mem_decrypt/DUT/temp_j              00000001
  /test_mem_decrypt/DUT/temp_k              00000001
  /test_mem_decrypt/DUT/f                   00000001
  /test_mem_decrypt/DUT/valid               1
  /test_mem_decrypt/DUT/other_finished      St1
STATE
  /test_mem_decrypt/DUT/state               finished
  /test_mem_decrypt/DUT/next_state          finished
```

RC4_BRUTE_FORCE SIMULATION:

Port Assignments:

*The state transitions are described in the order that they occur*

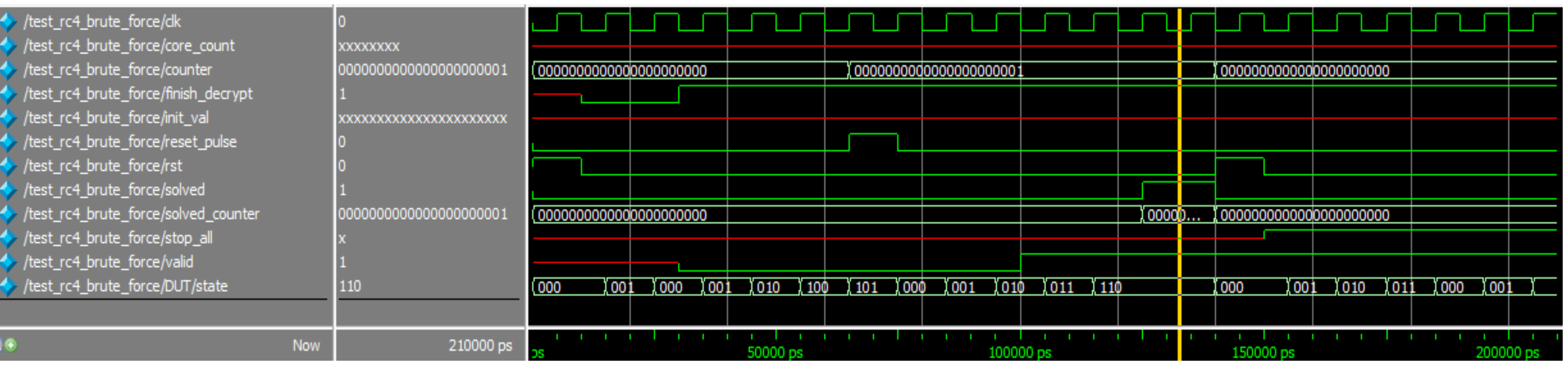


```
/test_mem_decrypt/q_data                    00000001
/test_mem_decrypt/clk                       1
/test_mem_decrypt/reset                     0
/test_mem_decrypt/start                     0
Outputs
  /test_mem_decrypt/address                 00000000
  /test_mem_decrypt/data                    00000000
  /test_mem_decrypt/wen                     1
  /test_mem_decrypt/memory_sel              11
  /test_mem_decrypt/finish                  0
  /test_mem_decrypt/decrypt_mem_handler     1
Internal Signals
  /test_mem_decrypt/DUT/i                   00000001
  /test_mem_decrypt/DUT/j                   00000001
  /test_mem_decrypt/DUT/k                   000000
  /test_mem_decrypt/DUT/temp_i              00000001
  /test_mem_decrypt/DUT/temp_j              00000001
  /test_mem_decrypt/DUT/temp_k              00000001
  /test_mem_decrypt/DUT/f                   00000001
  /test_mem_decrypt/DUT/valid               1
  /test_mem_decrypt/DUT/other_finished      St0
STATE
  /test_mem_decrypt/DUT/state               increment_k
  /test_mem_decrypt/DUT/next_state          increment_i
Now                                         860000 ps
```

All outputs for a specific state occur one cycle after the state arrives

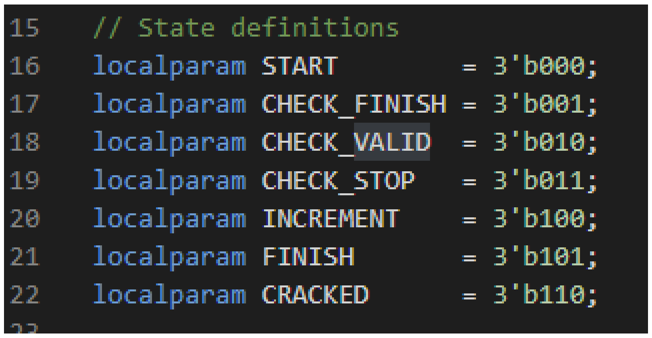
- When the rst signal is high, the state goes to START

    - The counter signal is initialized to init_val, which is set as 0
    - reset_pulse, solved, and solved_counter are all set to 0 as well
- At the next clock edge (rst is low), the state goes to CHECK_FINISH
- Since finish_decrypt is low, the state goes from CHECK_FINISH to START
- At the next clock edge, the state goes to CHECK_FINISH
- Since finish_decrypt is high this time, the state goes from CHECK_FINISH to CHECK_VALID
- Since the valid signal is low, the state goes from CHECK_VALID to INCREMENT
    - At increment, counter is assigned to counter + core_count, where core_count is 1
    - The reset_pulse signal is set to high
- At the next clock edge, the state goes to FINISH
- At the next clock edge, the state goes to START
    - The reset_pulse signal is grounded
- After a few clock_edges, the state goes back to CHECK_VALID
- Since the valid signal is high, the state goes to CHECK_STOP
- Since the stop_all signal is low, the state goes to CRACKED
    - solved_counter is assigned to be the counter signal
- The rst signal is toggled and the state goes back to CHECK_STOP after a few clock edges
- Since the stop_all signal is high, the state goes back to START

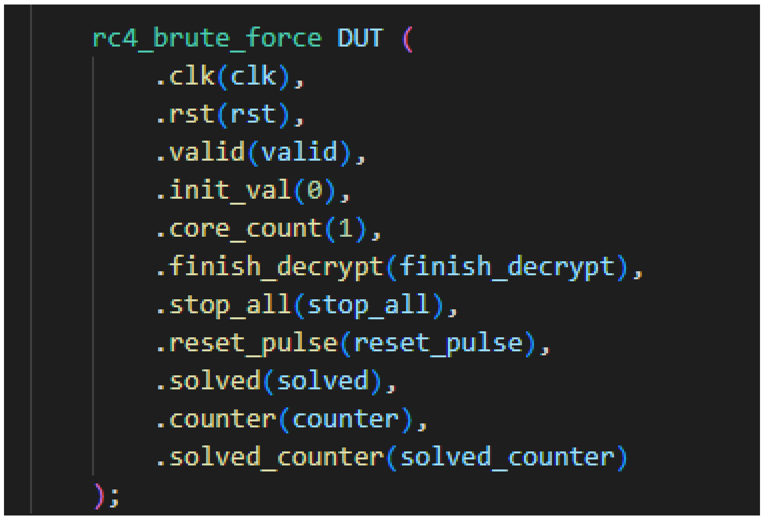


```
/test_rc4_brute_force/clk              0
/test_rc4_brute_force/core_count       xxxxxxxx
/test_rc4_brute_force/counter          0000000000000000000001
/test_rc4_brute_force/finish_decrypt   1
/test_rc4_brute_force/init_val         xxxxxxxxxxxxxxxxxxxxxx
/test_rc4_brute_force/reset_pulse      0
/test_rc4_brute_force/rst              0
/test_rc4_brute_force/solved           1
/test_rc4_brute_force/solved_counter   0000000000000000000001
/test_rc4_brute_force/stop_all         x
/test_rc4_brute_force/valid            1
/test_rc4_brute_force/DUT/state        110
Now                                    210000 ps
```

State Encodings:



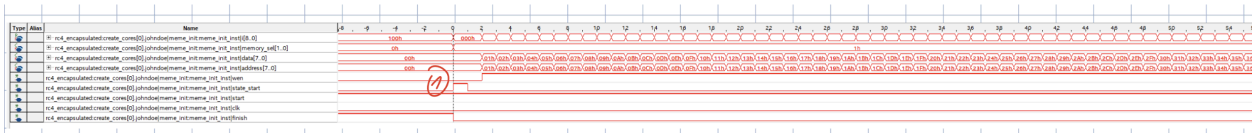
```
// State definitions
localparam START        = 3'b000;
localparam CHECK_FINISH = 3'b001;
localparam CHECK_VALID  = 3'b010;
localparam CHECK_STOP   = 3'b011;
localparam INCREMENT    = 3'b100;
localparam FINISH       = 3'b101;
localparam CRACKED      = 3'b110;
```



```
    rc4_brute_force DUT (
        .clk(clk),
        .rst(rst),
        .valid(valid),
        .init_val(0),
        .core_count(1),
        .finish_decrypt(finish_decrypt),
        .stop_all(stop_all),
        .reset_pulse(reset_pulse),
        .solved(solved),
        .counter(counter),
        .solved_counter(solved_counter)
    );
```

Signal Tap:

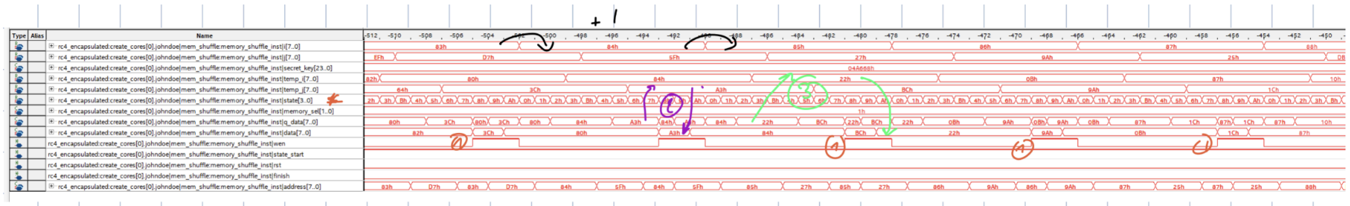
MEM_init (task1)



rc4_encapsulated:create_cores[0].johndoe|meme_init:meme_init_inst|i[8..0]
rc4_encapsulated:create_cores[0].johndoe|meme_init:meme_init_inst|memory_sel[1..0]
rc4_encapsulated:create_cores[0].johndoe|meme_init:meme_init_inst|data[7..0]
rc4_encapsulated:create_cores[0].johndoe|meme_init:meme_init_inst|address[7..0]
rc4_encapsulated:create_cores[0].johndoe|meme_init:meme_init_inst|wen
rc4_encapsulated:create_cores[0].johndoe|meme_init:meme_init_inst|state_start
rc4_encapsulated:create_cores[0].johndoe|meme_init:meme_init_inst|start
rc4_encapsulated:create_cores[0].johndoe|meme_init:meme_init_inst|rst
rc4_encapsulated:create_cores[0].johndoe|meme_init:meme_init_inst|clk
rc4_encapsulated:create_cores[0].johndoe|meme_init:meme_init_inst|finish
1

(1) When we receive state_start signal we start writing $s[i] = i$ from $i = 0 \Rightarrow 256$

MEM_shuffle (task2a)



+1
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|i[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|j[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|secret_key[23..0]
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|temp_i[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|temp_j[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|state[3..0]
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|memory_sel[1..0]
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|q_data[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|data[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|wen
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|state_start
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|rst
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|finish
rc4_encapsulated:create_cores[0].johndoe|mem_shuffle:memory_shuffle_inst|address[7..0]
1
2
3

(1) wen is high as we writing out data of the lata states

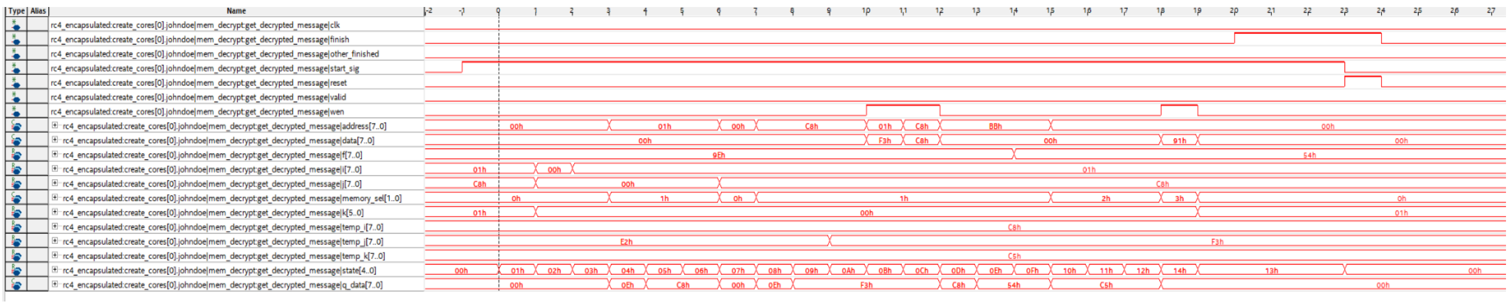
(2) We can see we got A5 from q_data put into temp_j

(3) Same as above (2) but for "i"

MEM_decrypt (task2b)

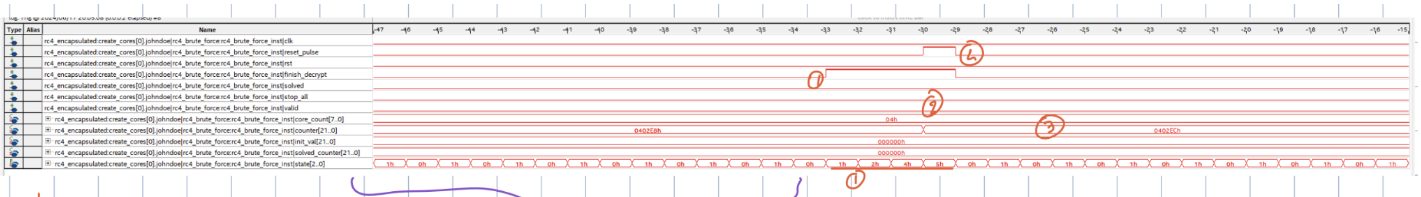
The states go in order as they were listed above and have the same outputs as described in the simulations

- When the state is a setup_read, read, or sample_state, the address signal takes in the address we want to read from
- When the state is a write state, the wen signal goes high and the data changes to the value we want to write to memory (address becomes where we want to write to)
- Memory_sel is 2 during the setup_k, read_k, and sample_k, states and 3 during the write_output state



rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|clk
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|finish
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|other_finished
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|start_sig
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|reset
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|valid
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|wen
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|address[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|data[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|f[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|i[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|j[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|memory_sel[1..0]
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|k[5..0]
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|temp_i[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|temp_j[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|temp_k[7..0]
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|state[4..0]
rc4_encapsulated:create_cores[0].johndoe|mem_decrypt:get_decrypted_message|q_data[7..0]

RC4_brute_force (task3)


Type
Alias
Name
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|clk
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|reset_pulse
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|rst
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|finish_decrypt
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|solved
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|stop_all
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|valid
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|core_count[7..0]
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|counter[21..0]
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|init_val[21..0]
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|solved_counter[21..0]
rc4_encapsulated:create_cores[0].johndoe|rc4_brute_force:rc4_brute_force_inst|state[2..0]
04h
0402E8h
0402ECh
000000h
000000h
1h 0h 1h 0h 1h 0h 1h 0h 1h 0h 1h 0h 1h 0h 1h 2h 4h 5h 0h 1h 0h 1h 0h 1h 0h 1h 0h 1h 0h 1h 0h 1h
(1)
(2)
(3)
(4)
(1)

states:

START : 0
Check_finish : 1
Check_valid : 2
Check_stop : 3
Increment : 4
Finish : 5
CRACKED : 6

toggeling btwn start and check finish cause we have not done decoding with current KEY yet

(1) Once we received signal we start decrypting

(2) Valid is low ⇔ key is incorrect ⇒ increment

(3) We have 4 cores, so next key for this core is 0402E8 + 4 = 0402EC

(4) send a system-wide reset to start over with new key