ENGINEERING INNOVATION DESIGN VI

AES ENCRYPTION COPROCESSOR

Name	Luigi Pizzolito
Student No.	1820212015
Class	13222101
Department	School of Information & Electronics
Email	luigi.pizzolito@hotmail.com
Date	April 16, 2024

Contents

1. RTL Design

This section documents the RTL design of the system, such as the submodules and their interconnects.

1.1 System Design

Below is the system block diagram. We have four submodules, their data interconnects (solid arrows) and control signal interconnects (dashed arrows).

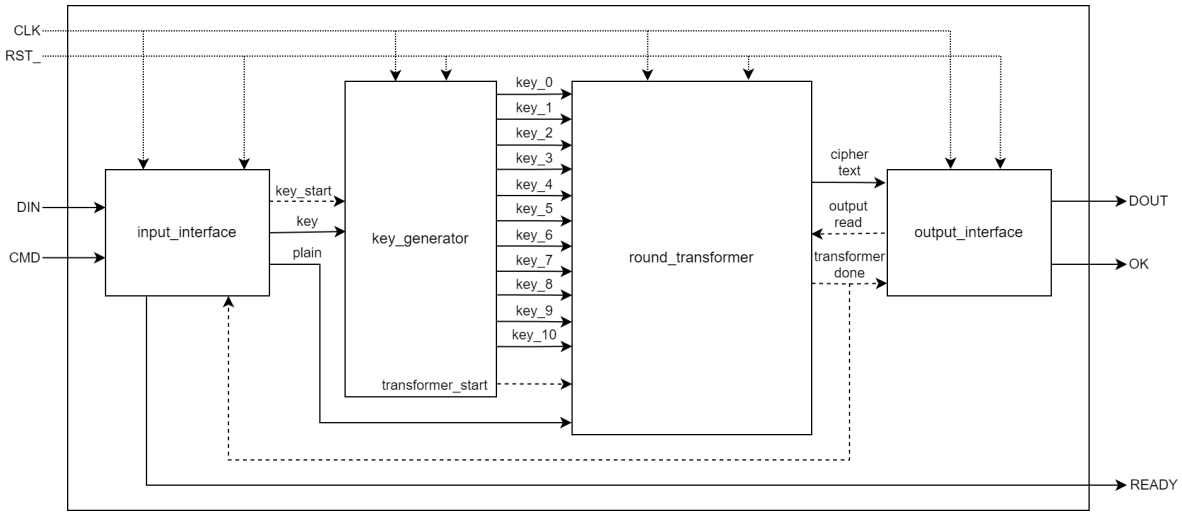


Figure 1.1: System block diagram

1.2 Module Interconnects

The `input_interface` and `output_interface` submodules handle the chip I/O (parallel-to-serial conversion) as well as the FSM used to process the input commands.

The `key_generator` generates the expanded round keys from the initial cipherkey, while the `round_transformer` performs the AES encryption and generates the ciphertext.

Note the design of the control signals, which are indicated by the dashed arrows. They signal when modules should start or when modules should reset (in addition to the `RST_` line). Clever design of these control signals allow pipelining of the next plaintext blocks to be encrypted, as later described in this report. Below is a detailed description of the function of each control signal:

- `key_start` signals that the key from `input_interface` is valid and `key_generator` may begin generating the round keys.

- `transformer_start` signals that the round keys have been generated and are valid and `round_transformer` may begin the encryption rounds.
- `transformer_done` signals that the encryption process is complete and ciphertext is valid. This signal accomplishes two tasks:
  - Signalling to the `output_interface` that ciphertext is valid and it can set `OK` and begin output.
  - Signalling to the `input_interface` that the encryption is complete and it may return from the start encryption state to the idle state (return to interface ready).
- `output_read` signals to the `round_transformer` that the ciphertext output was read, so it may reset its internal registers to be ready for the next plaintext encryption round.

2. RTL Implementation

This section describes the implementation details comprehensively and showcases the Verilog code used for the implementation.

2.1 Sub-Modules

Each module implementation from the system design is described in this section.

2.1.1 Input Interface

The input interface takes in a command (2-bit) determining the operation (idle, set plaintext, set key & start encryption). The data input is done in serial, one byte at a time.

This module outputs a 128-bit parallel plaintext and cipherkey to the `aes_engine` modules. As well as a ready output signal which goes high when the input interface is ready to accept commands.

The control signals for this module are as follows:

- `key_start` output signal to trigger the start of `aes_engine`.
- `transformer_done` input signal from `aes_engine` to determine when encryption is finished and this module can return from the start encrypt state to idle and reset it's plaintext.

In order to simplify the implementation; some Verilog tasks are used for resetting the internal state & some control signals are automatically generated from the FSM state using `assign` statements (such as the `key_start` signal when the current state is the start encryption command, and we have received 16 bytes of plaintext and cipherkey).

2.1.1.1 Finite-State Machine Design

This logic is implemented via a finite-state machine (FSM), whose state depends on both the input command, the counter for the serial-to-parallel receiver & the state of the control signals. Below is the state transition diagram for this FSM:

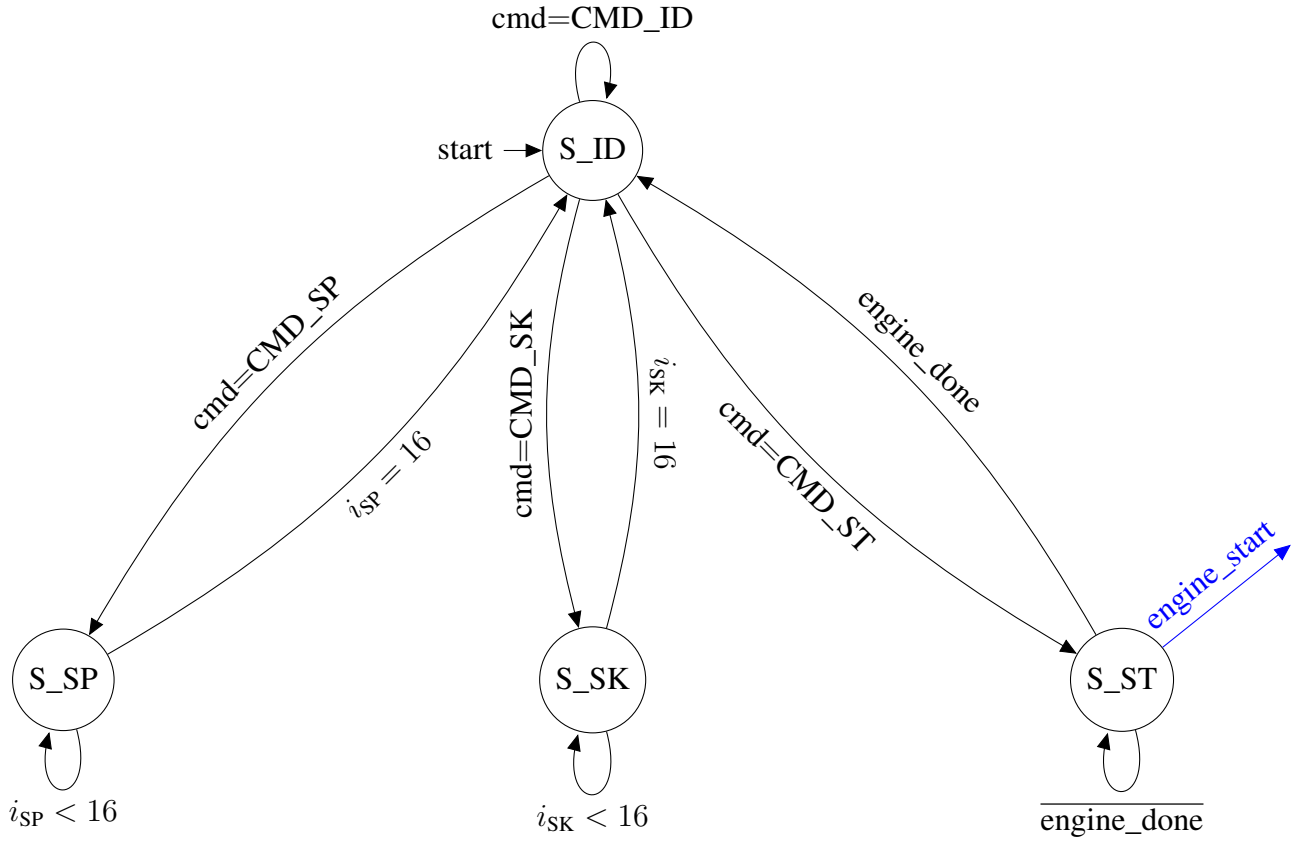


Figure 2.1: FSM for the input interface.

The initial state is the S_ID idle state. Then we can see that the input command will determine the next state. In the set plaintext (S_SP) and the set key (S_SK) states, until 16 bytes have been received through the serial data in, the state does not return to idle. In the start encryption state, we issue the engine_start signal an wait for the engine_done signal before returning to idle.

2.1.2 Round Keys Generator

The round keys generator takes in the 128-bit cipherkey from the input interface and the `key_start` control signal; as this is the first step of the AES engine encryption process.

This module outputs eleven 128-bit parallel round keys (for rounds 0 (pre-round) to round 11). As well as a `transformer_start` control signal to initiate the encryption round transformer once all the keys are ready.

The control signals for this module are as follows:

- `key_start` input signal to trigger this module to start generating the round encryption keys.
- `transformer_start` output signal to start the `engine_round_transformer` module when the round keys have finished generating.

The logic is implemented via an algorithm which computes the round keys operating on one word (32 bits) at a time.

For $i = 0...4R - 1$:

$$W_i = \begin{cases} K_i & \text{if } i \leq N \\ W_{i-1} \oplus \text{SubWord}(\text{RotWord}(W_{i-1})) \oplus Rcon_i & \text{if } i \geq N \text{ and } i \equiv 0 \pmod N \\ W_{i-1} \oplus \text{SubWord}(W_{i-1}) & \text{if } i \geq N \text{ and } N > 6 \text{ and } i \equiv 4 \pmod N \\ W_{i-N} \oplus W_{i-1} & \text{otherwise.} \end{cases}$$

Where:

$$\begin{aligned} N &= \text{the length of the key in 32-bit words ().} \\ K_0, K_1, ..., K_{N-1} &= \text{the 32-bit words of the original key.} \\ R &= \text{the number of round keys needed (11 for AES-128).} \\ W_0, W_1, ..., W_{4R-1} &= \text{the 32-bit words of the expanded key.} \\ \text{RotWord}([b_0\ b_1\ b_2\ b_3]) &= [b_1\ b_2\ b_3\ b_0] \\ \text{SubWord}([b_0\ b_1\ b_2\ b_3]) &= [\text{AES\_SBOX}(b_0)\ \text{AES\_SBOX}(b_1)\ \text{AES\_SBOX}(b_2)\ \text{AES\_SBOX}(b_3)] \\ Rcon_i &= [rc_i\ 00_{16}\ 00_{16}\ 00_{16}] \\ rc_i &= \begin{cases} 1 & \text{if } i = 1 \\ 2 \cdot rc_{i-1} & \text{if } i > 1 \text{ and } rc_{i-1} < 80_{16} \\ (2 \cdot rc_{i-1}) \oplus 11\text{B}_{16} & \text{if } i > 1 \text{ and } rc_{i-1} \geq 80_{16} \end{cases} \end{aligned} \tag{2.1}$$

This algorithm can be visualized in Figure 2.2.

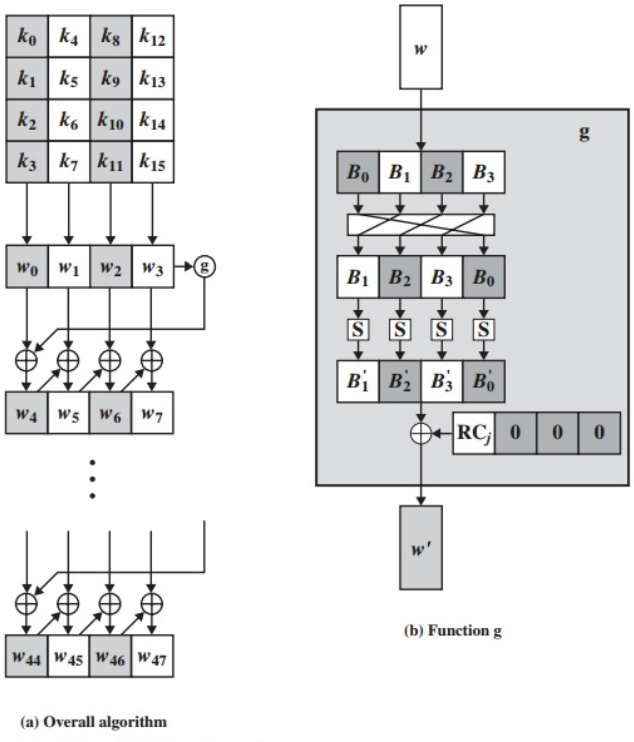


Figure 2.2: AES Key Schedule

This is done with the array `reg [31:0] w[43:0]` which holds all the 44 words for the eleven 128-bit round keys. The counter `i` is used to unwrap the for loop to perform one iteration per clock cycle, taking a total of 44 clock cycles to compute all the keys and start the `engine_round_transformer` module.

Additionally, there is a check to see if the requested key expansion is the same as one that has just been previously computed. In this case, the computation is skipped and `transformer_start` is issued directly as the round keys are already ready.

In order to simplify the implementation:

- Some Verilog tasks are used for resetting the internal state.
- `$write` and `$display` are used to print the round keys to the console for debugging.
- `round_constant`, `rotword`, `subword`, `aes_sbox` functions are implemented to perform each step of the algorithm on a word at a time.

2.1.3 Round Transformer

The round transformer takes in the 128-bit plaintext from the input interface, all eleven round keys from `engine_key_generator` and the `transformer_start` control signal; as this is the last step of the AES engine encryption process.

This module outputs 128-bit parallel ciphertext. As well as a `transformer_done` control signal to indicate that the engine is done to the `output_interface` module and auto-reset other modules.

The control signals for this module are as follows:

- `transformer_start` input signal to trigger this module to start the encryption rounds.
- `tranformer_done` output signal to indicate to `output_interface` that the ciphertext is ready and to reset other modules.
- `output_read` input signal from `output_interface` to indicate that the ciphertext output has been read and this module may now auto-reset.

The logic is implemented via the Rijndael algorithm for AES-128. Modified to to use the `aes_tbox` function which performs both `aes_sbox` and Gallois field multiplication for column mixing.

1. Pre-round:
    1. Round key addition - each byte of the state is added with a byte of the pre-round key using bitwise XOR.
2. Nine Rounds:
    1. `ShiftRow` - a transposition step where the last three rows of the state are shifted cyclically an increasing number of steps.

$$\begin{bmatrix} b_{1,1} & b_{2,1} & b_{3,2} & b_{4,1} \\ b_{1,2} & b_{2,2} & b_{3,2} & b_{4,2} \\ b_{1,3} & b_{2,3} & b_{3,3} & b_{4,3} \\ b_{1,4} & b_{2,4} & b_{3,4} & b_{4,4} \end{bmatrix} \Rightarrow \begin{bmatrix} b_{1,1} & b_{2,1} & b_{3,2} & b_{4,1} \\ b_{2,2} & b_{3,2} & b_{4,2} & b_{1,2} \\ b_{3,3} & b_{4,3} & b_{1,3} & b_{2,3} \\ b_{4,4} & b_{1,4} & b_{2,4} & b_{3,4} \end{bmatrix} \begin{matrix} \\ \rightarrow 0 \text{ left-shift} \\ \rightarrow 1 \text{ left-shift} \\ \rightarrow 2 \text{ left-shift} \\ \rightarrow 3 \text{ left-shift} \end{matrix} \tag{2.2}$$

    2. `MixCol` - a non-linear substitution step where each byte is replaced with another according to the `aes_sbox` lookup table. As well as a linear mixing operation which operates on the **columns** of the state, combining the four bytes in each column via matrix multiplication by a constant matrix:

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} \text{SubWord}(c_0) \\ \text{SubWord}(c_1) \\ \text{SubWord}(c_2) \\ \text{SubWord}(c_3) \end{bmatrix} \tag{2.3}$$

       Both of these steps are combined into a lookup table, the `aes_tbox`.
    3. Round key addition.
3. Final round (total of 11 rounds):

1. SubBytes - a non-linear substitution step where each byte is replaced with another according to the aes_sbox lookup table.
2. ShiftRow
3. Round key addition.

The current state stored in reg [127:0] state_block and is scrambled step-by-step by the algorithm above. The counter i is used to unwrap the for loop to perform one encryption round per clock cycle, taking a total of 11 clock cycles to compute the ciphertext.

In order to simplify the implementation:

- Some Verilog tasks are used for resetting the internal state.
- $write and $display are used to print the current state to the console for debugging (see task print_matrix).
- SubBytes, ShiftRow, rotword, MixCol, aes_sbox, aes_tbox functions are implemented to perform each step of the algorithm on a word at a time.

2.1.4 Output Interface

The output interface takes in the 128-bit ciphertext from the engine_round_transformer.

This module outputs the ciphertext in serial via an 8-bit data_out, as well as a data_ok signal.

The control signals for this module are as follows:

- transformer_done input signal from aes_round_transformer to determine that the encryption round transformer is done and the ciphertext input is valid.
- data_ok output signal goes high when the ciphertext is valid and it's 128-bits are being outputted through the 8-bit serial output, one byte per clock cycle.
- output_read output signal goes high when all 8 bytes of the 128-bit ciphertext have been outputted through the serial interface. Used to automatically reset the aes_round_transformer.

The logic is implemented with a counter i to convert the parallel ciphertext input to a serial output.

In order to simplify the implementation; some Verilog tasks are used for resetting the internal state.

2.2 Top Module

The top module is the aes_engine module, which includes all of the previous sub-modules and connects their control signals accordingly.

2.2.1 AES Engine

These are the steps to approach the implementation of the top module.
First, we define the inputs and outputs of the top module:

1. clk and rst_ for the input clock and active-low reset.
2. din 8-bit serial data input for setting plaintext and key.
3. cmd 2-bit command input for idle, set plaintext, set key & start encryption commands.
4. dout 8-bit serial data output for reading ciphertext.
5. data_ok output signal to indicate the dout is valid and outputting the currently requested ciphertext.

Second, we include each submodule:

1. input_interface to parse the input commands and control the rest of the module.
2. engine_key_generator to expand the input key using the AES key schedule to generate one 128-bit key for each encryption round.
3. engine_round_transformer to perform the encryption rounds using Rijndael.
4. output_interface to provide the output ciphertext in serial format.

Lastly, we interconnect the sub-modules with all the required control signals & interconnects as per the design:

- Interconnects:
  - plain
    brings the plaintext from input_interface → engine_round_transformer.
  - key
    brings the original key from input_interface → engine_key_generator.
  - roundX_key
    brings each of the eleven round keys from
    engine_key_generator → engine_round_transformer.
  - cipher
    brings the final ciphertext from
    engine_round_transformer → output_interface.
- Control Signals:

- `key_start`
  this signal from `input_interface` tells `engine_key_generator` to begin generating the expanded round keys.
- `transformer_start`
  this signal from `engine_key_generator` tells `engine_round_transformer` to begin the encryption rounds.
- `transformer_done`
  this signal from `engine_round_transformer` tells other sub-modules that the encryption has finished and they may auto-reset their plantexts.
- `output_read`
  this signal from `output_interface` tells `engine_round_transformer` that the output has been read and it may auto-reset.

3. RTL Simulation

This section shows the waveforms and console output of the simulation of each module and the top module with the use of testbench.

3.1 Top Module: AES Engine

3.1.1 Waveforms

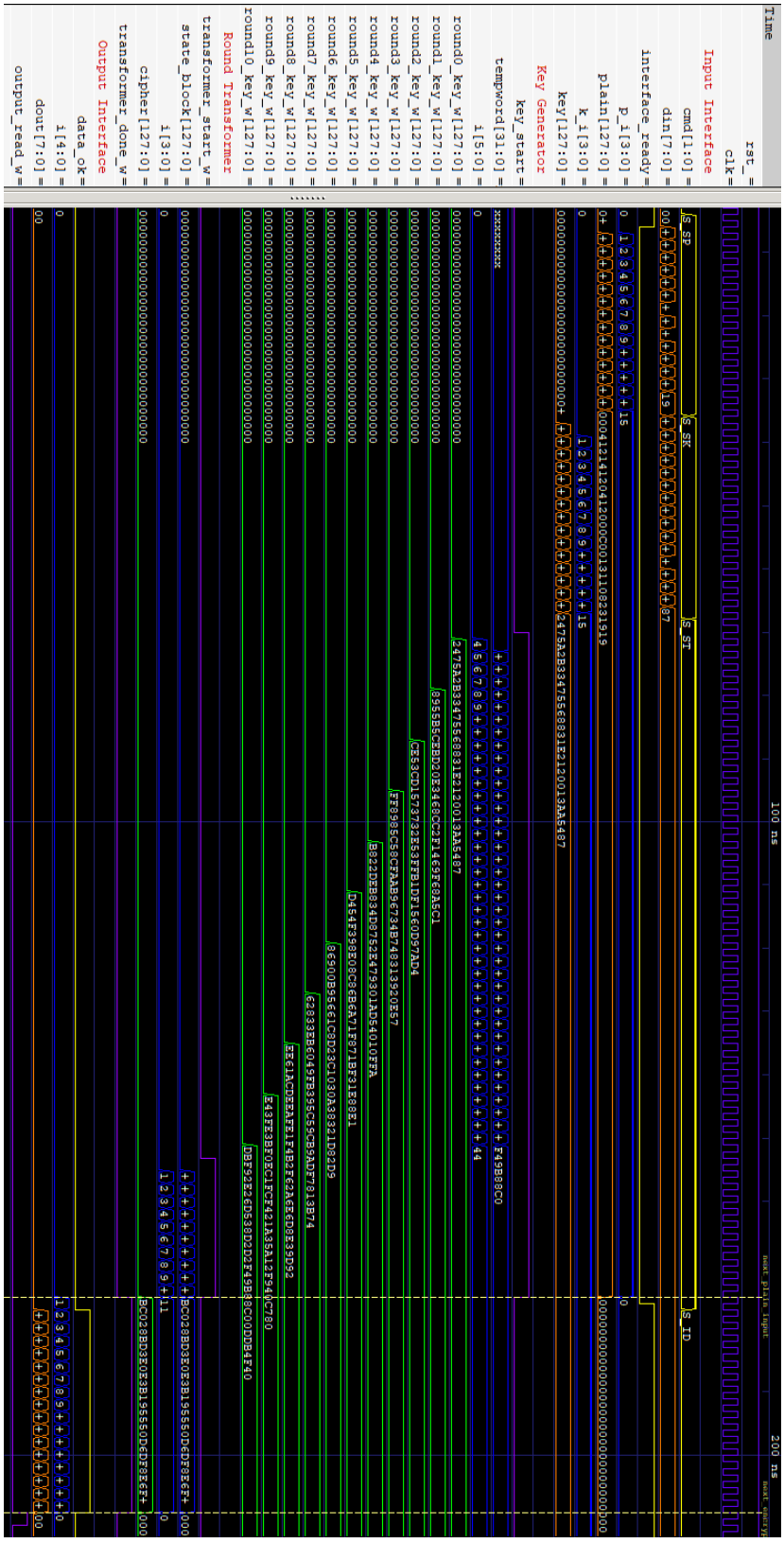
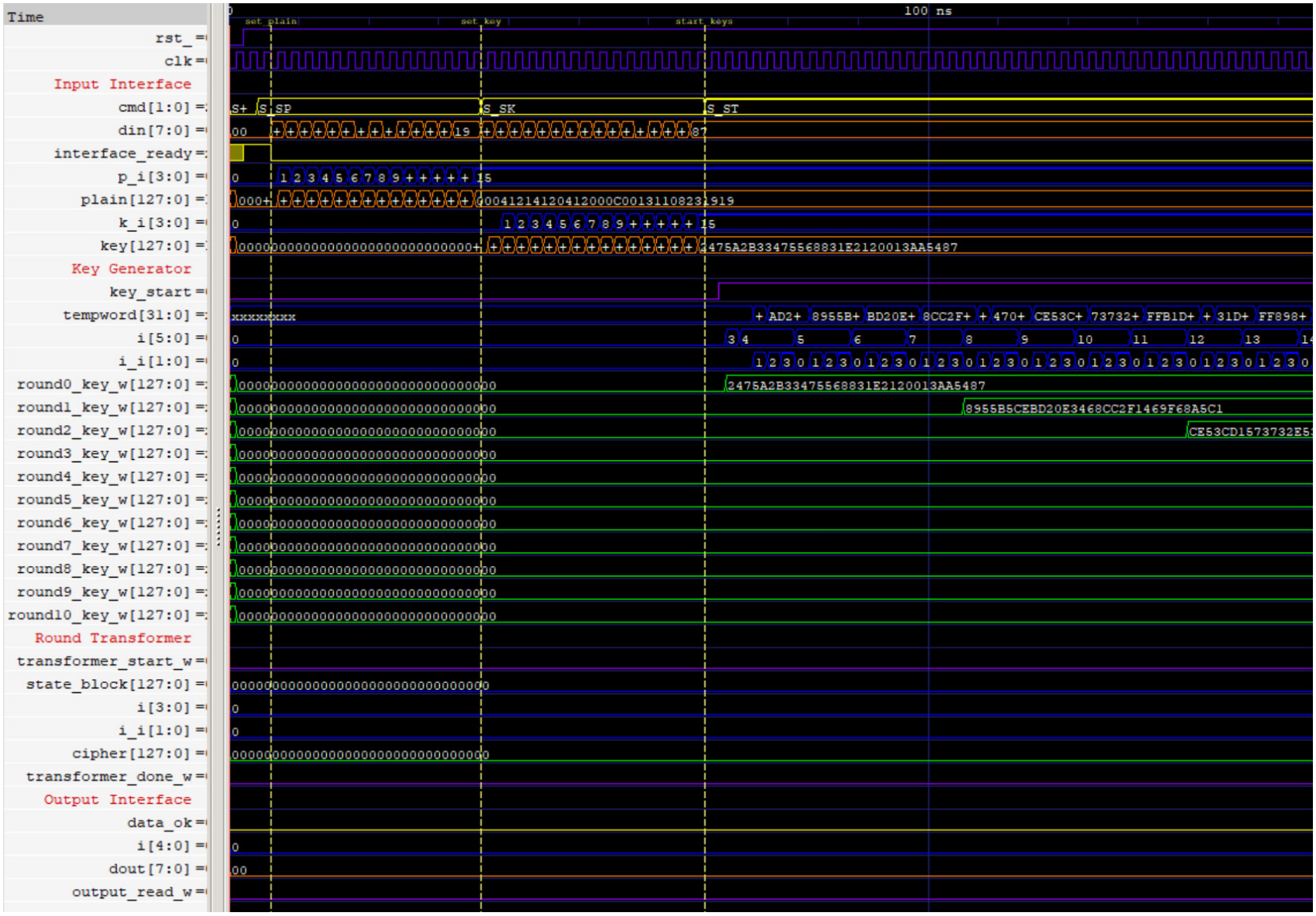
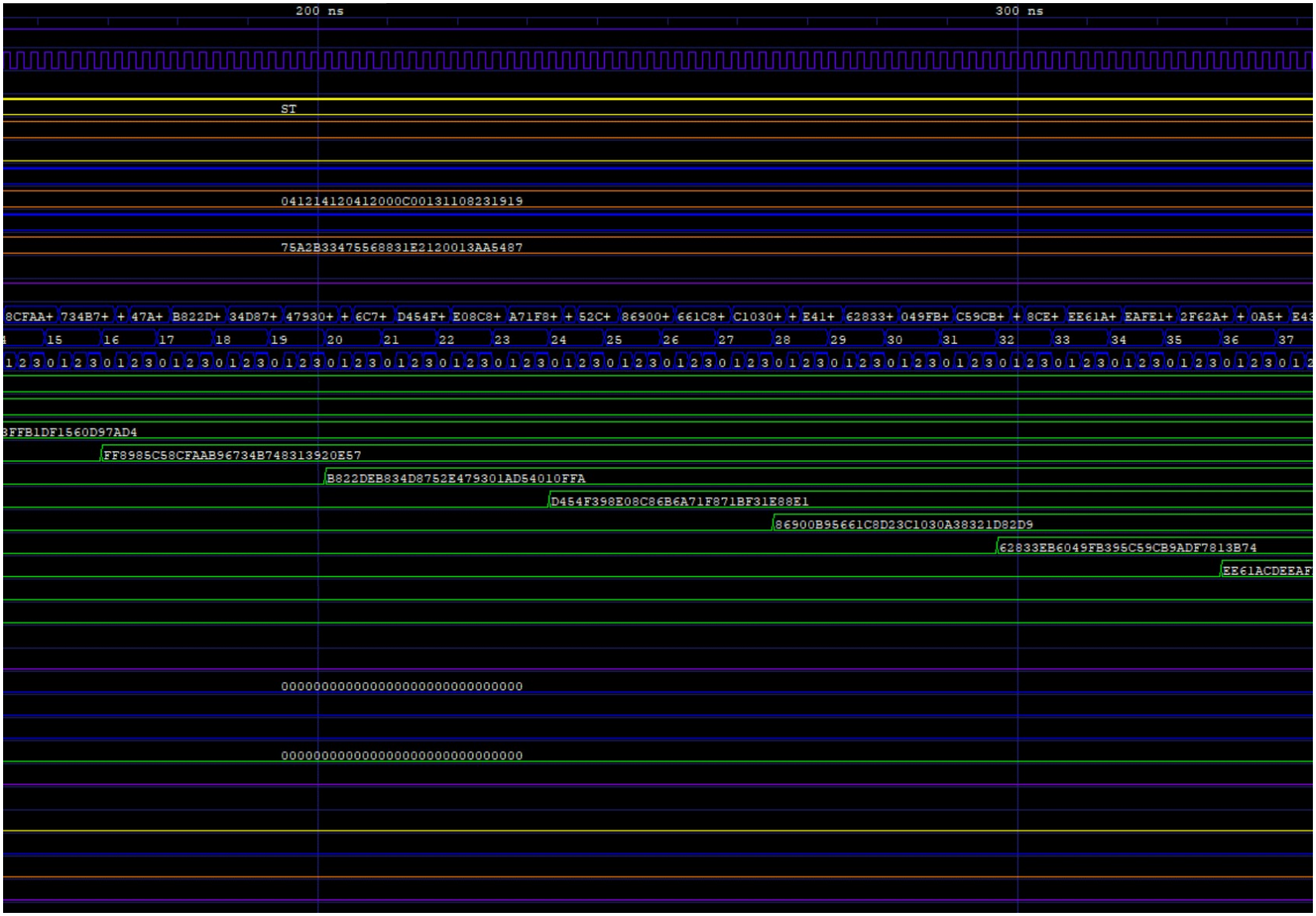


Figure 3.1: Waveforms for aes_engine (Pre-Synthesis)


Time
rst_
clk
Input Interface
cmd[1:0]
din[7:0]
interface_ready
p_i[3:0]
plain[127:0]
k_i[3:0]
key[127:0]
Key Generator
key_start
tempword[31:0]
i[5:0]
i_i[1:0]
round0_key_w[127:0]
round1_key_w[127:0]
round2_key_w[127:0]
round3_key_w[127:0]
round4_key_w[127:0]
round5_key_w[127:0]
round6_key_w[127:0]
round7_key_w[127:0]
round8_key_w[127:0]
round9_key_w[127:0]
round10_key_w[127:0]
Round Transformer
transformer_start_w
state_block[127:0]
i[3:0]
i_i[1:0]
cipher[127:0]
transformer_done_w
Output Interface
data_ok
i[4:0]
dout[7:0]
output_read_w
100 ns
set_plain
set_key
start_keys
S_SP
S_SK
S_ST
2475A2B33475568831E2120013AA5487
8955B5CEBD20E3468CC2F1469F68A5C1


200 ns
300 ns
ST
0412141204120000C00131108231919
75A2B33475568831E2120013AA5487
8CFAA+ 734B7+ + 47A+ B822D+ 34D87+ 47930+ + 6C7+ D454F+ E08C8+ A71F8+ + 52C+ 86900+ 661C8+ C1030+ + E41+ 62833+ 049FB+ C59CB+ + 8CE+ EE61A+ EAFE1+ 2F62A+ + 0A5+ E43
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
1 2 3 0 1 2 3 0 1 2 3 0

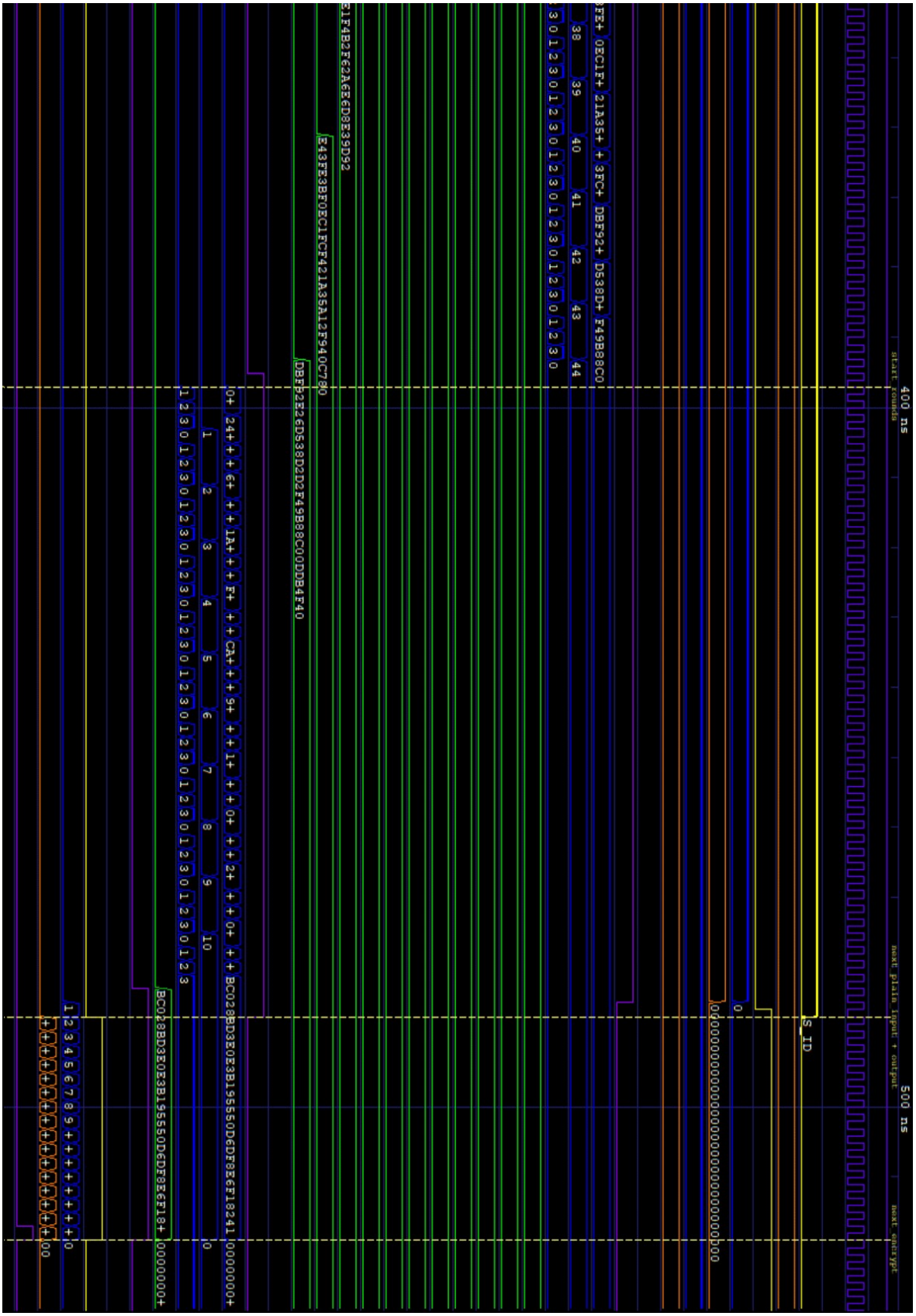


Figure 3.3: Waveforms for aes_engine (Synthesis-Ready)

Above is the complete waveforms of the entire top module. We can see how the sub-loop counters are added in the synthesis-ready version and their effect on the number of clock cycles needed to complete encryption of one plaintext block.

Notice the yellow X-axis markers that were added for clarity. They indicate; when the setting of plaintext begins, when the setting of cipherkey begins, when the round key generation begins, when the encryption rounds themselves begin, when the output begins (and the next plaintext can be set), and when the next encryption cycle may begin.

3.1.2 Console Output

In the console output, we can see that all the information is printed using block formatting; all 11 round keys, the state of the block during each encryption round, and the final ciphertext output.

```
----------------
Generating round keys for key
↪  2475a2b33475568831e2120013aa5487
Pre-Round Key:
24 34 31 13
75 75 e2 aa
a2 56 12 54
b3 88 00 87
Round 1 Key:
89 bd 8c 9f
55 20 c2 68
b5 e3 f1 a5
ce 46 46 c1
Round 2 Key:
ce 73 ff 60
53 73 b1 d9
cd 2e df 7a
15 53 15 d4
Round 3 Key:
ff 8c 73 13
89 fa 4b 92
85 ab 74 0e
c5 96 83 57
Round 4 Key:
b8 34 47 54
22 d8 93 01
de 75 01 0f
b8 2e ad fa
Round 5 Key:
d4 e0 a7 f3
54 8c 1f 1e
f3 86 87 88
98 b6 1b e1
Round 6 Key:
86 66 c1 32
90 1c 03 1d
0b 8d 0a 82
95 23 38 d9
Round 7 Key:
62 04 c5 f7
83 9f 9c 81
3e b3 b9 3b
b6 95 ad 74
Round 8 Key:
ee ea 2f d8
61 fe 62 e3
ac 1f a6 9d
de 4b e6 92
Round 9 Key:
e4 0e 21 f9
3f c1 a3 40
e3 fc 5a c7
bf f4 12 80
Round 10 Key:
db d5 f4 0d
f9 38 9b db
2e d2 88 4f
26 d2 c0 40
----------------
----------------
Plaintext:
00 12 0c 08
04 04 00 23
12 12 13 19
14 00 11 19
Pre-Round State:
24 26 3d 1b
71 71 e2 89
b0 44 01 4d
a7 88 11 9e
Round 1 State:
----------------
6c 44 13 bd
b1 9e 46 35
c5 b5 f3 02
5d 87 fc 8c
Round 2 State:
----------------
1a 90 15 b2
66 09 1d fc
20 55 5a b2
```

```
2b cb 8c 3c
Round 3 State:
----------------
f6 7d a2 b0
1b 61 b4 b8
67 09 c9 45
4a 5c 51 09
Round 4 State:
----------------
ca e5 48 bb
d8 42 af 71
d1 ba 98 2d
4e 60 9e df
Round 5 State:
----------------
90 35 13 60
2c fb 82 3a
9e fc 61 ed
49 39 cb 47
Round 6 State:
----------------
18 0a b9 b5
64 68 6a fb
5a ef d7 79
8e b2 10 4d
Round 7 State:
----------------
01 63 f1 96
55 24 3a 62
f4 8a de 4d
cc ba 88 03
Round 8 State:
----------------
2a 34 d8 46
2d 6b a2 d6
51 64 cf 5a
87 a8 f8 28
Round 9 State:
----------------
0a d9 f1 3c
95 63 9f 35
2a 80 29 00
16 76 09 77
Round 10 State / Ciphertext:
----------------
bc e0 55 e6
02 e3 0d f1
8b b1 6d 82
d3 95 f8 41
----------------
Recieved ciphertext from
↪  output_interface:
bc e0 55 e6
02 e3 0d f1
8b b1 6d 82
d3 95 f8 41
```

3.2 Sub-Modules

3.2.1 Input Interface

3.2.1.1 Waveforms

In the input interface waveforms, we can see; the command input, the current state of the FSM, the counters for the serial reception of the cipherkey and plaintext, as well as the input/output lines and control signals.

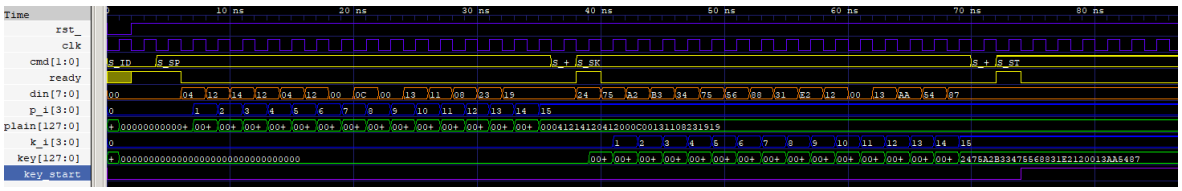


Figure 3.4: Waveforms for input_interface

There is no console output for this submodule.

3.2.2 Round Keys Generator

3.2.2.1 Waveforms

In the round keys generator waveforms, we can see; the input control signal to start generating the keys, the loop counter and tempword used to generate each round key sequentially, as well as the output round transformer start control signal.

Figure 3.5: Waveforms for engine_key_generator

3.2.2.2 Console Output

The console output for this module shows us the pre-round key, as well as each of the generated round keys.

```
Generating round keys for key
↪  2475a2b33475568831e2120013aa5487
Pre-Round Key:
24 34 31 13
75 75 e2 aa
a2 56 12 54
b3 88 00 87
Round 1 Key:
89 bd 8c 9f
55 20 c2 68
b5 e3 f1 a5
ce 46 46 c1
Round 2 Key:
ce 73 ff 60
53 73 b1 d9
cd 2e df 7a
15 53 15 d4
Round 3 Key:
ff 8c 73 13
89 fa 4b 92
85 ab 74 0e
c5 96 83 57
Round 4 Key:
b8 34 47 54
22 d8 93 01
de 75 01 0f
b8 2e ad fa
Round 5 Key:
d4 e0 a7 f3
54 8c 1f 1e
f3 86 87 88
98 b6 1b e1
Round 6 Key:
86 66 c1 32
90 1c 03 1d
0b 8d 0a 82
95 23 38 d9
Round 7 Key:
62 04 c5 f7
83 9f 9c 81
3e b3 b9 3b
b6 95 ad 74
Round 8 Key:
ee ea 2f d8
61 fe 62 e3
ac 1f a6 9d
de 4b e6 92
Round 9 Key:
e4 0e 21 f9
3f c1 a3 40
e3 fc 5a c7
bf f4 12 80
Round 10 Key:
db d5 f4 0d
f9 38 9b db
2e d2 88 4f
26 d2 c0 40
----------------
```

3.2.3 Round Transformer

3.2.3.1 Waveforms

In the round transformer waveforms; we can see the input transformer start control signal, the loop counter for each encryption round, the current block state, the ciphertext output and the transformer done output control signal.

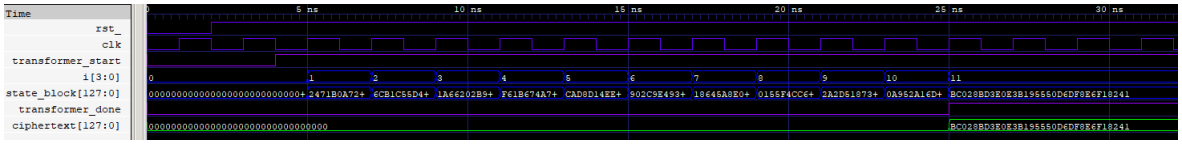


Figure 3.6: Waveforms for engine_round_transformer

3.2.3.2 Console Output

The console output shows us the plaintext and the state block after each encryption round.

```
Plaintext:
00 12 0c 08
04 04 00 23
12 12 13 19
14 00 11 19
Pre-Round State:
24 26 3d 1b
71 71 e2 89
b0 44 01 4d
a7 88 11 9e
Round 1 State:
6c 44 13 bd
b1 9e 46 35
c5 b5 f3 02
5d 87 fc 8c
Round 2 State:
1a 90 15 b2
66 09 1d fc
20 55 5a b2
2b cb 8c 3c
Round 3 State:
f6 7d a2 b0
1b 61 b4 b8
67 09 c9 45
4a 5c 51 09
Round 4 State:
ca e5 48 bb
d8 42 af 71
d1 ba 98 2d
4e 60 9e df
Round 5 State:
90 35 13 60
2c fb 82 3a
9e fc 61 ed
49 39 cb 47
Round 6 State:
18 0a b9 b5
64 68 6a fb
5a ef d7 79
8e b2 10 4d
Round 7 State:
01 63 f1 96
55 24 3a 62
f4 8a de 4d
cc ba 88 03
Round 8 State:
2a 34 d8 46
2d 6b a2 d6
51 64 cf 5a
87 a8 f8 28
Round 9 State:
0a d9 f1 3c
95 63 9f 35
2a 80 29 00
16 76 09 77
Round 10 State / Ciphertext:
bc e0 55 e6
02 e3 0d f1
8b b1 6d 82
d3 95 f8 41
```

3.2.4 Output Interface

3.2.4.1 Waveforms

In the output interface waveforms, we can see; the input transformer done control signal, the loop counter used to count the number of ciphertext output bits transmitted by serial, the output hold register that temporarily holds the data as it is shifted out, as well as the output control signals for data ok and output read done.

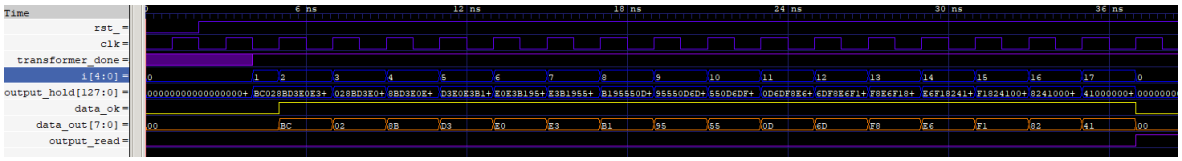


Figure 3.7: Waveforms for output_interface

3.2.4.2 Console Output

The console output for this module simply shows us the ciphertext received and outputted.

```
Recieved ciphertext from output_interface:
bc e0 55 e6
02 e3 0d f1
8b b1 6d 82
d3 95 f8 41
```

4. Debugging & Optimisation

Several notable points shall be mentioned of issues found during debugging and implementation considerations for optimisation.

4.1 Debugging

While debugging the system, the use of `$display` statement was very useful to see the variables quickly. However, both the round key generator and round transformer have many chained complex functions which caused difficulty in tracking down bugs.

4.1.1 Round Keys Generator Algorithm

Initially, an algorithm which computed each round key at once was implemented to generate the round keys. However, this did not port very well to Verilog and the bugs were too hard to iron out. The round keys generator was then re-implemented using the algorithm which sequentially computes the round keys, one word at a time (iterating over 44 words total). This was easier to implement as there were less multiple assignments of the same variable within one block execution and therefore could more easily be split up into separate steps to run as synthesisable Verilog code.

Some difficulties were encountered here, because in Verilog you are not allowed to use variables to address which bits of a register you are accessing. The indices must be constants at compile time. This means that for example; when printing each round key, we must only use a variable in the array index, not the byte access index:

```
98          $write("%02X %02X %02X %02X\n", w[((i/4)*4)][31:24],
            ↪  w[((i/4)*4)+1][31:24], w[((i/4)*4)+2][31:24],
            ↪  w[((i/4)*4)+3][31:24]);
99          $write("%02X %02X %02X %02X\n", w[((i/4)*4)][23:16],
            ↪  w[((i/4)*4)+1][23:16], w[((i/4)*4)+2][23:16],
            ↪  w[((i/4)*4)+3][23:16]);
100         $write("%02X %02X %02X %02X\n", w[((i/4)*4)][15:8] ,
            ↪  w[((i/4)*4)+1][15:8] , w[((i/4)*4)+2][15:8] , w[((i/4)*4)+3][15:8]
            ↪  );
101         $write("%02X %02X %02X %02X\n", w[((i/4)*4)][7:0]   , w[((i/4)*4)+1][7:0]
            ↪  , w[((i/4)*4)+2][7:0]   , w[((i/4)*4)+3][7:0]   );
```

Additionally, we must use the append operator in order to assign these separate words to the 128-bit output key registers (4 words to one register):

```
103         round_keys[i/4] = {w[i-3], w[i-2], w[i-1], w[i]};
```

4.1.2 Round Transformer Algorithm

The round transformer submodule calls many different sub-functions, including; `SubBytes`, `ShiftRow`, `rotword`, `MixCol`, `aes_sbox` and `aes_tbox`. Which made this the hardest module to debug.

A great bug found was when using the `aes_tbox` function. This function returns the result of `aes_tbox` multiplied by 1, 2 or 3 over GF^8. However the original implementation doesn't take the multiplier as a parameter and just returns all three 8-bit values as a 24-bit value. This is very confusing and not very useful at all. I re-implemented this function to take a multiplier parameter and return a single byte:

```
846   // where, for AES TBOX
847   // 03 = [23:16]
848   // 02 = [15:8]
849   // 01 = [7:0]
850   case (mult)
851     3: aes_tbox = full_tbox[23:16];
852     2: aes_tbox = full_tbox[15:8];
853     1: aes_tbox = full_tbox[7:0];
854   endcase
855   end
856 endfunction
```

This simplified the implementation of `MixCol` greatly, as now we can call `aes_tbox` as follows:

```
264       // Column 1
265       {
266         /*CM Row 1*/
267         {aes_tbox(input_block[127:120], 2) ^ aes_tbox(input_block[119:112], 3) ^
          ↪  aes_tbox(input_block[111:104], 1) ^ aes_tbox(input_block[103:96],
          ↪  1)},
268         /*CM Row 2*/
269         {aes_tbox(input_block[127:120], 1) ^ aes_tbox(input_block[119:112], 2) ^
          ↪  aes_tbox(input_block[111:104], 3) ^ aes_tbox(input_block[103:96],
          ↪  1)},
270         /*CM Row3*/
271         {aes_tbox(input_block[127:120], 1) ^ aes_tbox(input_block[119:112], 1) ^
          ↪  aes_tbox(input_block[111:104], 2) ^ aes_tbox(input_block[103:96],
          ↪  3)},
272         /*CM Row4*/
273         {aes_tbox(input_block[127:120], 3) ^ aes_tbox(input_block[119:112], 1) ^
          ↪  aes_tbox(input_block[111:104], 1) ^ aes_tbox(input_block[103:96], 2)}
274       },
```

By specifying the multiplier for each call according to the column-mixing constant matrix.

However, this is where the biggest mistake was made. I had forgotten to modify the `aes_tbox` function to return 8 bytes instead of the original 24 bytes:

```
578 function [7:0] aes_tbox(input  [7:0]in, input integer mult);
```

In Verilog, if you define a function to return more bytes than the value that you assign to the output of the function; the remaining bytes will be padded with 0. This meant that while

testing, the result of aes_tbox was always 8'b00000000 causing wrong results. It was only after fixing line 578 to return 8 bytes from aes_tbox that this bug was fixed and the whole module worked. Changing function [23:0] aes_tbox to the correct function [7:0] aes_tbox.

4.2 Optimisation

4.2.1 Checking for Round Keys Re-use

From the aes_engine RTL simulation waveform in Figure 3.3; we can see that generating the round keys takes much much longer than performing the encryption rounds, and usually the user will want to encrypt more than just one plaintext block (16 bytes) with the same key.

This means that we should optimise the design to allow for encrypting many plaintext blocks without needing to re-compute the round keys every single time. In the original design the transformer_done control signal as used to reset all the modules. However, this does not allow for the re-use of the round-keys. An optimisation was made so that transformer_done only resets the input_interface to its idle state but does not clear the contents of the round keys in engine_keys_generator. A further optimisation was done to allow pipelining.

4.2.2 Concurrent Pipelining

A design choice to allow for pipelining is the inclusion of the output_read control signal. This signal is sent from the output_interface to the enging_round_transformer so that it can clear its ciphertext to get ready to encrypt the next plaintext. By doing this, the input_interface can immediately return to idle and accept the next plaintext while the engine_round_transformer is cleared and the output_interface is still busy outputting the current ciphertext.

Additionally, the engine_key_generator submodule also checks if the requested key on key_start has already been computed, if yes, it skips the re-computation and immediately signals transformer_start:

```
72      if (key_start && !edgedetect_keystart) begin
73        // ? if the key is the same as the last computed key, dont recompute
74        // ? just issue transformer start
75        if (i > 43 && round_keys[0] == key_in) begin
76          // keys already computed
77          i <= 44;
78          $display("----------------");
79          $display("Same key requested, skipping computation.");
80        end
81        else if (i == 0 || i>43) begin
82          // new key requested
83          i <= 0;
84          $display("----------------");
85          $write("Generating round keys for key %032X\n", key_in);
86        end
87      end
```

Thanks to this design choice, it becomes possible to input the next plaintext at the same time as reading the last ciphertext. This allows for fast back-to-back encryption rounds of many plaintext blocks using the same cipherkey without the need to re-generate the round keys for optimum speed.

This is best illustrated by examining Figure 3.3. If we zoom into the section where the ciphertext is outputted and overlay the signals if we were to input the next plaintext during the output of the ciphertext:

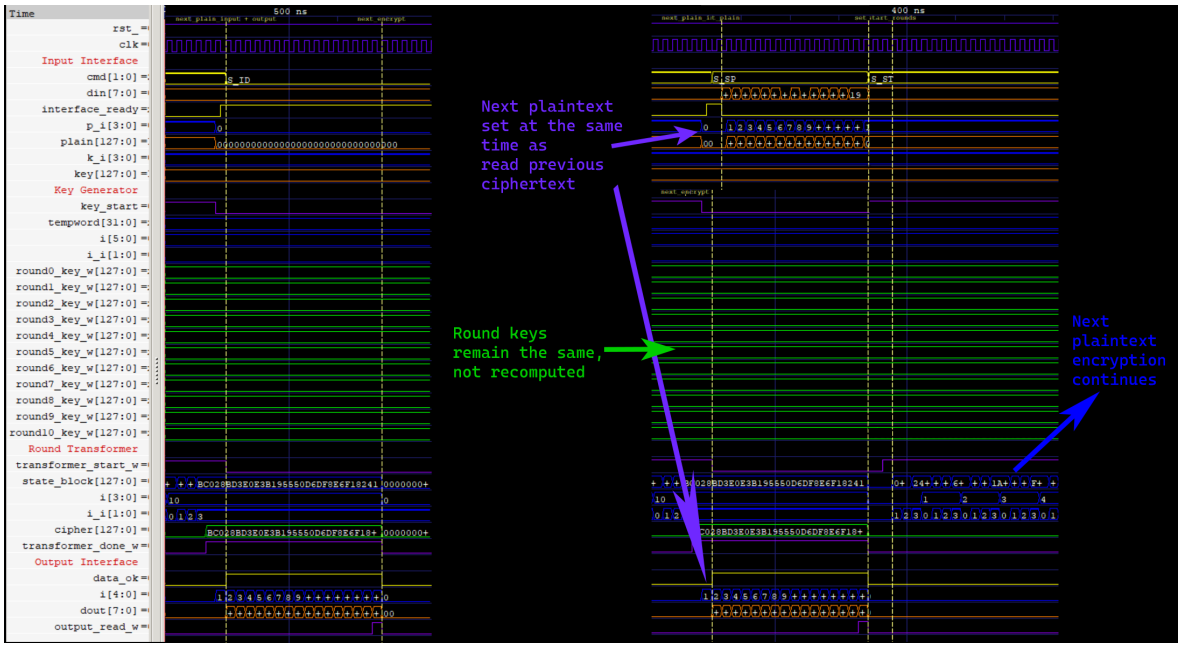


Figure 4.1: Waveform for pipelinging aes_engine

We can see that the input interface is ready to accept commands, the key generator still holds the roundkeys for the current cipherkey, and the round transformer has been reset. Allowing us to simultaneously read the last plaintext block's ciphertext and input the next plaintext block to be encrypted with the same cipherkey.

This means that, if we continuously read the ciphertext while inputting the next plaintext. Given that the encryption rounds take 94ns to compute, the theoretical maximum throughput is $1.063 \cdot 10^7$ plaintext blocks per second, or 162.328 MB/s.

5. RTL Synthesis

5.1 Preparing for Synthesis

In order to make sure the code compiled when performing synthesis using Synopsis, the following changes had to be made:

1. Use asynchronous assignments (`<=`) inside `always` blocks
2. Only assign one variable per execution of an `always @(posedge clk)` through the use of sub-loop counters (usually `i_i` and nested if statements)
3. Only assigning variables in a single `always @(posedge clk)` block (no `always` on edges of any other signals)

The first requirement brought no difficulties per se, however it's consequences lead to great complications. Since it means that variable values are assigned asynchronously at the end of the `always` block. Variables that are assigned and then immediately used will have the wrong value and those steps must be moved to the next execution of the block. Creating the need to break up every algorithm step requiring immediate assignment into sub-steps.

The second requirement in particular, greatly complicated the code. As before, assigning variables synchronously allowed a variable to be assigned multiple times and it's value reused within the same clock cycle. Now each variable may only be assigned once per clock cycle. In order to obey this, many modules needed sub-loop counters implemented using `i_i` and nested if statements. This means that not only must we write more logic to keep track of the current step of the algorithm, but also, the code runs `i_i` times slower; as a step that took one cycle before now takes `i_i` clock cycles in order to step through each asynchronous assignment. Generally, as `i_i` had usually four steps, this made the design four times slower.

The third requirement, complicated the code. As blocks could not be run triggered by the edge of a control or reset signal. All the control signals have to be checked for with nested if statements inside of the main `always @(posedge clk)` block.

This means that for some signals such as `key_start` an **edge detector** had to be programmed; where we store the last value of `key_start` in the variable `edgedetect_keystart` and use a comparison to detect if a positive edge (`key_start && !edgedetect_keystart`) or negative edge (`!key_start && edgedetect_keystart`) occurred at the clock pulse.

5.2 Synthesis Results

5.2.1 Synthesis Schematics

Below are the resulting logic gate schematics when the Synopsis synthesis result is opened with DesignVision.

5.2.1.1 Top Module

First we can see the topmost chip `TOP_PAD_opt` and its input and output pins:

Figure 5.1: Schematic of TOP_PAD_opt

Viewing the insides of this module, we can see the pin drivers and how they connect to `chip_core`:

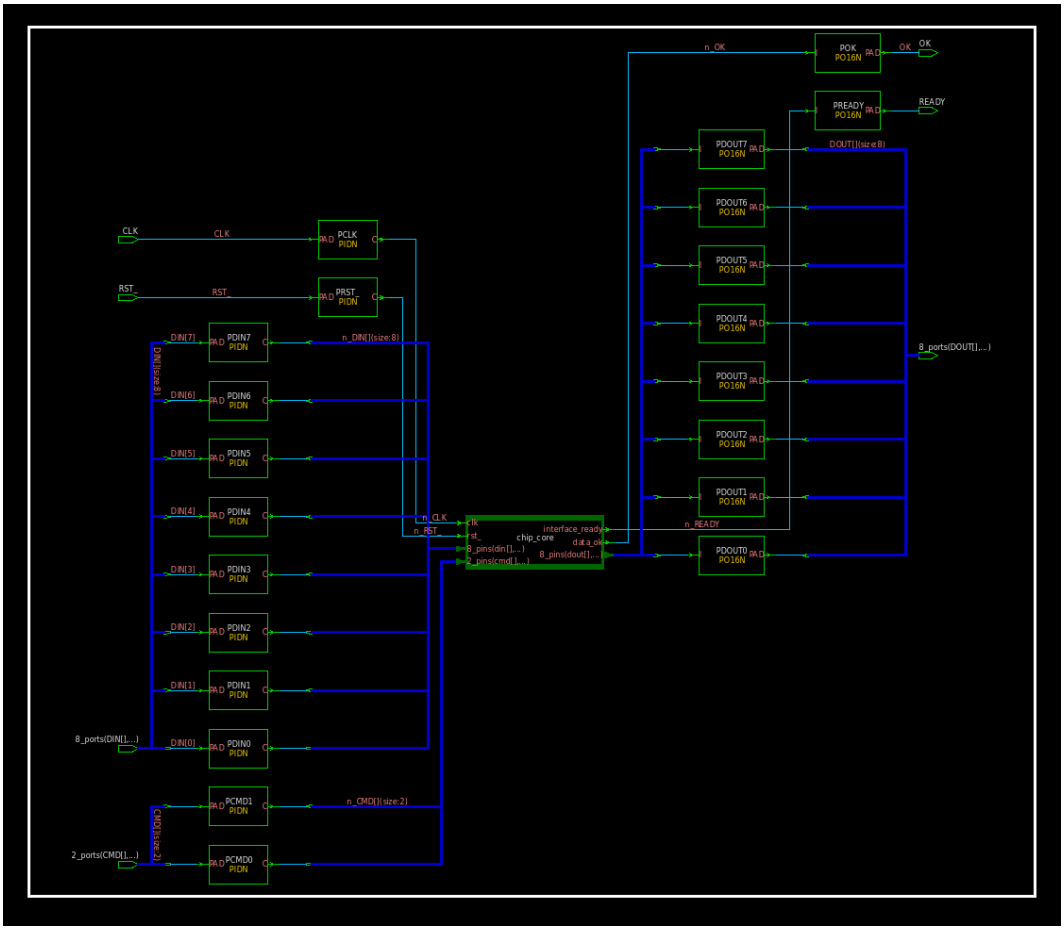


Figure 5.2: Schematic of inside of TOP_PAD_opt

Entering the `chip_core` submodule, we can see another layer of input/output drivers:

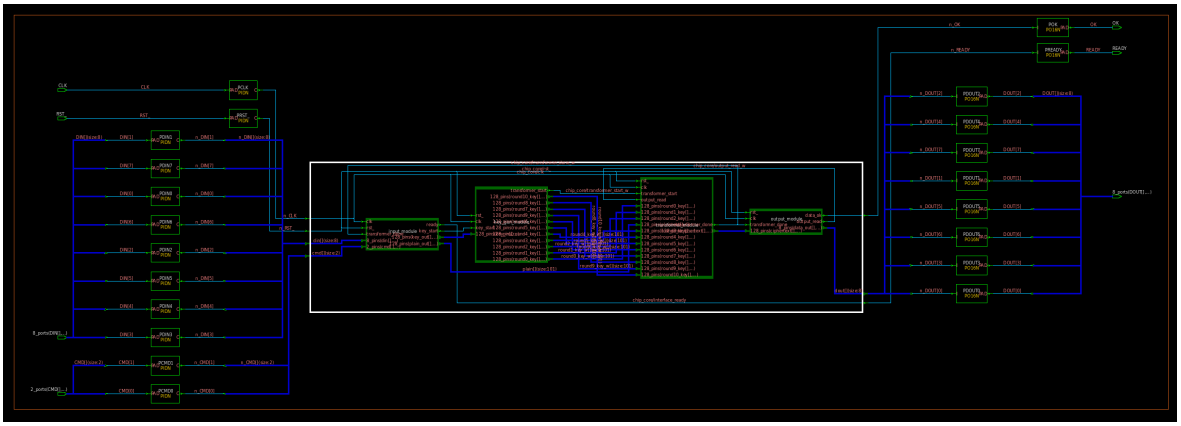


Figure 5.3: Schematic of inside of chip_core

Here we see how our submodules are connected:

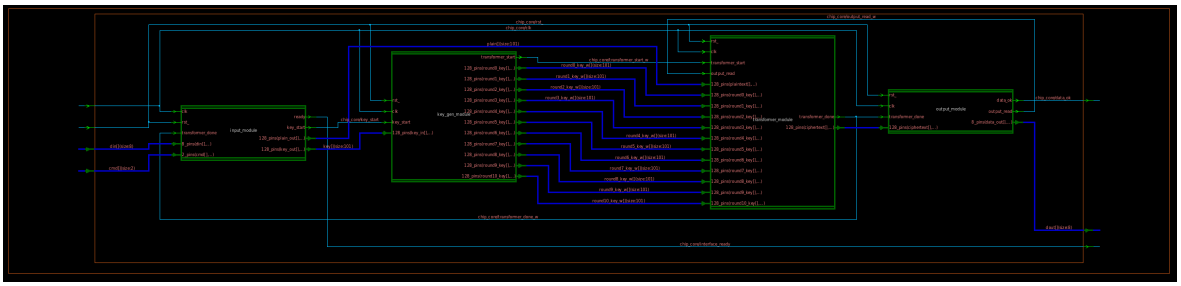


Figure 5.4: Schematic of inside of chip_core submodules

5.2.1.2 Input Interface

Below is the schematic for the `input_interface` module:

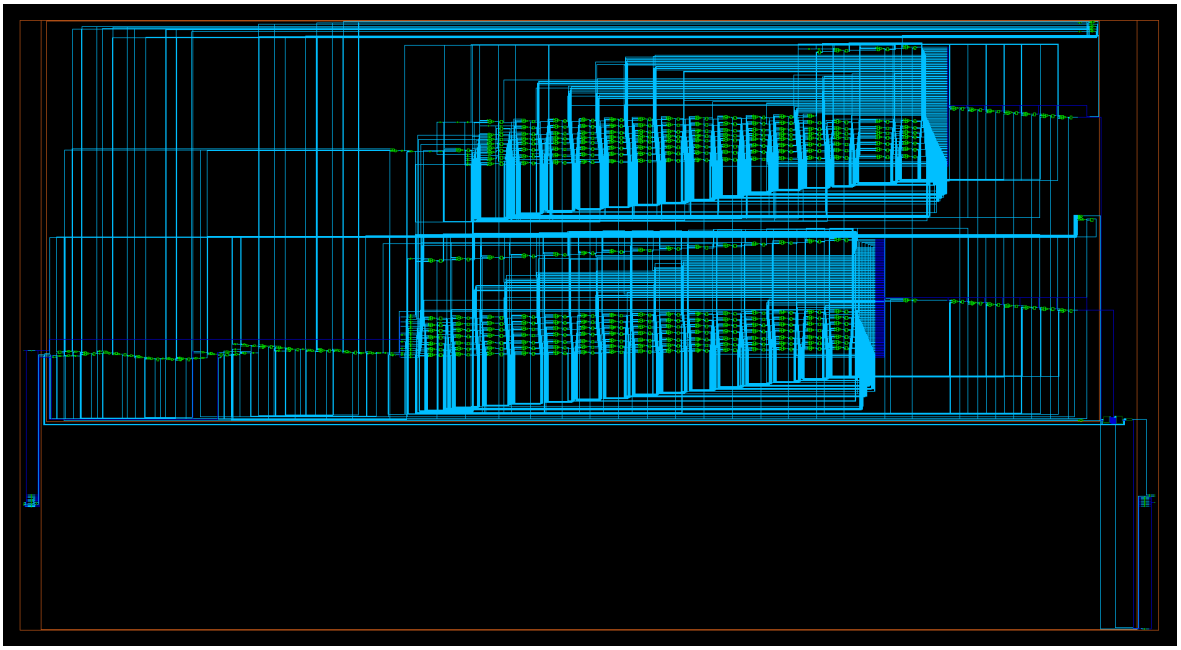


Figure 5.5: Schematic of input_interface

5.2.1.3 Round Keys Generator

Below is the schematic for the `engine_key_generator` module:

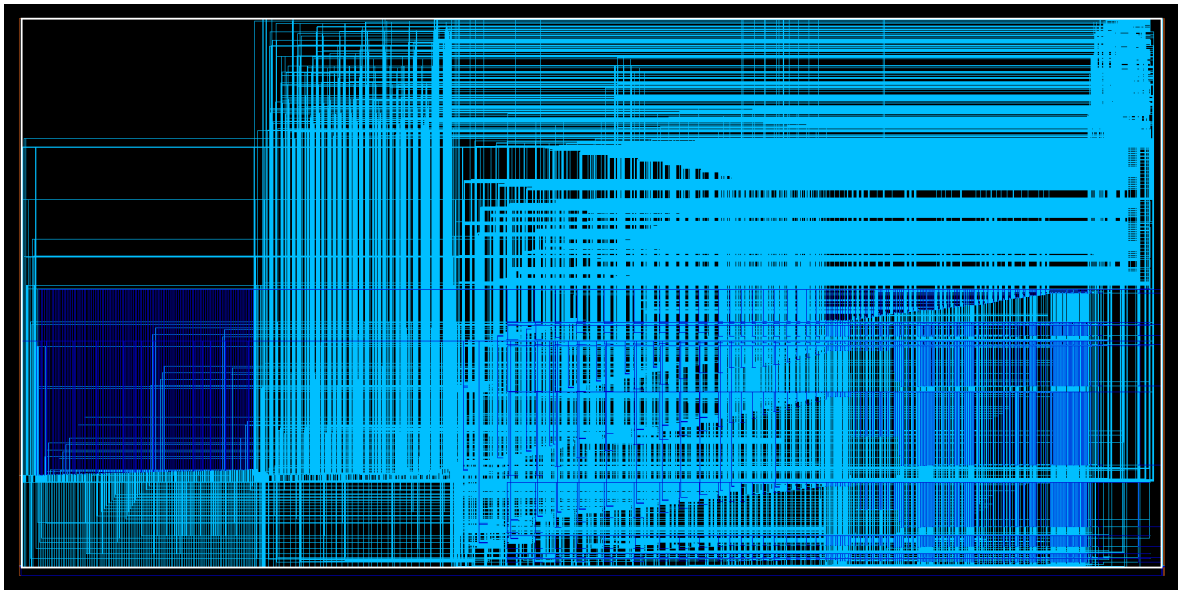


Figure 5.6: Schematic of engine_key_generator

Note that there are so many cyan wires, that the numerous green logic gates are too small to be seen without zooming in.

5.2.1.4 Round Transformer

Below is the schematic for the `engine_round_transformer` module:

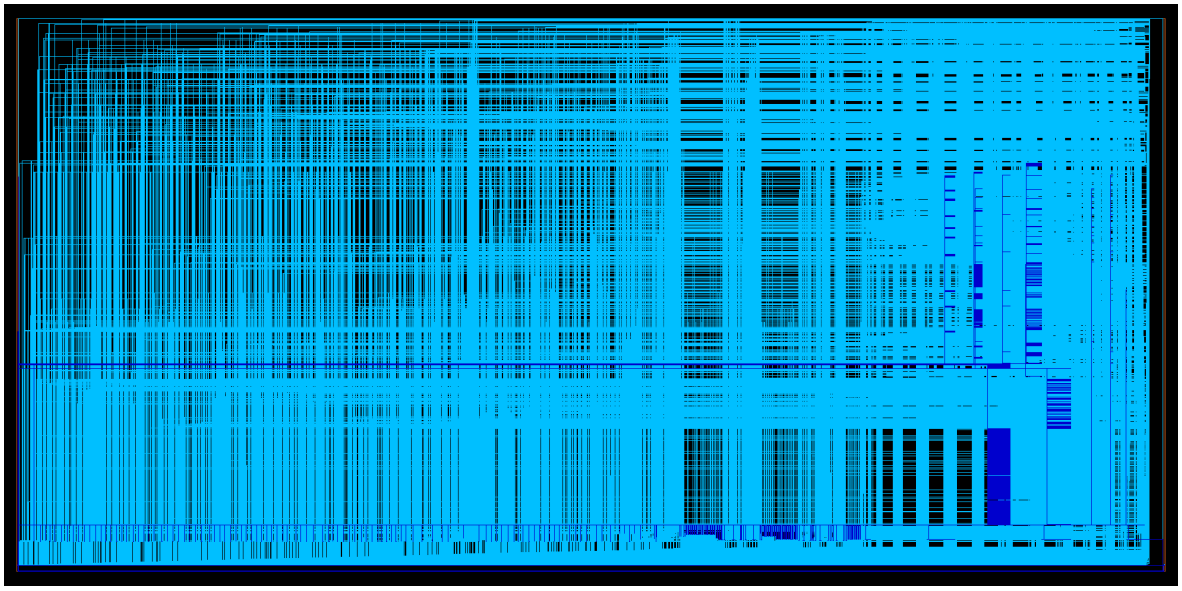


Figure 5.7: Schematic of engine_round_transformer

Note that there are so many cyan wires, that the numerous green logic gates are too small to be seen without zooming in.

5.2.1.5 Output Interface

Below is the schematic for the `output_interface` module:

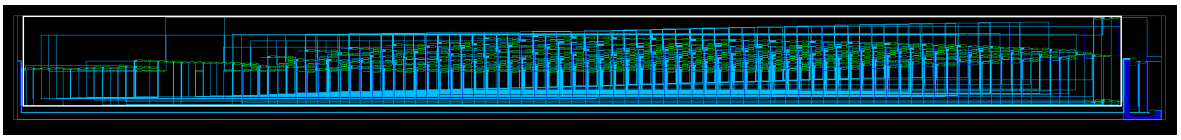


Figure 5.8: Schematic of output_interface

5.2.2 Synopsis Synthesis Reports

5.2.2.1 Area Report

```
1
2  ****************************************
3  Report : area
4  Design : TOP_PAD_opt
5  Version: O-2018.06-SP1
6  Date   : Fri Apr  5 01:05:05 2024
7  ****************************************
8
9  Library(s) Used:
10
11     slow (File: /opt/tech_lib/smic180/digital/sc/synopsys/slow.db)
12     SP018N_V1p0_typ (File:
       ↪  /opt/tech_lib/smic180/digital/io/synopsys/SP018N_V1p0_typ.db)
13
14 Number of ports:                         3928
15 Number of nets:                         36252
16 Number of cells:                        31980
17 Number of combinational cells:          27033
18 Number of sequential cells:              4941
19 Number of macros/black boxes:               0
20 Number of buf/inv:                       4075
21 Number of references:                       3
22
23 Combinational area:             613029.406203
24 Buf/Inv area:                   213821.485530
25 Noncombinational area:          271304.514019
26 Macro/Black Box area:                0.000000
27 Net Interconnect area:        21161345.245010
28
29 Total cell area:                884333.920222
30 Total area:                   22045679.165233
31 1
```

5.2.2.2 Timing Report

```
1
2  ****************************************
3  Report : timing
4          -path full
5          -delay max
```

```
        -nworst 10
        -max_paths 10
Design : TOP_PAD_opt
Version: O-2018.06-SP1
Date   : Fri Apr  5 01:05:05 2024
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: slow   Library: slow
Wire Load Model Mode: top

  Startpoint: chip_core/transformer_module/i_reg[1]
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint: chip_core/transformer_module/state_block_reg[25]
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  TOP_PAD_opt        smic18_wl50           slow

  Point                                                   Incr       Path
  ---------------------------------------------------------------------------
  clock clk (rise edge)                                   0.00       0.00
  clock network delay (ideal)                             3.00       3.00
  chip_core/transformer_module/i_reg[1]/CK (DFFHQX1)      0.00 #     3.00 r
  chip_core/transformer_module/i_reg[1]/Q (DFFHQX1)       2.11       5.11 r
  chip_core/transformer_module/U273/Y (OR3XL)             1.37       6.48 r
  chip_core/transformer_module/U9848/Y (INVX1)            0.87       7.35 f
  chip_core/transformer_module/U3923/Y (INVX1)            1.02       8.37 r
  chip_core/transformer_module/U1039/Y (CLKINVX3)         1.61       9.99 f
  chip_core/transformer_module/U14882/Y (NAND2X1)         1.65      11.64 r
  chip_core/transformer_module/U9846/Y (NAND2X1)          1.18      12.82 f
  chip_core/transformer_module/U3921/Y (INVX1)            1.51      14.33 r
  chip_core/transformer_module/U9845/Y (NAND2X1)          0.88      15.21 f
  chip_core/transformer_module/U3922/Y (INVX1)            1.49      16.70 r
  chip_core/transformer_module/U1037/Y (CLKINVX3)         2.26      18.95 f
  chip_core/transformer_module/U3920/Y (OAI21XL)          1.86      20.81 r
  chip_core/transformer_module/U941/Y (OAI21X1)           0.65      21.46 f
  chip_core/transformer_module/U9837/Y (OAI211X1)         1.62      23.08 r
  chip_core/transformer_module/U9836/Y (INVX1)            1.78      24.86 f
  chip_core/transformer_module/U1014/Y (INVX1)            1.57      26.42 r
  chip_core/transformer_module/U3907/Y (AND2X2)           0.77      27.20 r
  chip_core/transformer_module/U9835/Y (NAND2X1)          1.63      28.83 f
  chip_core/transformer_module/U3911/Y (INVX1)            2.40      31.23 r
  chip_core/transformer_module/U1016/Y (INVX1)            1.85      33.08 f
  chip_core/transformer_module/U631/Y (INVX1)             2.29      35.37 r
  chip_core/transformer_module/U630/Y (INVX1)             1.86      37.24 f
  chip_core/transformer_module/U609/Y (INVX1)             2.30      39.54 r
  chip_core/transformer_module/U11727/Y (AOI32X1)         0.90      40.44 f
  chip_core/transformer_module/U11725/Y (OAI221XL)        0.84      41.28 r
  chip_core/transformer_module/state_block_reg[25]/D (DFFHQX1)
                                                          0.00      41.28 r
  data arrival time                                                 41.28

  clock clk (rise edge)                                  40.00      40.00
```

```
clock network delay (ideal)                              3.00       43.00
clock uncertainty                                       -1.30       41.70
chip_core/transformer_module/state_block_reg[25]/CK (DFFHQX1)
                                                         0.00       41.70 r
library setup time                                      -0.23       41.47
data required time                                                  41.47
--------------------------------------------------------------------------
data required time                                                  41.47
data arrival time                                                  -41.28
--------------------------------------------------------------------------
slack (MET)                                                         0.19


Startpoint: chip_core/transformer_module/i_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: chip_core/transformer_module/state_block_reg[123]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
TOP_PAD_opt        smic18_wl50           slow

Point                                                    Incr       Path
--------------------------------------------------------------------------
clock clk (rise edge)                                    0.00       0.00
clock network delay (ideal)                              3.00       3.00
chip_core/transformer_module/i_reg[1]/CK (DFFHQX1)       0.00 #     3.00 r
chip_core/transformer_module/i_reg[1]/Q (DFFHQX1)        2.11       5.11 r
chip_core/transformer_module/U273/Y (OR3XL)              1.37       6.48 r
chip_core/transformer_module/U9848/Y (INVX1)             0.87       7.35 f
chip_core/transformer_module/U3923/Y (INVX1)             1.02       8.37 r
chip_core/transformer_module/U1039/Y (CLKINVX3)          1.61       9.99 f
chip_core/transformer_module/U14882/Y (NAND2X1)          1.65      11.64 r
chip_core/transformer_module/U9846/Y (NAND2X1)           1.18      12.82 f
chip_core/transformer_module/U3921/Y (INVX1)             1.51      14.33 r
chip_core/transformer_module/U9845/Y (NAND2X1)           0.88      15.21 f
chip_core/transformer_module/U3922/Y (INVX1)             1.49      16.70 r
chip_core/transformer_module/U1037/Y (CLKINVX3)          2.26      18.95 f
chip_core/transformer_module/U3920/Y (OAI21XL)           1.86      20.81 r
chip_core/transformer_module/U941/Y (OAI21X1)            0.65      21.46 f
chip_core/transformer_module/U9837/Y (OAI211X1)          1.62      23.08 r
chip_core/transformer_module/U9836/Y (INVX1)             1.78      24.86 f
chip_core/transformer_module/U1014/Y (INVX1)             1.57      26.42 r
chip_core/transformer_module/U3907/Y (AND2X2)            0.77      27.20 r
chip_core/transformer_module/U9835/Y (NAND2X1)           1.63      28.83 f
chip_core/transformer_module/U3911/Y (INVX1)             2.40      31.23 r
chip_core/transformer_module/U1016/Y (INVX1)             1.85      33.08 f
chip_core/transformer_module/U631/Y (INVX1)              2.29      35.37 r
chip_core/transformer_module/U630/Y (INVX1)              1.86      37.24 f
chip_core/transformer_module/U609/Y (INVX1)              2.30      39.54 r
chip_core/transformer_module/U11771/Y (AOI32X1)          0.90      40.44 f
chip_core/transformer_module/U11769/Y (OAI221XL)         0.84      41.28 r
chip_core/transformer_module/state_block_reg[123]/D (DFFHQX1)
                                                         0.00      41.28 r
data arrival time                                                  41.28
```

```
clock clk (rise edge)                                   40.00      40.00
clock network delay (ideal)                              3.00      43.00
clock uncertainty                                       -1.30      41.70
chip_core/transformer_module/state_block_reg[123]/CK (DFFHQX1)
                                                         0.00      41.70 r
library setup time                                      -0.23      41.47
data required time                                                 41.47
--------------------------------------------------------------------------
data required time                                                 41.47
data arrival time                                                 -41.28
--------------------------------------------------------------------------
slack (MET)                                                        0.19


Startpoint: chip_core/transformer_module/i_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: chip_core/transformer_module/state_block_reg[124]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
TOP_PAD_opt        smic18_wl50           slow

Point                                                    Incr       Path
--------------------------------------------------------------------------
clock clk (rise edge)                                    0.00       0.00
clock network delay (ideal)                              3.00       3.00
chip_core/transformer_module/i_reg[1]/CK (DFFHQX1)       0.00 #     3.00 r
chip_core/transformer_module/i_reg[1]/Q (DFFHQX1)        2.11       5.11 r
chip_core/transformer_module/U273/Y (OR3XL)              1.37       6.48 r
chip_core/transformer_module/U9848/Y (INVX1)             0.87       7.35 f
chip_core/transformer_module/U3923/Y (INVX1)             1.02       8.37 r
chip_core/transformer_module/U1039/Y (CLKINVX3)          1.61       9.99 f
chip_core/transformer_module/U14882/Y (NAND2X1)          1.65      11.64 r
chip_core/transformer_module/U9846/Y (NAND2X1)           1.18      12.82 f
chip_core/transformer_module/U3921/Y (INVX1)             1.51      14.33 r
chip_core/transformer_module/U9845/Y (NAND2X1)           0.88      15.21 f
chip_core/transformer_module/U3922/Y (INVX1)             1.49      16.70 r
chip_core/transformer_module/U1037/Y (CLKINVX3)          2.26      18.95 f
chip_core/transformer_module/U3920/Y (OAI21XL)           1.86      20.81 r
chip_core/transformer_module/U941/Y (OAI21X1)            0.65      21.46 f
chip_core/transformer_module/U9837/Y (OAI211X1)          1.62      23.08 r
chip_core/transformer_module/U9836/Y (INVX1)             1.78      24.86 f
chip_core/transformer_module/U1014/Y (INVX1)             1.57      26.42 r
chip_core/transformer_module/U3907/Y (AND2X2)            0.77      27.20 r
chip_core/transformer_module/U9835/Y (NAND2X1)           1.63      28.83 f
chip_core/transformer_module/U3911/Y (INVX1)             2.40      31.23 r
chip_core/transformer_module/U1016/Y (INVX1)             1.85      33.08 f
chip_core/transformer_module/U631/Y (INVX1)              2.29      35.37 r
chip_core/transformer_module/U630/Y (INVX1)              1.86      37.24 f
chip_core/transformer_module/U610/Y (INVX1)              2.30      39.54 r
chip_core/transformer_module/U11767/Y (AOI32X1)          0.90      40.44 f
chip_core/transformer_module/U11765/Y (OAI221XL)         0.84      41.28 r
chip_core/transformer_module/state_block_reg[124]/D (DFFHQX1)
                                                         0.00      41.28 r
data arrival time                                                  41.28
```

```
                                                          40.00       40.00
clock clk (rise edge)                                      3.00       43.00
clock network delay (ideal)                               -1.30       41.70
chip_core/transformer_module/state_block_reg[124]/CK (DFFHQX1)
                                                           0.00       41.70 r
library setup time                                        -0.23       41.47
data required time                                                    41.47
--------------------------------------------------------------------------
data required time                                                    41.47
data arrival time                                                    -41.28
--------------------------------------------------------------------------
slack (MET)                                                           0.19


Startpoint: chip_core/transformer_module/i_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: chip_core/transformer_module/state_block_reg[126]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
TOP_PAD_opt        smic18_wl50           slow

Point                                                     Incr       Path
--------------------------------------------------------------------------
clock clk (rise edge)                                     0.00       0.00
clock network delay (ideal)                               3.00       3.00
chip_core/transformer_module/i_reg[1]/CK (DFFHQX1)        0.00 #     3.00 r
chip_core/transformer_module/i_reg[1]/Q (DFFHQX1)         2.11       5.11 r
chip_core/transformer_module/U273/Y (OR3XL)               1.37       6.48 r
chip_core/transformer_module/U9848/Y (INVX1)              0.87       7.35 f
chip_core/transformer_module/U3923/Y (INVX1)              1.02       8.37 r
chip_core/transformer_module/U1039/Y (CLKINVX3)           1.61       9.99 f
chip_core/transformer_module/U14882/Y (NAND2X1)           1.65      11.64 r
chip_core/transformer_module/U9846/Y (NAND2X1)            1.18      12.82 f
chip_core/transformer_module/U3921/Y (INVX1)              1.51      14.33 r
chip_core/transformer_module/U9845/Y (NAND2X1)            0.88      15.21 f
chip_core/transformer_module/U3922/Y (INVX1)              1.49      16.70 r
chip_core/transformer_module/U1037/Y (CLKINVX3)           2.26      18.95 f
chip_core/transformer_module/U3920/Y (OAI21XL)            1.86      20.81 r
chip_core/transformer_module/U941/Y (OAI21X1)             0.65      21.46 f
chip_core/transformer_module/U9837/Y (OAI211X1)           1.62      23.08 r
chip_core/transformer_module/U9836/Y (INVX1)              1.78      24.86 f
chip_core/transformer_module/U1014/Y (INVX1)              1.57      26.42 r
chip_core/transformer_module/U3907/Y (AND2X2)             0.77      27.20 r
chip_core/transformer_module/U9835/Y (NAND2X1)            1.63      28.83 f
chip_core/transformer_module/U3911/Y (INVX1)              2.40      31.23 r
chip_core/transformer_module/U1016/Y (INVX1)              1.85      33.08 f
chip_core/transformer_module/U631/Y (INVX1)               2.29      35.37 r
chip_core/transformer_module/U630/Y (INVX1)               1.86      37.24 f
chip_core/transformer_module/U608/Y (INVX1)               2.30      39.54 r
chip_core/transformer_module/U11763/Y (AOI32X1)           0.90      40.44 f
chip_core/transformer_module/U11761/Y (OAI221XL)          0.84      41.28 r
chip_core/transformer_module/state_block_reg[126]/D (DFFX2)
                                                          0.00      41.28 r
```

```
data arrival time                                                  41.28

clock clk (rise edge)                                 40.00        40.00
clock network delay (ideal)                            3.00        43.00
clock uncertainty                                     -1.30        41.70
chip_core/transformer_module/state_block_reg[126]/CK (DFFX2)
                                                       0.00        41.70 r
library setup time                                    -0.21        41.49
data required time                                                 41.49
-----------------------------------------------------------------------------
data required time                                                 41.49
data arrival time                                                 -41.28
-----------------------------------------------------------------------------
slack (MET)                                                        0.21


Startpoint: chip_core/transformer_module/i_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: chip_core/transformer_module/state_block_reg[125]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
TOP_PAD_opt        smic18_wl50           slow

Point                                                    Incr       Path
-----------------------------------------------------------------------------
clock clk (rise edge)                                    0.00       0.00
clock network delay (ideal)                              3.00       3.00
chip_core/transformer_module/i_reg[1]/CK (DFFHQX1)       0.00 #     3.00 r
chip_core/transformer_module/i_reg[1]/Q (DFFHQX1)        2.11       5.11 r
chip_core/transformer_module/U273/Y (OR3XL)              1.37       6.48 r
chip_core/transformer_module/U9848/Y (INVX1)             0.87       7.35 f
chip_core/transformer_module/U3923/Y (INVX1)             1.02       8.37 r
chip_core/transformer_module/U1039/Y (CLKINVX3)          1.61       9.99 f
chip_core/transformer_module/U14882/Y (NAND2X1)          1.65      11.64 r
chip_core/transformer_module/U9846/Y (NAND2X1)           1.18      12.82 f
chip_core/transformer_module/U3921/Y (INVX1)             1.51      14.33 r
chip_core/transformer_module/U9845/Y (NAND2X1)           0.88      15.21 f
chip_core/transformer_module/U3922/Y (INVX1)             1.49      16.70 r
chip_core/transformer_module/U1037/Y (CLKINVX3)          2.26      18.95 f
chip_core/transformer_module/U3920/Y (OAI21XL)           1.86      20.81 r
chip_core/transformer_module/U941/Y (OAI21X1)            0.65      21.46 f
chip_core/transformer_module/U9837/Y (OAI211X1)          1.62      23.08 r
chip_core/transformer_module/U9836/Y (INVX1)             1.78      24.86 f
chip_core/transformer_module/U1014/Y (INVX1)             1.57      26.42 r
chip_core/transformer_module/U3907/Y (AND2X2)            0.77      27.20 r
chip_core/transformer_module/U9835/Y (NAND2X1)           1.63      28.83 f
chip_core/transformer_module/U3911/Y (INVX1)             2.40      31.23 r
chip_core/transformer_module/U1016/Y (INVX1)             1.85      33.08 f
chip_core/transformer_module/U631/Y (INVX1)              2.29      35.37 r
chip_core/transformer_module/U630/Y (INVX1)              1.86      37.24 f
chip_core/transformer_module/U610/Y (INVX1)              2.30      39.54 r
chip_core/transformer_module/U11775/Y (AOI32X1)          0.90      40.44 f
chip_core/transformer_module/U11773/Y (OAI221XL)         0.83      41.28 r
chip_core/transformer_module/state_block_reg[125]/D (DFFX1)
```

```
                                                          0.00       41.28 r
data arrival time                                                    41.28

clock clk (rise edge)                                    40.00       40.00
clock network delay (ideal)                               3.00       43.00
clock uncertainty                                        -1.30       41.70
chip_core/transformer_module/state_block_reg[125]/CK (DFFX1)
                                                          0.00       41.70 r
library setup time                                       -0.20       41.50
data required time                                                   41.50
--------------------------------------------------------------------------
data required time                                                   41.50
data arrival time                                                  -41.28
--------------------------------------------------------------------------
slack (MET)                                                          0.22


Startpoint: chip_core/transformer_module/i_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: chip_core/transformer_module/state_block_reg[127]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
TOP_PAD_opt        smic18_wl50           slow

Point                                                    Incr       Path
--------------------------------------------------------------------------
clock clk (rise edge)                                    0.00       0.00
clock network delay (ideal)                              3.00       3.00
chip_core/transformer_module/i_reg[1]/CK (DFFHQX1)       0.00 #     3.00 r
chip_core/transformer_module/i_reg[1]/Q (DFFHQX1)        2.11       5.11 r
chip_core/transformer_module/U273/Y (OR3XL)              1.37       6.48 r
chip_core/transformer_module/U9848/Y (INVX1)             0.87       7.35 f
chip_core/transformer_module/U3923/Y (INVX1)             1.02       8.37 r
chip_core/transformer_module/U1039/Y (CLKINVX3)          1.61       9.99 f
chip_core/transformer_module/U14882/Y (NAND2X1)          1.65      11.64 r
chip_core/transformer_module/U9846/Y (NAND2X1)           1.18      12.82 f
chip_core/transformer_module/U3921/Y (INVX1)             1.51      14.33 r
chip_core/transformer_module/U9845/Y (NAND2X1)           0.88      15.21 f
chip_core/transformer_module/U3922/Y (INVX1)             1.49      16.70 r
chip_core/transformer_module/U1037/Y (CLKINVX3)          2.26      18.95 f
chip_core/transformer_module/U3920/Y (OAI21XL)           1.86      20.81 r
chip_core/transformer_module/U941/Y (OAI21X1)            0.65      21.46 f
chip_core/transformer_module/U9837/Y (OAI211X1)          1.62      23.08 r
chip_core/transformer_module/U9836/Y (INVX1)             1.78      24.86 f
chip_core/transformer_module/U1014/Y (INVX1)             1.57      26.42 r
chip_core/transformer_module/U3907/Y (AND2X2)            0.77      27.20 r
chip_core/transformer_module/U9835/Y (NAND2X1)           1.63      28.83 f
chip_core/transformer_module/U3911/Y (INVX1)             2.40      31.23 r
chip_core/transformer_module/U1016/Y (INVX1)             1.85      33.08 f
chip_core/transformer_module/U631/Y (INVX1)              2.29      35.37 r
chip_core/transformer_module/U630/Y (INVX1)              1.86      37.24 f
chip_core/transformer_module/U608/Y (INVX1)              2.30      39.54 r
chip_core/transformer_module/U11759/Y (AOI32X1)          0.90      40.44 f
chip_core/transformer_module/U11757/Y (OAI221XL)         0.83      41.28 r
```

```
chip_core/transformer_module/state_block_reg[127]/D (DFFX1)
                                                          0.00       41.28 r
data arrival time                                                    41.28

clock clk (rise edge)                                    40.00       40.00
clock network delay (ideal)                               3.00       43.00
clock uncertainty                                        -1.30       41.70
chip_core/transformer_module/state_block_reg[127]/CK (DFFX1)
                                                          0.00       41.70 r
library setup time                                       -0.20       41.50
data required time                                                   41.50
--------------------------------------------------------------------------
data required time                                                   41.50
data arrival time                                                  -41.28
--------------------------------------------------------------------------
slack (MET)                                                          0.22


Startpoint: chip_core/transformer_module/i_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: chip_core/transformer_module/state_block_reg[122]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
TOP_PAD_opt        smic18_wl50           slow

Point                                                    Incr       Path
--------------------------------------------------------------------------
clock clk (rise edge)                                    0.00       0.00
clock network delay (ideal)                              3.00       3.00
chip_core/transformer_module/i_reg[1]/CK (DFFHQX1)       0.00 #     3.00 r
chip_core/transformer_module/i_reg[1]/Q (DFFHQX1)        2.11       5.11 r
chip_core/transformer_module/U273/Y (OR3XL)              1.37       6.48 r
chip_core/transformer_module/U9848/Y (INVX1)             0.87       7.35 f
chip_core/transformer_module/U3923/Y (INVX1)             1.02       8.37 r
chip_core/transformer_module/U1039/Y (CLKINVX3)          1.61       9.99 f
chip_core/transformer_module/U14882/Y (NAND2X1)          1.65      11.64 r
chip_core/transformer_module/U9846/Y (NAND2X1)           1.18      12.82 f
chip_core/transformer_module/U3921/Y (INVX1)             1.51      14.33 r
chip_core/transformer_module/U9845/Y (NAND2X1)           0.88      15.21 f
chip_core/transformer_module/U3922/Y (INVX1)             1.49      16.70 r
chip_core/transformer_module/U1037/Y (CLKINVX3)          2.26      18.95 f
chip_core/transformer_module/U3920/Y (OAI21XL)           1.86      20.81 r
chip_core/transformer_module/U941/Y (OAI21X1)            0.65      21.46 f
chip_core/transformer_module/U9837/Y (OAI211X1)          1.62      23.08 r
chip_core/transformer_module/U9836/Y (INVX1)             1.78      24.86 f
chip_core/transformer_module/U1014/Y (INVX1)             1.57      26.42 r
chip_core/transformer_module/U3907/Y (AND2X2)            0.77      27.20 r
chip_core/transformer_module/U9835/Y (NAND2X1)           1.63      28.83 f
chip_core/transformer_module/U3911/Y (INVX1)             2.40      31.23 r
chip_core/transformer_module/U1016/Y (INVX1)             1.85      33.08 f
chip_core/transformer_module/U631/Y (INVX1)              2.29      35.37 r
chip_core/transformer_module/U630/Y (INVX1)              1.86      37.24 f
chip_core/transformer_module/U609/Y (INVX1)              2.30      39.54 r
chip_core/transformer_module/U11755/Y (AOI32X1)          0.90      40.44 f
```

```
chip_core/transformer_module/U11753/Y (OAI221XL)        0.83       41.28 r
chip_core/transformer_module/state_block_reg[122]/D (DFFX1)
                                                         0.00       41.28 r
data arrival time                                                   41.28

clock clk (rise edge)                                   40.00       40.00
clock network delay (ideal)                              3.00       43.00
clock uncertainty                                       -1.30       41.70
chip_core/transformer_module/state_block_reg[122]/CK (DFFX1)
                                                         0.00       41.70 r
library setup time                                      -0.20       41.50
data required time                                                  41.50
--------------------------------------------------------------------------
data required time                                                  41.50
data arrival time                                                  -41.28
--------------------------------------------------------------------------
slack (MET)                                                         0.22


Startpoint: chip_core/transformer_module/i_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: chip_core/transformer_module/state_block_reg[58]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
TOP_PAD_opt        smic18_wl50           slow

Point                                                    Incr       Path
--------------------------------------------------------------------------
clock clk (rise edge)                                    0.00       0.00
clock network delay (ideal)                              3.00       3.00
chip_core/transformer_module/i_reg[1]/CK (DFFHQX1)       0.00 #     3.00 r
chip_core/transformer_module/i_reg[1]/Q (DFFHQX1)        2.11       5.11 r
chip_core/transformer_module/U273/Y (OR3XL)              1.37       6.48 r
chip_core/transformer_module/U9848/Y (INVX1)             0.87       7.35 f
chip_core/transformer_module/U3923/Y (INVX1)             1.02       8.37 r
chip_core/transformer_module/U1039/Y (CLKINVX3)          1.61       9.99 f
chip_core/transformer_module/U14882/Y (NAND2X1)          1.65      11.64 r
chip_core/transformer_module/U9846/Y (NAND2X1)           1.18      12.82 f
chip_core/transformer_module/U3921/Y (INVX1)             1.51      14.33 r
chip_core/transformer_module/U9845/Y (NAND2X1)           0.88      15.21 f
chip_core/transformer_module/U3922/Y (INVX1)             1.49      16.70 r
chip_core/transformer_module/U1037/Y (CLKINVX3)          2.26      18.95 f
chip_core/transformer_module/U3920/Y (OAI21XL)           1.86      20.81 r
chip_core/transformer_module/U941/Y (OAI21X1)            0.65      21.46 f
chip_core/transformer_module/U9837/Y (OAI211X1)          1.62      23.08 r
chip_core/transformer_module/U9836/Y (INVX1)             1.78      24.86 f
chip_core/transformer_module/U1014/Y (INVX1)             1.57      26.42 r
chip_core/transformer_module/U3907/Y (AND2X2)            0.77      27.20 r
chip_core/transformer_module/U9835/Y (NAND2X1)           1.63      28.83 f
chip_core/transformer_module/U3911/Y (INVX1)             2.40      31.23 r
chip_core/transformer_module/U1016/Y (INVX1)             1.85      33.08 f
chip_core/transformer_module/U631/Y (INVX1)              2.29      35.37 r
chip_core/transformer_module/U630/Y (INVX1)              1.86      37.24 f
chip_core/transformer_module/U608/Y (INVX1)              2.30      39.54 r
```

```
chip_core/transformer_module/U11711/Y (AOI32X1)          0.90       40.44 f
chip_core/transformer_module/U11709/Y (OAI221XL)         0.83       41.28 r
chip_core/transformer_module/state_block_reg[58]/D (DFFX1)
                                                         0.00       41.28 r
data arrival time                                                   41.28

clock clk (rise edge)                                   40.00       40.00
clock network delay (ideal)                              3.00       43.00
clock uncertainty                                       -1.30       41.70
chip_core/transformer_module/state_block_reg[58]/CK (DFFX1)
                                                         0.00       41.70 r
library setup time                                      -0.20       41.50
data required time                                                  41.50
--------------------------------------------------------------------------
data required time                                                  41.50
data arrival time                                                  -41.28
--------------------------------------------------------------------------
slack (MET)                                                         0.22


Startpoint: chip_core/transformer_module/i_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: chip_core/transformer_module/state_block_reg[31]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
TOP_PAD_opt        smic18_wl50           slow

Point                                                    Incr       Path
--------------------------------------------------------------------------
clock clk (rise edge)                                    0.00       0.00
clock network delay (ideal)                              3.00       3.00
chip_core/transformer_module/i_reg[1]/CK (DFFHQX1)       0.00 #     3.00 r
chip_core/transformer_module/i_reg[1]/Q (DFFHQX1)        2.11       5.11 r
chip_core/transformer_module/U273/Y (OR3XL)              1.37       6.48 r
chip_core/transformer_module/U9848/Y (INVX1)             0.87       7.35 f
chip_core/transformer_module/U3923/Y (INVX1)             1.02       8.37 r
chip_core/transformer_module/U1039/Y (CLKINVX3)          1.61       9.99 f
chip_core/transformer_module/U14882/Y (NAND2X1)          1.65      11.64 r
chip_core/transformer_module/U9846/Y (NAND2X1)           1.18      12.82 f
chip_core/transformer_module/U3921/Y (INVX1)             1.51      14.33 r
chip_core/transformer_module/U9845/Y (NAND2X1)           0.88      15.21 f
chip_core/transformer_module/U3922/Y (INVX1)             1.49      16.70 r
chip_core/transformer_module/U1037/Y (CLKINVX3)          2.26      18.95 f
chip_core/transformer_module/U3920/Y (OAI21XL)           1.86      20.81 r
chip_core/transformer_module/U941/Y (OAI21X1)            0.65      21.46 f
chip_core/transformer_module/U9837/Y (OAI211X1)          1.62      23.08 r
chip_core/transformer_module/U9836/Y (INVX1)             1.78      24.86 f
chip_core/transformer_module/U1014/Y (INVX1)             1.57      26.42 r
chip_core/transformer_module/U3907/Y (AND2X2)            0.77      27.20 r
chip_core/transformer_module/U9835/Y (NAND2X1)           1.63      28.83 f
chip_core/transformer_module/U3911/Y (INVX1)             2.40      31.23 r
chip_core/transformer_module/U1016/Y (INVX1)             1.85      33.08 f
chip_core/transformer_module/U631/Y (INVX1)              2.29      35.37 r
chip_core/transformer_module/U603/Y (INVX1)              1.86      37.23 f
```

```
chip_core/transformer_module/U579/Y (INVX1)              2.30      39.54 r
chip_core/transformer_module/U11739/Y (AOI32X1)          0.90      40.44 f
chip_core/transformer_module/U11737/Y (OAI221XL)         0.83      41.28 r
chip_core/transformer_module/state_block_reg[31]/D (DFFX1)
                                                         0.00      41.28 r
data arrival time                                                  41.28

clock clk (rise edge)                                   40.00      40.00
clock network delay (ideal)                              3.00      43.00
clock uncertainty                                       -1.30      41.70
chip_core/transformer_module/state_block_reg[31]/CK (DFFX1)
                                                         0.00      41.70 r
library setup time                                      -0.20      41.50
data required time                                                 41.50
--------------------------------------------------------------------------
data required time                                                 41.50
data arrival time                                                 -41.28
--------------------------------------------------------------------------
slack (MET)                                                        0.23


Startpoint: chip_core/transformer_module/i_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
Endpoint: chip_core/transformer_module/state_block_reg[56]
          (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
TOP_PAD_opt        smic18_wl50           slow

Point                                                    Incr       Path
--------------------------------------------------------------------------
clock clk (rise edge)                                    0.00       0.00
clock network delay (ideal)                              3.00       3.00
chip_core/transformer_module/i_reg[1]/CK (DFFHQX1)       0.00 #     3.00 r
chip_core/transformer_module/i_reg[1]/Q (DFFHQX1)        2.11       5.11 r
chip_core/transformer_module/U273/Y (OR3XL)              1.37       6.48 r
chip_core/transformer_module/U9848/Y (INVX1)             0.87       7.35 f
chip_core/transformer_module/U3923/Y (INVX1)             1.02       8.37 r
chip_core/transformer_module/U1039/Y (CLKINVX3)          1.61       9.99 f
chip_core/transformer_module/U14882/Y (NAND2X1)          1.65      11.64 r
chip_core/transformer_module/U9846/Y (NAND2X1)           1.18      12.82 f
chip_core/transformer_module/U3921/Y (INVX1)             1.51      14.33 r
chip_core/transformer_module/U9845/Y (NAND2X1)           0.88      15.21 f
chip_core/transformer_module/U3922/Y (INVX1)             1.49      16.70 r
chip_core/transformer_module/U1037/Y (CLKINVX3)          2.26      18.95 f
chip_core/transformer_module/U3920/Y (OAI21XL)           1.86      20.81 r
chip_core/transformer_module/U941/Y (OAI21X1)            0.65      21.46 f
chip_core/transformer_module/U9837/Y (OAI211X1)          1.62      23.08 r
chip_core/transformer_module/U9836/Y (INVX1)             1.78      24.86 f
chip_core/transformer_module/U1014/Y (INVX1)             1.57      26.42 r
chip_core/transformer_module/U3907/Y (AND2X2)            0.77      27.20 r
chip_core/transformer_module/U9835/Y (NAND2X1)           1.63      28.83 f
chip_core/transformer_module/U3911/Y (INVX1)             2.40      31.23 r
chip_core/transformer_module/U1016/Y (INVX1)             1.85      33.08 f
chip_core/transformer_module/U632/Y (INVX1)              2.27      35.35 r
```

```
  chip_core/transformer_module/U596/Y (INVX1)              1.85      37.20 f
  chip_core/transformer_module/U565/Y (INVX1)              2.30      39.50 r
  chip_core/transformer_module/U11695/Y (AOI32X1)          0.90      40.41 f
  chip_core/transformer_module/U11693/Y (OAI221XL)         0.84      41.24 r
  chip_core/transformer_module/state_block_reg[56]/D (DFFHQX1)
                                                           0.00      41.24 r
  data arrival time                                                  41.24

  clock clk (rise edge)                                   40.00      40.00
  clock network delay (ideal)                              3.00      43.00
  clock uncertainty                                       -1.30      41.70
  chip_core/transformer_module/state_block_reg[56]/CK (DFFHQX1)
                                                           0.00      41.70 r
  library setup time                                      -0.23      41.47
  data required time                                                 41.47
  --------------------------------------------------------------------------
  data required time                                                 41.47
  data arrival time                                                 -41.24
  --------------------------------------------------------------------------
  slack (MET)                                                         0.23


1
```

5.2.2.3 Power Report

```
Loading db file '/opt/tech_lib/smic180/digital/io/synopsys/SP018N_V1p0_typ.db'
Loading db file '/opt/tech_lib/smic180/digital/sc/synopsys/slow.db'
Information: Propagating switching activity (low effort zero delay simulation).
↪  (PWR-6)
Warning: Design has unannotated primary inputs. (PWR-414)
Warning: Design has unannotated sequential cell outputs. (PWR-415)

****************************************
Report : power
        -analysis_effort low
Design : TOP_PAD_opt
Version: O-2018.06-SP1
Date   : Fri Apr  5 01:05:08 2024
****************************************


Library(s) Used:

    slow (File: /opt/tech_lib/smic180/digital/sc/synopsys/slow.db)
    SP018N_V1p0_typ (File:
    ↪  /opt/tech_lib/smic180/digital/io/synopsys/SP018N_V1p0_typ.db)


Operating Conditions: slow   Library: slow
Wire Load Model Mode: top

Design        Wire Load Model            Library
------------------------------------------------
TOP_PAD_opt            smic18_wl50       slow


```

```
Global Operating Voltage = 1.62
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  =   7.1791 mW   (89%)
  Net Switching Power  = 854.4008 uW   (11%)
                         ---------
Total Dynamic Power    =   8.0335 mW  (100%)

Cell Leakage Power     = 284.6701 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (
↪  %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.4715        3.8898e-02        2.5608e+08            0.7665  (
↪  9.21%)
memory             0.0000            0.0000            0.0000            0.0000  (
↪  0.00%)
black_box          0.0000            0.0000            0.0000            0.0000  (
↪  0.00%)
clock_network      0.0000            0.0000            0.0000            0.0000  (
↪  0.00%)
register           6.6889        2.5616e-02        8.9904e+06            6.7236  (
↪  80.83%)
sequential         0.0000            0.0000            0.0000            0.0000  (
↪  0.00%)
combinational  1.8541e-02            0.7899        1.9600e+07            0.8280  (
↪  9.95%)
--------------------------------------------------------------------------------------------------
Total              7.1790 mW         0.8544 mW     2.8467e+08 pW         8.3181 mW
1
```

6. Verilog Code

6.1 Using IVerilog and GTKWave

In order to speed up the development process, instead of Modelsim, Icarus Verilog and GTKWave were used to quickly simulate and display waveforms and console output.

The following commands were used to simulate each module:

```
# simulate input_interface
clear && iverilog -o build/input_interface_tb.out input_interface.v
↪  input_interface_tb.v && vvp build/input_interface_tb.out && gtkwave
↪  simulation/input_interface.vcd
# simulate engine_key_generator
clear && iverilog -o build/engine_key_generator_tb.out engine_key_generator.v
↪  engine_key_generator_tb.v && vvp build/engine_key_generator_tb.out &&
↪  gtkwave simulation/engine_key_generator.vcd
# simulate engine_round_transformer
clear && iverilog -o build/engine_round_transformer_tb.out
↪  engine_round_transformer.v engine_round_transformer_tb.v && vvp
↪  build/engine_round_transformer_tb.out && gtkwave
↪  simulation/engine_round_transformer.vcd
# simulate output_interface
clear && iverilog -o build/output_interface_tb.out output_interface.v
↪  output_interface_tb.v && vvp build/output_interface_tb.out && gtkwave
↪  simulation/output_interface.vcd

# simulate aes_engine
clear && iverilog -o build/aes_engine_tb.out input_interface.v
↪  engine_key_generator.v engine_round_transformer.v output_interface.v
↪  aes_engine.v aes_engine_tb.v && vvp build/aes_engine_tb.out && gtkwave
↪  gtkwave/aes_engine_testbench.gtkw
```

Additionally, some `.gtkw` save files were edited in order to always open a GTKWave graph with the colour and radix of each waveform already displayed and highlighted. As well as setting a custom enum radix in order to display the state of the `input_interface` module.

Lastly, when using Icarus Verilog, in order to generate an output `.vcd` wave file that can be opened in GTKWave; the following macro must be appended to the end of the current module being simulated:

```
`ifndef TOPMODULE
// the "macro" to dump signals
initial begin
  $dumpfile ("simulation/module_name.vcd");
  $dumpvars(0, module_name);
end
`endif
```

Where we replace `module_name` with the name of the current Verilog module.

6.2 Pre-Synthesis

Below is the pre-synthesis version of the code. This version will run fine in a Verilog simulator, however it will not compiled because it uses; synchronous assignment, assigns multiple variables from different always blocks on different signal edges, and assigns the same variable multiple times (synchronously) within the execution of a single always block.

6.2.1 Top Module

```
// define testbench module for input interface + engine_key_generator
module aes_engine(
  // -- input interface
  input clk, input rst_,
  input[7:0] din,
  input[1:0] cmd,
  output interface_ready,

  // -- output interface
  output[7:0] dout,
  output data_ok
);

// control signals
wire key_start, transformer_start_w,  transformer_done_w, output_read_w;

wire [127:0] plain, key, cipher;

wire[127:0] round0_key_w; // pre-round key
wire[127:0] round1_key_w;
wire[127:0] round2_key_w;
wire[127:0] round3_key_w;
wire[127:0] round4_key_w;
wire[127:0] round5_key_w;
wire[127:0] round6_key_w;
wire[127:0] round7_key_w;
wire[127:0] round8_key_w;
wire[127:0] round9_key_w;
wire[127:0] round10_key_w;

input_interface input_module (
  .clk            (clk),
  .rst_           (rst_),

  // inputs
  .din            (din),
  .cmd            (cmd),
  .transformer_done    (transformer_done_w),

  // outputs
  .key_start     (key_start),
  .plain_out      (plain),
  .key_out        (key),
```

```
  .ready          (interface_ready)
);

engine_key_generator key_gen_module (
  .rst_               (rst_),
  .clk        (clk),

  // inputs
  .key_in             (key),
  .key_start       (key_start), //rename input wire to key_gen_start

  // outputs
  .transformer_start  (transformer_start_w),
  // -- keys
  .round0_key      (round0_key_w),
  .round1_key      (round1_key_w),
  .round2_key      (round2_key_w),
  .round3_key      (round3_key_w),
  .round4_key      (round4_key_w),
  .round5_key      (round5_key_w),
  .round6_key      (round6_key_w),
  .round7_key      (round7_key_w),
  .round8_key      (round8_key_w),
  .round9_key      (round9_key_w),
  .round10_key    (round10_key_w)
);

engine_round_transformer transformer_module (
  .rst_        (rst_),
  .clk        (clk),

  // inputs
  .plaintext      (plain),
  .transformer_start  (transformer_start_w),
  .output_read        (output_read_w),
  // -- keys
  .round0_key      (round0_key_w),
  .round1_key      (round1_key_w),
  .round2_key      (round2_key_w),
  .round3_key      (round3_key_w),
  .round4_key      (round4_key_w),
  .round5_key      (round5_key_w),
  .round6_key      (round6_key_w),
  .round7_key      (round7_key_w),
  .round8_key      (round8_key_w),
  .round9_key      (round9_key_w),
  .round10_key    (round10_key_w),

  // output
  .ciphertext      (cipher),
  .transformer_done  (transformer_done_w)
);

output_interface output_module (
  .rst_        (rst_),
  .clk        (clk),

  // inputs
```

```
  .transformer_done  (transformer_done_w),
  .ciphertext      (cipher),

  // outputs
  .data_out      (dout),
  .data_ok      (data_ok),
  .output_read    (output_read_w)
);

`define TOPMODULE
// the "macro" to dump signals
initial begin
$dumpfile ("simulation/aes_engine.vcd");
$dumpvars(0, aes_engine);
end

endmodule
```

6.2.2 Input Interface

```
// Define module IO
module input_interface (
  // -- input interface
  input clk, input rst_,
  input[7:0] din,
  input[1:0] cmd,
  output ready,
  input transformer_done,
  // -- to engine
  output key_start,
  output[127:0] plain_out,
  output[127:0] key_out
);

// Define FSM states
localparam S_ID = 2'b00;  // Idle state
localparam S_SP = 2'b01;  // Set plaintext state
localparam S_SK = 2'b10;  // Set key state
localparam S_ST = 2'b11;  // Start encryption

// Define registers
// -- FSM state registers
reg[1:0] state, next_state;
initial state=S_ID;
// -- plaintext and key registers
reg[127:0] plain, key;
initial plain = 128'hF00DBABEF00DBABEF00DBABEF00DBABE;
initial key = 128'hF00DBABEF00DBABEF00DBABEF00DBABE;
assign plain_out = plain;
assign key_out = key;
// -- serial input counter registers
reg[3:0] p_i, k_i;
initial p_i=0;
initial k_i=0;
// output key_start if plain and key ready and start cmd
assign key_start = (p_i == 4'hF) && (k_i == 4'hF) && (state == S_ST);
```

```
37  // input interface ready if state is idle
38  assign ready = (state == S_ID);
39
40  // Reset functions
41  task resetkey;
42    begin
43      key = 128'h00000000000000000000000000000000;
44      k_i = 4'h0;
45    end
46  endtask
47  task resetplain;
48    begin
49      plain = 128'h00000000000000000000000000000000;
50      p_i = 4'h0;
51    end
52  endtask
53
54  // Next state combinational logic
55  always @(negedge clk) state <= next_state;
56  always @(posedge clk)
57  begin:next_state_decode
58    // reset logic
59    if (!rst_) begin
60      resetplain();
61      resetkey();
62      next_state = S_ID;
63    end
64    // command parse
65    else case (cmd)
66      // Set plaintext state
67      S_SP: begin
68        // reset if first input
69        if (state != S_SP) resetplain();
70        else p_i <= p_i+1;
71        // return to idle or keep state
72        if (p_i == 4'hF) next_state = S_ID; // recieved 16 bytes, return to idle
73        else
74        begin
75          // still reciving bytes
76          // read plain from din
77          plain = {plain[119:0], din}; // Left shift and insert
78          next_state = S_SP;
79        end
80      end
81
82      // Set key state
83      S_SK: begin
84        // reset if first input
85        if (state != S_SK) resetkey();
86        else k_i <= k_i+1;
87
88        // return to idle or keep state
89        if (k_i == 4'hF) next_state = S_ID; // recieved 16 bytes, return to idle
90        else
91        begin
92          // still recieving bytes
93          // read key from din
94          key = {key[119:0], din}; // Left shift and insert
```

```
        next_state = S_SK;
      end
    end

    // Start encryption state
    S_ST: begin
      // start engine
      if ((p_i == 4'hF) && (k_i == 4'hF)) next_state = S_ST;
      // return to idle
      //? this is a synchronous reset of plain text after encryption is complete
      //! reset only the plain text, in case we want to encrypt more plaintext
      ↪  with the same key
      if (transformer_done) begin
        // encyrption engine is done, we can reset
        resetplain();
        // resetkey();
        next_state = S_ID;
      end
      // otherwise wait in this state
      else next_state = S_ST;
    end

    // stay idle
    default: next_state = S_ID;
  endcase
end

```

6.2.3 Round Keys Generator

```
// Define module IO
module engine_key_generator (
  input rst_, clk,
  // input from input_interface
  input[127:0] key_in,
  input key_start,
  // output to round transformer
  output transformer_start, // start transformer signal
  output[127:0] round0_key, // pre-round key
  output[127:0] round1_key,
  output[127:0] round2_key,
  output[127:0] round3_key,
  output[127:0] round4_key,
  output[127:0] round5_key,
  output[127:0] round6_key,
  output[127:0] round7_key,
  output[127:0] round8_key,
  output[127:0] round9_key,
  output[127:0] round10_key
);

// Define registers
// -- round transformer start signal
reg transformer_start_r;
initial transformer_start_r = 0;
// initial transformer_start_r = 11'b000000000000;
```

```
assign transformer_start = transformer_start_r;
// -- keys array register
reg [127:0] round_keys[10:0];
assign round0_key = round_keys[0];
assign round1_key = round_keys[1];
assign round2_key = round_keys[2];
assign round3_key = round_keys[3];
assign round4_key = round_keys[4];
assign round5_key = round_keys[5];
assign round6_key = round_keys[6];
assign round7_key = round_keys[7];
assign round8_key = round_keys[8];
assign round9_key = round_keys[9];
assign round10_key = round_keys[10];

// Main Logic
// -- Asynchronous reset logic
always @(negedge rst_) begin
  reset_round_keys();
end
// -- Engine start logic
//? maybe we can do the key gen per round and the encryption rounds in parallel
// ---- loop counter
reg [5:0] i; // max 64 counts
initial i = 6'b000000;
// ---- key byte array for calculations
reg [31:0] w[43:0];
// ---- temp word for each word calculation
reg [31:0] tempword;

always @(posedge clk) begin
  // engine start cmd issued
  if (key_start) begin

    if (i == 0) begin
      // for loop iteration 0, computes first 4 keys

      // set pre-round key
      w[0] = key_in[127:96];
      w[1] = key_in[95:64];
      w[2] = key_in[63:32];
      w[3] = key_in[31:0];
      // -- copy to output register
      round_keys[0] = key_in;
      $display("Pre-Round Key:");
      $write("%02X %02X %02X %02X\n", round_keys[0][127:120],
      ↪  round_keys[0][95:88], round_keys[0][63:56], round_keys[0][31:24]);
      $write("%02X %02X %02X %02X\n", round_keys[0][119:112],
      ↪  round_keys[0][87:80], round_keys[0][55:48], round_keys[0][23:16]);
      $write("%02X %02X %02X %02X\n", round_keys[0][111:104],
      ↪  round_keys[0][79:72], round_keys[0][47:40], round_keys[0][15:8]);
      $write("%02X %02X %02X %02X\n", round_keys[0][103:96],
      ↪  round_keys[0][71:64], round_keys[0][39:32], round_keys[0][7:0]);

      // update for loop with offset since this iteration counts as 4 iterations
      i = i + 3;
    end

```

```
81      if (i > 3 && i < 44) begin
82        // for loop iterations 4-43
83
84        //// for (i=4; i<44; i=i+1) begin
85          tempword = w[i-1];
86          if ((i % 4) == 0) begin
87            // every 4th round, we need the round mixing function
88            tempword = subword(rotword(tempword)) ^ round_constant(i/4);
89          end
90
91          w[i] = w[i-4] ^ tempword;
92
93          // $display(i/4, i%4);
94          if ((i % 4) == 3) begin
95            // round done, print round keys
96            $write("Round %0d Key:", (i/4));
97            $display("");
98            $write("%02X %02X %02X %02X\n", w[((i/4)*4)][31:24],
              ↪  w[((i/4)*4)+1][31:24], w[((i/4)*4)+2][31:24],
              ↪  w[((i/4)*4)+3][31:24]);
99            $write("%02X %02X %02X %02X\n", w[((i/4)*4)][23:16],
              ↪  w[((i/4)*4)+1][23:16], w[((i/4)*4)+2][23:16],
              ↪  w[((i/4)*4)+3][23:16]);
100           $write("%02X %02X %02X %02X\n", w[((i/4)*4)][15:8] ,
              ↪  w[((i/4)*4)+1][15:8] , w[((i/4)*4)+2][15:8] , w[((i/4)*4)+3][15:8]
              ↪  );
101           $write("%02X %02X %02X %02X\n", w[((i/4)*4)][7:0]  , w[((i/4)*4)+1][7:0]
              ↪  , w[((i/4)*4)+2][7:0]  , w[((i/4)*4)+3][7:0]  );
102           // copy to output registers
103           round_keys[i/4] = {w[i-3], w[i-2], w[i-1], w[i]};
104         end
105       //// end
106     end
107
108     if (i > 43) begin
109       // for loop iteration 44+ (last iteration)
110
111       $display("----------------");
112       // issue round transformer start
113       transformer_start_r = 1;
114     end
115
116     // update for loop i
117     if (i < 44) begin
118       i = i + 1;
119     end
120
121   end
122 end
123
124 // handling when new key is requested
125 always @(posedge key_start) begin
126   // ? if the key is the same as the last computed key, dont recompute
127   // ? just issue transformer start
128   if (i > 43 && round_keys[0] == key_in) begin
129     // keys already computed
130     i = 44;
131     $display("----------------");
```

```
132       $display("Same key requested, skipping computation.");
133     end
134     else if (i == 0 || i>43) begin
135       // new key requested
136       i = 0;
137       $display("----------------");
138       $write("Generating round keys for key %032X\n", key_in);
139     end
140   end
141   // when key generator is finished, reset the transformer_start signal
142   always @(negedge key_start) begin
143     transformer_start_r = 0;
144   end
145
146   // Define functions
147   // -- Reset Round Keys
148   task reset_round_keys;
149     begin:rst_keys
150       integer i;
151       //// for (i = 0; i < 11; i = i + 1) begin
152       ////     round_keys[i] = 0;
153       //// end
154       round_keys[0] = 0;
155       round_keys[1] = 0;
156       round_keys[2] = 0;
157       round_keys[3] = 0;
158       round_keys[4] = 0;
159       round_keys[5] = 0;
160       round_keys[6] = 0;
161       round_keys[7] = 0;
162       round_keys[8] = 0;
163       round_keys[9] = 0;
164       round_keys[10] = 0;
165       transformer_start_r = 0;
166     end
167   endtask
168   // -- Round Constant (RCon)
169   // ---- Function to get the round constant
170   function [31:0] round_constant(input integer round);
171   begin
172     case (round)
173       1: round_constant = 32'h01000000;
174       2: round_constant = 32'h02000000;
175       3: round_constant = 32'h04000000;
176       4: round_constant = 32'h08000000;
177       5: round_constant = 32'h10000000;
178       6: round_constant = 32'h20000000;
179       7: round_constant = 32'h40000000;
180       8: round_constant = 32'h80000000;
181       9: round_constant = 32'h1B000000;
182       10: round_constant = 32'h36000000;
183   //    11: round_constant = 32'h6C000000;
184   //    12: round_constant = 32'hD8000000;
185       default: round_constant = 32'h00000000;
186     endcase
187   end
188   endfunction
189   // -- Function for rotate words
```

```
function [31:0] rotword(input [31:0]word);
  begin
    // circular left shift of words
    // b0, b1, b2, b3 ==> b1, b2, b3, b0
    rotword = {word[23:0], word[31:24]};
  end
endfunction
// -- Function for substitute words
function [31:0] subword(input [31:0]word);
  begin
    // substitute words using AES SBOX
    subword = {aes_sbox(word[31:24]), aes_sbox(word[23:16]),
    ↪  aes_sbox(word[15:8]), aes_sbox(word[7:0])};
  end
endfunction

// AES BOX Substitutions
// -- Function for AES SBOX
function [7:0] aes_sbox(input [7:0]in);
  begin
  case(in)    // synopsys full_case parallel_case
     8'h00: aes_sbox=8'h63;
```

```
     8'hff: aes_sbox=8'h16;
  endcase
  end
endfunction
```

6.2.4 Round Transformer

```
// Define module IO
module engine_round_transformer (
  input rst_, clk,
  // input from input_interface
  input[127:0] plaintext,
  // input from engine_key_generator
  input transformer_start,
  input[127:0] round0_key, // pre-round key
  input[127:0] round1_key,
  input[127:0] round2_key,
  input[127:0] round3_key,
  input[127:0] round4_key,
  input[127:0] round5_key,
  input[127:0] round6_key,
  input[127:0] round7_key,
  input[127:0] round8_key,
  input[127:0] round9_key,
  input[127:0] round10_key,
  // input from output_interface
  input output_read,
  // output to output_interface
  output[127:0] ciphertext,
  // control signal output to engine_key_generator, input_interface,
  ↪  output_interface
  output transformer_done
);
```

```
26
27  // Define registers
28  // -- round transformer done signal
29  reg transformer_done_r;
30  initial transformer_done_r = 0;
31  assign transformer_done = transformer_done_r;
32  // -- round transformer key array
33  reg [127:0] round_keys[10:0];
34  initial resetkeys;
35  // -- round transformer working block register
36  reg [127:0] state_block;
37  initial state_block = 128'h00000000000000000000000000000000;
38  // -- loop counter
39  reg [3:0] i; // max 15 counts
40  initial i = 4'h0;
41  // -- round trasnformer ciphertext register
42  reg [127:0] ciphertext_r;
43  initial ciphertext_r = 128'h00000000000000000000000000000000;
44  assign ciphertext = ciphertext_r;
45
46  // Main Logic
47  // -- Asynchronous reset logic
48  always @(negedge rst_ or posedge output_read) begin
49    resetcipher;
50    resetkeys;
51    state_block = 128'h00000000000000000000000000000000;
52    transformer_done_r = 0;
53    i = 0;
54  end
55  // -- Transformer start logic
56  always @(posedge clk) begin
57    // transformer start cmd issued
58    if (transformer_start) begin
59
60      if (i == 0) begin
61        // for loop pre-iteration
62
63        transformer_done_r = 0;
64        // read plaintext
65        state_block = plaintext;
66        $display("Plaintext:");
67        print_matrix(state_block);
68        // read round keys
69        round_keys[0] = round0_key;
70        round_keys[1] = round1_key;
71        round_keys[2] = round2_key;
72        round_keys[3] = round3_key;
73        round_keys[4] = round4_key;
74        round_keys[5] = round5_key;
75        round_keys[6] = round6_key;
76        round_keys[7] = round7_key;
77        round_keys[8] = round8_key;
78        round_keys[9] = round9_key;
79        round_keys[10] = round10_key;
80
81        // pre-round key
82        state_block = state_block ^ round_keys[0];
83        $display("Pre-Round State:");
```

```
84        print_matrix(state_block);
85      end
86
87      if ( i > 0 && i < 10) begin
88        // for loop iterations 1-9
89
90        // encryption rounds
91      //// for (i=1; i<10; i=i+1) begin
92        // Rijndael
93        // -- SubBytes
94        // state_block = SubBytes(state_block);
95        // actually, using TBOX already does SubBytes (aes_tbox(byte,1) ==
          ↪  aes_sbox(byte))
96        // -- ShiftRow
97        state_block = ShiftRow(state_block);
98        // -- MixCol
99        state_block = MixCol(state_block);
100       // -- Key XOR
101       state_block = state_block ^ round_keys[i];
102
103       // -- Print
104       $write("Round %0d State:", i);
105       $display("");
106       print_matrix(state_block);
107     //// end
108     end
109
110     if (i == 10) begin
111       // for loop iteration 10
112
113       // last round
114       // -- SubBytes
115       state_block = SubBytes(state_block);
116       // -- ShiftRow
117       state_block = ShiftRow(state_block);
118       // -- Key XOR
119       state_block = state_block ^ round_keys[10];
120       // -- Print
121       $display("Round 10 State / Ciphertext:");
122       print_matrix(state_block);
123     end
124
125     if (i == 10) begin
126       // for loop iteration 10 (last iteration)
127
128       // output ciphertext
129       ciphertext_r = state_block;
130       transformer_done_r = 1;
131     end
132
133     // if output is ready, but has not been read yet
134     // just idle; by not incrementing i
135     if (i > 10) begin
136       // for loop iteration 11+
137       // idle waiting for output_read to go high
138       // posedge check on output_read in code above
139     end
140     else begin
```

```
141       // 0 <= i <= 10, increment normally
142       // update for loop i
143       i = i + 1;
144     end
145 
146   end
147 end
148 
149 // Define functions
150 // -- Reset functions
151 task resetcipher;
152   begin
153     ciphertext_r = 128'h00000000000000000000000000000000;
154   end
155 endtask
156 task resetkeys;
157 begin
158   round_keys[0] =  128'h00000000000000000000000000000000;
159   round_keys[1] =  128'h00000000000000000000000000000000;
160   round_keys[2] =  128'h00000000000000000000000000000000;
161   round_keys[3] =  128'h00000000000000000000000000000000;
162   round_keys[4] =  128'h00000000000000000000000000000000;
163   round_keys[5] =  128'h00000000000000000000000000000000;
164   round_keys[6] =  128'h00000000000000000000000000000000;
165   round_keys[7] =  128'h00000000000000000000000000000000;
166   round_keys[8] =  128'h00000000000000000000000000000000;
167   round_keys[9] =  128'h00000000000000000000000000000000;
168   round_keys[10] = 128'h00000000000000000000000000000000;
169 end
170 endtask
171 // -- Print functions
172 // Function to print a 128-bit register as a 4x4 table of two hex digits in
    ↪  column-major order
173 task print_matrix(input [127:0] data);
174 begin
175   // Print the 4x4 table
176   $write("%02X %02X %02X %02X\n", data[127:120], data[95:88], data[63:56],
    ↪  data[31:24]);
177   $write("%02X %02X %02X %02X\n", data[119:112], data[87:80], data[55:48],
    ↪  data[23:16]);
178   $write("%02X %02X %02X %02X\n", data[111:104], data[79:72], data[47:40],
    ↪  data[15:8]);
179   $write("%02X %02X %02X %02X\n", data[103:96],  data[71:64], data[39:32],
    ↪  data[7:0]);
180   end
181 endtask
182 
183 // AES Functions
184 // -- AES SBOX SubBytes on state block
185 function [127:0] SubBytes(input [127:0] input_block);
186   //? since aes_tbox(byte,1) == aes_sbox(byte)
187   //? we could just do aes_tbox(byte,1)
188   //? to avoid importing aes_sbox
189   //? but actually, last round needs SubBytes without MixCol, so AES_SBOX still
    ↪  needed!
190   begin
191     SubBytes = {
192       aes_sbox(input_block[127:120]), aes_sbox(input_block[119:112]),
```

```
193       aes_sbox(input_block[111:104]), aes_sbox(input_block[103:96]),
194       aes_sbox(input_block[95:88]),   aes_sbox(input_block[87:80]),
195       aes_sbox(input_block[79:72]),   aes_sbox(input_block[71:64]),
196       aes_sbox(input_block[63:56]),   aes_sbox(input_block[55:48]),
197       aes_sbox(input_block[47:40]),   aes_sbox(input_block[39:32]),
198       aes_sbox(input_block[31:24]),   aes_sbox(input_block[23:16]),
199       aes_sbox(input_block[15:8]),    aes_sbox(input_block[7:0])
200     };
201   end
202 endfunction
203 // -- AES ShiftRow on state block
204 function [127:0] ShiftRow(input [127:0] input_block);
205   reg [127:0]roworder;
206   reg [127:0]shiftedroworder;
207   begin
208     //! rotword on logical rows! not collums
209     // column order to row order
210     roworder = {
211       input_block[127:120], input_block[95:88], input_block[63:56],
          ↪ input_block[31:24],
212       input_block[119:112], input_block[87:80], input_block[55:48],
          ↪ input_block[23:16],
213       input_block[111:104], input_block[79:72], input_block[47:40],
          ↪ input_block[15:8],
214       input_block[103:96],  input_block[71:64], input_block[39:32],
          ↪ input_block[7:0]
215     };
216     // circular left shift
217     shiftedroworder = {
218       roworder[127:96],                          // First row (unchanged)
219       rotword(roworder[95:64]),                  // Circular left-shift the second
          ↪ row by 1
220       rotword(rotword(roworder[63:32])),         // Circular left-shift the third
          ↪ row  by 2
221       rotword(rotword(rotword(roworder[31:0])))  // Circular left-shift the fourth
          ↪ row by 3
222     };
223     // row order back to column order
224     ShiftRow = {
225       shiftedroworder[127:120], shiftedroworder[95:88], shiftedroworder[63:56],
          ↪ shiftedroworder[31:24],
226       shiftedroworder[119:112], shiftedroworder[87:80], shiftedroworder[55:48],
          ↪ shiftedroworder[23:16],
227       shiftedroworder[111:104], shiftedroworder[79:72], shiftedroworder[47:40],
          ↪ shiftedroworder[15:8],
228       shiftedroworder[103:96],  shiftedroworder[71:64], shiftedroworder[39:32],
          ↪ shiftedroworder[7:0]
229     };
230   end
231 endfunction
232 // -- Function for rotate words
233 function [31:0] rotword(input [31:0]word);
234   begin
235     // circular left shift of words
236     // b0, b1, b2, b3 ==> b1, b2, b3, b0
237     rotword = {word[23:0], word[31:24]};
238   end
239 endfunction
```

```
240 // -- AES MixCol on state block
241 function [127:0] MixCol(input [127:0] input_block);
242   begin
243     // IMPLEMENTATION USING AES_TBOX
244     MixCol = {
245       // if column = {a, b, c, d}
246       // then AES TBOX is
247       // a = TBOX(a)[02] ^ TBOX(b)[03] ^ TBOX(c)[01] ^ TBOX(d)[01]
248       // b = TBOX(a)[01] ^ TBOX(b)[02] ^ TBOX(c)[03] ^ TBOX(d)[01]
249       // c = TBOX(a)[01] ^ TBOX(b)[01] ^ TBOX(c)[02] ^ TBOX(d)[03]
250       // d = TBOX(a)[03] ^ TBOX(b)[01] ^ TBOX(c)[01] ^ TBOX(d)[02]
251       // using our TBOX implementation
252       // a = aes_tbox(a,2) ^ aes_tbox(b,3) ^ aes_tbox(c,1) ^ aes_tbox(d,1)
253       // b = aes_tbox(a,1) ^ aes_tbox(b,2) ^ aes_tbox(c,3) ^ aes_tbox(d,1)
254       // c = aes_tbox(a,1) ^ aes_tbox(b,1) ^ aes_tbox(c,2) ^ aes_tbox(d,3)
255       // d = aes_tbox(a,3) ^ aes_tbox(b,1) ^ aes_tbox(c,1) ^ aes_tbox(d,2)
256       // but our {a,b,c,d} columns are:
257       // Column 1: a=input_block[127:120], b=input_block[119:112],
          ↪  c=input_block[111:104], d=input_block[103:96]
258       // Column 2: a=input_block[95:88],   b=input_block[87:80],
          ↪  c=input_block[79:72],   d=input_block[71:64]
259       // Column 3: a=input_block[63:56],   b=input_block[55:48],
          ↪  c=input_block[47:40],   d=input_block[39:32]
260       // Column 4: a=input_block[31:24],   b=input_block[23:16],
          ↪  c=input_block[15:8],    d=input_block[7:0]
261       // so we have to substitute each a,b,c,d value for each column with the
          ↪  index references above:
262
263
264       // Column 1
265       {
266         /*CM Row 1*/
267         {aes_tbox(input_block[127:120], 2) ^ aes_tbox(input_block[119:112], 3) ^
            ↪  aes_tbox(input_block[111:104], 1) ^ aes_tbox(input_block[103:96],
            ↪  1)},
268         /*CM Row 2*/
269         {aes_tbox(input_block[127:120], 1) ^ aes_tbox(input_block[119:112], 2) ^
            ↪  aes_tbox(input_block[111:104], 3) ^ aes_tbox(input_block[103:96],
            ↪  1)},
270         /*CM Row3*/
271         {aes_tbox(input_block[127:120], 1) ^ aes_tbox(input_block[119:112], 1) ^
            ↪  aes_tbox(input_block[111:104], 2) ^ aes_tbox(input_block[103:96],
            ↪  3)},
272         /*CM Row4*/
273         {aes_tbox(input_block[127:120], 3) ^ aes_tbox(input_block[119:112], 1) ^
            ↪  aes_tbox(input_block[111:104], 1) ^ aes_tbox(input_block[103:96], 2)}
274       },
275       // Column 2
276       {
277         /*CM Row 1*/
278         {aes_tbox(input_block[95:88], 2) ^ aes_tbox(input_block[87:80], 3) ^
            ↪  aes_tbox(input_block[79:72], 1) ^ aes_tbox(input_block[71:64], 1)},
279         /*CM Row 2*/
280         {aes_tbox(input_block[95:88], 1) ^ aes_tbox(input_block[87:80], 2) ^
            ↪  aes_tbox(input_block[79:72], 3) ^ aes_tbox(input_block[71:64], 1)},
281         /*CM Row3*/
282         {aes_tbox(input_block[95:88], 1) ^ aes_tbox(input_block[87:80], 1) ^
            ↪  aes_tbox(input_block[79:72], 2) ^ aes_tbox(input_block[71:64], 3)},
```

```
283         /*CM Row4*/
284         {aes_tbox(input_block[95:88], 3) ^ aes_tbox(input_block[87:80], 1) ^
            ↪  aes_tbox(input_block[79:72], 1) ^ aes_tbox(input_block[71:64], 2)}
285       },
286       // Column 3
287       {
288         /*CM Row 1*/
289         {aes_tbox(input_block[63:56], 2) ^ aes_tbox(input_block[55:48], 3) ^
            ↪  aes_tbox(input_block[47:40], 1) ^ aes_tbox(input_block[39:32], 1)},
290         /*CM Row 2*/
291         {aes_tbox(input_block[63:56], 1) ^ aes_tbox(input_block[55:48], 2) ^
            ↪  aes_tbox(input_block[47:40], 3) ^ aes_tbox(input_block[39:32], 1)},
292         /*CM Row3*/
293         {aes_tbox(input_block[63:56], 1) ^ aes_tbox(input_block[55:48], 1) ^
            ↪  aes_tbox(input_block[47:40], 2) ^ aes_tbox(input_block[39:32], 3)},
294         /*CM Row4*/
295         {aes_tbox(input_block[63:56], 3) ^ aes_tbox(input_block[55:48], 1) ^
            ↪  aes_tbox(input_block[47:40], 1) ^ aes_tbox(input_block[39:32], 2)}
296       },
297       // Column 4
298       {
299         /*CM Row 1*/
300         {aes_tbox(input_block[31:24], 2) ^ aes_tbox(input_block[23:16], 3) ^
            ↪  aes_tbox(input_block[15:8], 1) ^ aes_tbox(input_block[7:0], 1)},
301         /*CM Row 2*/
302         {aes_tbox(input_block[31:24], 1) ^ aes_tbox(input_block[23:16], 2) ^
            ↪  aes_tbox(input_block[15:8], 3) ^ aes_tbox(input_block[7:0], 1)},
303         /*CM Row3*/
304         {aes_tbox(input_block[31:24], 1) ^ aes_tbox(input_block[23:16], 1) ^
            ↪  aes_tbox(input_block[15:8], 2) ^ aes_tbox(input_block[7:0], 3)},
305         /*CM Row4*/
306         {aes_tbox(input_block[31:24], 3) ^ aes_tbox(input_block[23:16], 1) ^
            ↪  aes_tbox(input_block[15:8], 1) ^ aes_tbox(input_block[7:0], 2)}
307       }
308     };
309
310   end
311 endfunction
312
313 // AES BOX Substitutions
314 // -- Function for AES SBOX
315 function [7:0] aes_sbox(input [7:0]in);
316   begin
317   case(in)    // synopsys full_case parallel_case
318     8'h00: aes_sbox=8'h63;
```

```
573     8'hff: aes_sbox=8'h16;
574   endcase
575   end
576 endfunction
577 // -- Function for AES SBOX
578 function [7:0] aes_tbox(input  [7:0]in, input integer mult);
579   // full_tbox maps to {Srd, 2*Srd, 3*Srd}
580   reg [23:0] full_tbox;
581   begin
582   case(in)    // synopsys full_case parallel_case
583     8'h00: full_tbox=24'b101001011100011001100011;
```

```
    8'hff: full_tbox=24'b001110100010110000010110;
  endcase
  // split for required mult
  // Constant matrix:
  // 02 03 01 01
  // 01 02 03 01
  // 01 01 02 03
  // 03 01 01 02
  // where, for AES TBOX
  // 03 = [23:16]
  // 02 = [15:8]
  // 01 = [7:0]
  case (mult)
    3: aes_tbox = full_tbox[23:16];
    2: aes_tbox = full_tbox[15:8];
    1: aes_tbox = full_tbox[7:0];
  endcase
  end
endfunction
```

6.2.5 Output Interface

```
// Define module IO
module output_interface(
  input rst_, clk,
  // input from engine_round_transformer
  input transformer_done,
  input[127:0] ciphertext,
  // outputs
  output[7:0] data_out,
  output data_ok,
  output output_read
);

// Define registers
// -- ciphertext hold register
reg [127:0] output_hold;
initial output_hold = 128'h00000000000000000000000000000000;
// -- output register
reg [7:0] output_port;
initial output_port = 8'h00;
assign data_out = output_port;
// -- output control signals
reg data_ok_r;
initial data_ok_r = 0;
assign data_ok = data_ok_r;
reg output_read_r;
initial output_read_r = 0;
assign output_read = output_read_r;
// -- internal control signals
reg[4:0] i;
initial i = 5'b00000;

// Main Logic
// -- Asynchronous reset logic
```

```
always @(negedge rst_) begin
  output_hold = 128'h00000000000000000000000000000000;
  output_port = 8'h00;
  i = 5'b00000;
end
// -- Output latch logic
always @(posedge transformer_done) begin
  // ? add check here to see if output_read is finished
  // ? not needed; it takes 8 clock cycles to output the ciphertext
  // ? and 8 clock cycles to input the new plaintext
  // ? therefore these two tasks may be done independently and simultaneously
  // ? without any conflicts
  // copy ciphertext
  output_hold = ciphertext;
  // set in to begin
  i = 1;
end
always @(posedge clk) begin
  if (i == 0) begin
    // idling
    output_read_r = 0;
  end
  if (i == 1) begin
    // read first byte and set output data ok
    output_port = output_hold[127:120];
    data_ok_r = 1;
  end
  else if (i <= 16) begin
    // sequentially shift and output bytes 2-16
    output_hold = output_hold << 8;
    output_port = output_hold[127:120];
  end
  else if (i == 17) begin
    // finished outputing reset
    data_ok_r = 0;
    output_hold = 128'h00000000000000000000000000000000;
    output_port = 8'h00;
    i = 5'b00000;
    output_read_r = 1;
  end
  // increment i counter to next byte
  if (i >= 1) begin
    i = i + 1;
  end
end
```

6.3 Synthesis Ready

Below is the synthesis ready version of the code. Many modules have been edited to; use asynchronous assigns (<=), only assign one variable per execution of an `always @(posedge clk)` through the use of sub-loop counters (usually `i_i` and nested if statements), as well as only assigning variables in a single `always @(posedge clk)` block (no `always` on edges of any other signals).

6.3.1 Top Module

```
// define testbench module for input interface + engine_key_generator
module aes_engine(
  // -- input interface
  input clk, input rst_,
  input[7:0] din,
  input[1:0] cmd,
  output interface_ready,

  // -- output interface
  output[7:0] dout,
  output data_ok
);

// control signals
wire key_start, transformer_start_w,  transformer_done_w, output_read_w;

wire [127:0] plain, key, cipher;

wire[127:0] round0_key_w; // pre-round key
wire[127:0] round1_key_w;
wire[127:0] round2_key_w;
wire[127:0] round3_key_w;
wire[127:0] round4_key_w;
wire[127:0] round5_key_w;
wire[127:0] round6_key_w;
wire[127:0] round7_key_w;
wire[127:0] round8_key_w;
wire[127:0] round9_key_w;
wire[127:0] round10_key_w;

input_interface input_module (
  .clk            (clk),
  .rst_           (rst_),

  // inputs
  .din            (din),
  .cmd            (cmd),
  .transformer_done    (transformer_done_w),

  // outputs
  .key_start     (key_start),
  .plain_out      (plain),
  .key_out        (key),
  .ready          (interface_ready)
);
```

```
46
47  engine_key_generator key_gen_module (
48    .rst_               (rst_),
49    .clk        (clk),
50
51    // inputs
52    .key_in             (key),
53    .key_start       (key_start), //rename input wire to key_gen_start
54
55    // outputs
56    .transformer_start  (transformer_start_w),
57    // -- keys
58    .round0_key      (round0_key_w),
59    .round1_key      (round1_key_w),
60    .round2_key      (round2_key_w),
61    .round3_key      (round3_key_w),
62    .round4_key      (round4_key_w),
63    .round5_key      (round5_key_w),
64    .round6_key      (round6_key_w),
65    .round7_key      (round7_key_w),
66    .round8_key      (round8_key_w),
67    .round9_key      (round9_key_w),
68    .round10_key    (round10_key_w)
69  );
70
71  engine_round_transformer transformer_module (
72    .rst_        (rst_),
73    .clk        (clk),
74
75    // inputs
76    .plaintext      (plain),
77    .transformer_start  (transformer_start_w),
78    .output_read        (output_read_w),
79    // -- keys
80    .round0_key      (round0_key_w),
81    .round1_key      (round1_key_w),
82    .round2_key      (round2_key_w),
83    .round3_key      (round3_key_w),
84    .round4_key      (round4_key_w),
85    .round5_key      (round5_key_w),
86    .round6_key      (round6_key_w),
87    .round7_key      (round7_key_w),
88    .round8_key      (round8_key_w),
89    .round9_key      (round9_key_w),
90    .round10_key    (round10_key_w),
91
92    // output
93    .ciphertext      (cipher),
94    .transformer_done  (transformer_done_w)
95  );
96
97  output_interface output_module (
98    .rst_        (rst_),
99    .clk        (clk),
100
101   // inputs
102   .transformer_done  (transformer_done_w),
103   .ciphertext      (cipher),
```

```
104
105   // outputs
106   .data_out       (dout),
107   .data_ok       (data_ok),
108   .output_read     (output_read_w)
109 );
110
111 endmodule
```

6.3.2 Input Interface

```
1   // Define module IO
2   module input_interface (
3     // -- input interface
4     input clk, input rst_,
5     input[7:0] din,
6     input[1:0] cmd,
7     output ready,
8     input transformer_done,
9     // -- to engine
10    output key_start,
11    output[127:0] plain_out,
12    output[127:0] key_out
13  );
14
15  // Define FSM states
16  localparam S_ID = 2'b00;  // Idle state
17  localparam S_SP = 2'b01;  // Set plaintext state
18  localparam S_SK = 2'b10;  // Set key state
19  localparam S_ST = 2'b11;  // Start encryption
20
21  // Define registers
22  // -- FSM state registers
23  reg[1:0] state, next_state;
24  initial state=S_ID;
25  // -- plaintext and key registers
26  reg[127:0] plain, key;
27  initial plain = 128'hF00DBABEF00DBABEF00DBABEF00DBABE;
28  initial key = 128'hF00DBABEF00DBABEF00DBABEF00DBABE;
29  assign plain_out = plain;
30  assign key_out = key;
31  // -- serial input counter registers
32  reg[3:0] p_i, k_i;
33  initial p_i=0;
34  initial k_i=0;
35  // output key_start if plain and key ready and start cmd
36  assign key_start = (p_i == 4'hF) && (k_i == 4'hF) && (state == S_ST);
37  // input interface ready if state is idle
38  assign ready = (state == S_ID);
39
40  // Reset functions
41  task resetkey;
42    begin
43      key <= 128'h00000000000000000000000000000000;
44      k_i <= 4'h0;
45    end
```

```
endtask
task resetplain;
  begin
    plain <= 128'h00000000000000000000000000000000;
    p_i <= 4'h0;
  end
endtask

// Next state combinational logic
always @(negedge clk) state <= next_state;
always @(posedge clk)
begin:next_state_decode
  // reset logic
  if (!rst_) begin
    resetplain();
    resetkey();
    next_state <= S_ID;
  end
  // command parse
  else case (cmd)
    // Set plaintext state
    S_SP: begin
      // reset if first input
      if (state != S_SP) resetplain();
      else p_i <= p_i+1;
      // return to idle or keep state
      if (p_i == 4'hF) next_state <= S_ID; // recieved 16 bytes, return to idle
      else
      begin
        // still reciving bytes
        // read plain from din
        plain <= {plain[119:0], din}; // Left shift and insert
        next_state <= S_SP;
      end
    end

    // Set key state
    S_SK: begin
      // reset if first input
      if (state != S_SK) resetkey();
      else k_i <= k_i+1;

      // return to idle or keep state
      if (k_i == 4'hF) next_state <= S_ID; // recieved 16 bytes, return to idle
      else
      begin
        // still recieving bytes
        // read key from din
        key <= {key[119:0], din}; // Left shift and insert
        next_state <= S_SK;
      end
    end

    // Start encryption state
    S_ST: begin
      // start engine
      if ((p_i == 4'hF) && (k_i == 4'hF)) next_state <= S_ST;
      // return to idle
```

```
      //? this is a synchronous reset of plain text after encryption is complete
      //! reset only the plain text, in case we want to encrypt more plaintext
      ↪  with the same key
      if (transformer_done) begin
        // encyrption engine is done, we can reset
        resetplain();
        // resetkey();
        next_state <= S_ID;
      end
      // otherwise wait in this state
      else next_state <= S_ST;
    end

    // stay idle
    default: next_state <= S_ID;
  endcase
end

endmodule
```

6.3.3 Round Keys Generator

```
// Define module IO
module engine_key_generator (
  input rst_, clk,
  // input from input_interface
  input[127:0] key_in,
  input key_start,
  // output to round transformer
  output transformer_start, // start transformer signal
  output[127:0] round0_key, // pre-round key
  output[127:0] round1_key,
  output[127:0] round2_key,
  output[127:0] round3_key,
  output[127:0] round4_key,
  output[127:0] round5_key,
  output[127:0] round6_key,
  output[127:0] round7_key,
  output[127:0] round8_key,
  output[127:0] round9_key,
  output[127:0] round10_key
);

// Define registers
// -- round transformer start signal
reg transformer_start_r;
initial transformer_start_r = 0;
// initial transformer_start_r = 11'b000000000000;
assign transformer_start = transformer_start_r;
// -- keys array register
reg [127:0] round_keys[10:0];
assign round0_key = round_keys[0];
assign round1_key = round_keys[1];
assign round2_key = round_keys[2];
assign round3_key = round_keys[3];
assign round4_key = round_keys[4];
```

```
35  assign round5_key = round_keys[5];
36  assign round6_key = round_keys[6];
37  assign round7_key = round_keys[7];
38  assign round8_key = round_keys[8];
39  assign round9_key = round_keys[9];
40  assign round10_key = round_keys[10];
41
42  // Main Logic
43  // -- Engine start logic
44  //? maybe we can do the key gen per round and the encryption rounds in parallel
45  // ---- loop counter
46  reg [5:0] i; // max 64 counts
47  initial i = 6'b000000;
48  // -- sub-loop counter
49  reg [1:0] i_i;
50  initial i_i = 2'b00;
51  // ---- key byte array for calculations
52  reg [31:0] w[43:0];
53  // ---- temp word for each word calculation
54  reg [31:0] tempword;
55
56  reg edgedetect_keystart;
57  initial edgedetect_keystart = 0;
58
59  always @(posedge clk) begin
60    // reset
61    if (!rst_) begin
62      reset_round_keys();
63    end
64    // on negedge key_start
65    if (!key_start && edgedetect_keystart) begin
66      // when key generator is finished, reset the transformer_start signal
67      transformer_start_r <= 0;
68    end
69    else begin
70      // on posedge key_start
71      // handling when new key is requested
72      if (key_start && !edgedetect_keystart) begin
73        // ? if the key is the same as the last computed key, dont recompute
74        // ? just issue transformer start
75        if (i > 43 && round_keys[0] == key_in) begin
76          // keys already computed
77          i <= 44;
78          $display("-----------------");
79          $display("Same key requested, skipping computation.");
80        end
81        else if (i == 0 || i>43) begin
82          // new key requested
83          i <= 0;
84          $display("-----------------");
85          $write("Generating round keys for key %032X\n", key_in);
86        end
87      end
88      // normal clock
89      // engine start cmd issued
90      if (key_start) begin
91
92        if (i == 0) begin
```

```
93          // for loop iteration 0, computes first 4 keys
94
95          // set pre-round key
96          w[0] <= key_in[127:96];
97          w[1] <= key_in[95:64];
98          w[2] <= key_in[63:32];
99          w[3] <= key_in[31:0];
100         // -- copy to output register
101         round_keys[0] <= key_in;
102
103         // update for loop with offset since this iteration counts as 4 iterations
104         i <= i + 3;
105       end
106
107       else if (i == 3) begin
108         $display("Pre-Round Key:");
109         $write("%02X %02X %02X %02X\n", round_keys[0][127:120],
          ↪  round_keys[0][95:88], round_keys[0][63:56], round_keys[0][31:24]);
110         $write("%02X %02X %02X %02X\n", round_keys[0][119:112],
          ↪  round_keys[0][87:80], round_keys[0][55:48], round_keys[0][23:16]);
111         $write("%02X %02X %02X %02X\n", round_keys[0][111:104],
          ↪  round_keys[0][79:72], round_keys[0][47:40], round_keys[0][15:8]);
112         $write("%02X %02X %02X %02X\n", round_keys[0][103:96],
          ↪  round_keys[0][71:64], round_keys[0][39:32], round_keys[0][7:0]);
113         i <= i + 1;
114       end
115
116       else if (i > 3 && i < 44) begin
117         // for loop iterations 4-43
118
119         //// for (i=4; i<44; i=i+1) begin
120           // sub-loop step 0
121           if (i_i == 0) begin
122             tempword <= w[i-1];
123             i_i <= i_i + 1;
124           end
125           // sub-loop step 1
126           else if (i_i == 1) begin
127             if ((i % 4) == 0) begin
128               // every 4th round, we need the round mixing function
129               tempword <= subword(rotword(tempword)) ^ round_constant(i/4);
130             end
131             i_i <= i_i + 1;
132           end
133           // sub-loop step 2
134           else if (i_i == 2) begin
135             w[i] <= w[i-4] ^ tempword;
136             i_i <= i_i + 1;
137           end
138           // sub-loop step 3
139           else if (i_i == 3) begin
140             // $display(i/4, i%4);
141             if ((i % 4) == 3) begin
142               // round done, print round keys
143               $write("Round %0d Key:", (i/4));
144               $display("");
```

```
145                 $write("%02X %02X %02X %02X\n", w[((i/4)*4)][31:24],
                    ↪  w[((i/4)*4)+1][31:24], w[((i/4)*4)+2][31:24],
                    ↪  w[((i/4)*4)+3][31:24]);
146                 $write("%02X %02X %02X %02X\n", w[((i/4)*4)][23:16],
                    ↪  w[((i/4)*4)+1][23:16], w[((i/4)*4)+2][23:16],
                    ↪  w[((i/4)*4)+3][23:16]);
147                 $write("%02X %02X %02X %02X\n", w[((i/4)*4)][15:8] ,
                    ↪  w[((i/4)*4)+1][15:8] , w[((i/4)*4)+2][15:8] ,
                    ↪  w[((i/4)*4)+3][15:8] );
148                 $write("%02X %02X %02X %02X\n", w[((i/4)*4)][7:0]  ,
                    ↪  w[((i/4)*4)+1][7:0]  , w[((i/4)*4)+2][7:0]  ,
                    ↪  w[((i/4)*4)+3][7:0]  );
149                 // copy to output registers
150                 round_keys[i/4] <= {w[i-3], w[i-2], w[i-1], w[i]};
151               end
152               i_i <= 0;
153               i <= i + 1;
154             end
155           //// end
156         end
157
158         else if (i > 43) begin
159           // for loop iteration 44+ (last iteration)
160
161           $display("-----------------");
162           // issue round transformer start
163           transformer_start_r <= 1;
164           // i <= 0;
165         end
166
167         // // update for loop i
168         // if (i < 44) begin
169         //   i <= i + 1;
170         // end
171
172       end
173     end
174     // edge detector set
175     edgedetect_keystart <= key_start;
176   end
177
178   // Define functions
179   // -- Reset Round Keys
180   task reset_round_keys;
181     begin:rst_keys
182       integer i;
183       //// for (i = 0; i < 11; i = i + 1) begin
184       ////     round_keys[i] = 0;
185       //// end
186       round_keys[0] <= 0;
187       round_keys[1] <= 0;
188       round_keys[2] <= 0;
189       round_keys[3] <= 0;
190       round_keys[4] <= 0;
191       round_keys[5] <= 0;
192       round_keys[6] <= 0;
193       round_keys[7] <= 0;
194       round_keys[8] <= 0;
```

```
    round_keys[9] <= 0;
    round_keys[10] <= 0;
    transformer_start_r <= 0;
  end
endtask
// -- Round Constant (RCon)
// ---- Function to get the round constant
function [31:0] round_constant(input integer round);
begin
  case (round)
    1: round_constant = 32'h01000000;
    2: round_constant = 32'h02000000;
    3: round_constant = 32'h04000000;
    4: round_constant = 32'h08000000;
    5: round_constant = 32'h10000000;
    6: round_constant = 32'h20000000;
    7: round_constant = 32'h40000000;
    8: round_constant = 32'h80000000;
    9: round_constant = 32'h1B000000;
    10: round_constant = 32'h36000000;
//    11: round_constant = 32'h6C000000;
//    12: round_constant = 32'hD8000000;
    default: round_constant = 32'h00000000;
  endcase
end
endfunction
// -- Function for rotate words
function [31:0] rotword(input [31:0]word);
  begin
    // circular left shift of words
    // b0, b1, b2, b3 ==> b1, b2, b3, b0
    rotword = {word[23:0], word[31:24]};
  end
endfunction
// -- Function for substitute words
function [31:0] subword(input [31:0]word);
  begin
    // substitute words using AES SBOX
    subword = {aes_sbox(word[31:24]), aes_sbox(word[23:16]),
    ↪  aes_sbox(word[15:8]), aes_sbox(word[7:0])};
  end
endfunction

// AES BOX Substitutions
// -- Function for AES SBOX
function [7:0] aes_sbox(input [7:0]in);
  begin
  case(in)    // synopsys full_case parallel_case
     8'h00: aes_sbox=8'h63;
```

```
     8'hff: aes_sbox=8'h16;
  endcase
  end
endfunction

endmodule
```

6.3.4 Round Transformer

```
// Define module IO
module engine_round_transformer (
  input rst_, clk,
  // input from input_interface
  input[127:0] plaintext,
  // input from engine_key_generator
  input transformer_start,
  input[127:0] round0_key, // pre-round key
  input[127:0] round1_key,
  input[127:0] round2_key,
  input[127:0] round3_key,
  input[127:0] round4_key,
  input[127:0] round5_key,
  input[127:0] round6_key,
  input[127:0] round7_key,
  input[127:0] round8_key,
  input[127:0] round9_key,
  input[127:0] round10_key,
  // input from output_interface
  input output_read,
  // output to output_interface
  output[127:0] ciphertext,
  // control signal output to engine_key_generator, input_interface,
  ↪  output_interface
  output transformer_done
);

// Define registers
// -- round transformer done signal
reg transformer_done_r;
initial transformer_done_r = 0;
assign transformer_done = transformer_done_r;
// -- round transformer key array
reg [127:0] round_keys[10:0];
initial resetkeys;
// -- round transformer working block register
reg [127:0] state_block;
initial state_block = 128'h00000000000000000000000000000000;
// -- loop counter
reg [3:0] i; // max 15 counts
initial i = 4'h0;
// -- sub-loop counter
reg [1:0] i_i;
initial i_i = 2'b00;
// -- round trasnformer ciphertext register
reg [127:0] ciphertext_r;
initial ciphertext_r = 128'h00000000000000000000000000000000;
assign ciphertext = ciphertext_r;

// Main Logic
always @(posedge clk) begin
  // negedge rst_ or posedge output_read
  if (!rst_ || output_read) begin
    //reset logic
    resetcipher;
    resetkeys;
```

```
    state_block <= 128'h00000000000000000000000000000000;
    transformer_done_r <= 0;
    i <= 0;
  end
  else begin
  // normal clock
    // transformer start cmd issued
    if (transformer_start) begin

      if (i == 0) begin
        // for loop pre-iteration

        // sub-loop step 0
        if (i_i == 0) begin
          transformer_done_r <= 0;
          // read plaintext
          state_block <= plaintext;

          i_i <= i_i + 1;
        end
        // sub-loop step 1
        else if (i_i == 1) begin
          $display("Plaintext:");
          print_matrix(state_block);
          // read round keys
          round_keys[0] <= round0_key;
          round_keys[1] <= round1_key;
          round_keys[2] <= round2_key;
          round_keys[3] <= round3_key;
          round_keys[4] <= round4_key;
          round_keys[5] <= round5_key;
          round_keys[6] <= round6_key;
          round_keys[7] <= round7_key;
          round_keys[8] <= round8_key;
          round_keys[9] <= round9_key;
          round_keys[10] <= round10_key;

          i_i <= i_i + 1;
        end
        // sub-loop step 2
        else if (i_i == 2) begin
          // pre-round key
          state_block <= state_block ^ round_keys[0];

          i_i <= i_i + 1;
        end
        // sub-loop step 3
        else if (i_i == 3) begin
          $display("Pre-Round State:");
          print_matrix(state_block);
        end
      end

      if ( i > 0 && i < 10) begin
        // for loop iterations 1-9

        // encryption rounds
      //// for (i=1; i<10; i=i+1) begin
```

```
        // Rijndael
        // -- SubBytes
        // state_block <= SubBytes(state_block);
        // actually, using TBOX already does SubBytes (aes_tbox(byte,1) ==
        ↪  aes_sbox(byte))
        if (i_i == 0) begin
          // -- ShiftRow
          state_block <= ShiftRow(state_block);
          i_i <= i_i + 1;
        end
        else if (i_i == 1) begin
          // -- MixCol
          state_block <= MixCol(state_block);
          i_i <= i_i + 1;
        end
        else if (i_i == 2) begin
          // -- Key XOR
          state_block <= state_block ^ round_keys[i];
          i_i <= i_i + 1;
        end
        else if (i_i == 3) begin
          // -- Print
          $write("Round %0d State:", i);
          $display("");
          print_matrix(state_block);
        end
      //// end
      end

      if (i == 10) begin
        // for loop iteration 10 (last iteration)

        // last round
        if (i_i == 0) begin
          // -- SubBytes
          state_block <= SubBytes(state_block);
          i_i <= i_i + 1;
        end
        else if (i_i == 1) begin
          // -- ShiftRow
          state_block <= ShiftRow(state_block);
          i_i <= i_i + 1;
        end
        else if (i_i == 2) begin
          // -- Key XOR
          state_block <= state_block ^ round_keys[10];
          i_i <= i_i + 1;
        end
        else if (i_i == 3) begin
          // -- Print
          $display("Round 10 State / Ciphertext:");
          print_matrix(state_block);
          // output ciphertext
          ciphertext_r <= state_block;
          transformer_done_r <= 1;
        end
      end

```

```
171       // if output is ready, but has not been read yet
172       // just idle; by not incrementing i
173       if (i < 10 && i_i == 3) begin
174         // 0 <= i <= 10, increment normally
175         // update for loop i
176         i <= i + 1;
177         i_i <= 0;
178       end
179 
180     end
181   end
182 end
183 
184 // Define functions
185 // -- Reset functions
186 task resetcipher;
187   begin
188     ciphertext_r <= 128'h00000000000000000000000000000000;
189   end
190 endtask
191 task resetkeys;
192 begin
193   round_keys[0] <=  128'h00000000000000000000000000000000;
194   round_keys[1] <=  128'h00000000000000000000000000000000;
195   round_keys[2] <=  128'h00000000000000000000000000000000;
196   round_keys[3] <=  128'h00000000000000000000000000000000;
197   round_keys[4] <=  128'h00000000000000000000000000000000;
198   round_keys[5] <=  128'h00000000000000000000000000000000;
199   round_keys[6] <=  128'h00000000000000000000000000000000;
200   round_keys[7] <=  128'h00000000000000000000000000000000;
201   round_keys[8] <=  128'h00000000000000000000000000000000;
202   round_keys[9] <=  128'h00000000000000000000000000000000;
203   round_keys[10] <= 128'h00000000000000000000000000000000;
204 end
205 endtask
206 // -- Print functions
207 // Function to print a 128-bit register as a 4x4 table of two hex digits in
    ↪  column-major order
208 task print_matrix(input [127:0] data);
209 begin
210   // Print the 4x4 table
211   $write("%02X %02X %02X %02X\n", data[127:120], data[95:88], data[63:56],
    ↪  data[31:24]);
212   $write("%02X %02X %02X %02X\n", data[119:112], data[87:80], data[55:48],
    ↪  data[23:16]);
213   $write("%02X %02X %02X %02X\n", data[111:104], data[79:72], data[47:40],
    ↪  data[15:8]);
214   $write("%02X %02X %02X %02X\n", data[103:96],  data[71:64], data[39:32],
    ↪  data[7:0]);
215   end
216 endtask
217 
218 // AES Functions
219 // -- AES SBOX SubBytes on state block
220 function [127:0] SubBytes(input [127:0] input_block);
221   //? since aes_tbox(byte,1) == aes_sbox(byte)
222   //? we could just do aes_tbox(byte,1)
223   //? to avoid importing aes_sbox
```

```
224   //? but actually, last round needs SubBytes without MixCol, so AES_SBOX still
      ↪  needed!
225   begin
226     SubBytes = {
227       aes_sbox(input_block[127:120]), aes_sbox(input_block[119:112]),
228       aes_sbox(input_block[111:104]), aes_sbox(input_block[103:96]),
229       aes_sbox(input_block[95:88]),   aes_sbox(input_block[87:80]),
230       aes_sbox(input_block[79:72]),   aes_sbox(input_block[71:64]),
231       aes_sbox(input_block[63:56]),   aes_sbox(input_block[55:48]),
232       aes_sbox(input_block[47:40]),   aes_sbox(input_block[39:32]),
233       aes_sbox(input_block[31:24]),   aes_sbox(input_block[23:16]),
234       aes_sbox(input_block[15:8]),    aes_sbox(input_block[7:0])
235     };
236   end
237 endfunction
238 // -- AES ShiftRow on state block
239 function [127:0] ShiftRow(input [127:0] input_block);
240   reg [127:0]roworder;
241   reg [127:0]shiftedroworder;
242   begin
243     //! rotword on logical rows! not collums
244     // column order to row order
245     roworder = {
246       input_block[127:120], input_block[95:88], input_block[63:56],
      ↪  input_block[31:24],
247       input_block[119:112], input_block[87:80], input_block[55:48],
      ↪  input_block[23:16],
248       input_block[111:104], input_block[79:72], input_block[47:40],
      ↪  input_block[15:8],
249       input_block[103:96],  input_block[71:64], input_block[39:32],
      ↪  input_block[7:0]
250     };
251     // circular left shift
252     shiftedroworder = {
253       roworder[127:96],                          // First row (unchanged)
254       rotword(roworder[95:64]),                  // Circular left-shift the second
      ↪  row by 1
255       rotword(rotword(roworder[63:32])),         // Circular left-shift the third
      ↪  row  by 2
256       rotword(rotword(rotword(roworder[31:0])))  // Circular left-shift the fourth
      ↪  row by 3
257     };
258     // row order back to column order
259     ShiftRow = {
260       shiftedroworder[127:120], shiftedroworder[95:88], shiftedroworder[63:56],
      ↪  shiftedroworder[31:24],
261       shiftedroworder[119:112], shiftedroworder[87:80], shiftedroworder[55:48],
      ↪  shiftedroworder[23:16],
262       shiftedroworder[111:104], shiftedroworder[79:72], shiftedroworder[47:40],
      ↪  shiftedroworder[15:8],
263       shiftedroworder[103:96],  shiftedroworder[71:64], shiftedroworder[39:32],
      ↪  shiftedroworder[7:0]
264     };
265   end
266 endfunction
267 // -- Function for rotate words
268 function [31:0] rotword(input [31:0]word);
269   begin
```

```
270     // circular left shift of words
271     // b0, b1, b2, b3 ==> b1, b2, b3, b0
272     rotword = {word[23:0], word[31:24]};
273   end
274 endfunction
275 // -- AES MixCol on state block
276 function [127:0] MixCol(input [127:0] input_block);
277   begin
278     // IMPLEMENTATION USING AES_TBOX
279     MixCol = {
280       // if column = {a, b, c, d}
281       // then AES TBOX is
282       // a = TBOX(a)[02] ^ TBOX(b)[03] ^ TBOX(c)[01] ^ TBOX(d)[01]
283       // b = TBOX(a)[01] ^ TBOX(b)[02] ^ TBOX(c)[03] ^ TBOX(d)[01]
284       // c = TBOX(a)[01] ^ TBOX(b)[01] ^ TBOX(c)[02] ^ TBOX(d)[03]
285       // d = TBOX(a)[03] ^ TBOX(b)[01] ^ TBOX(c)[01] ^ TBOX(d)[02]
286       // using our TBOX implementation
287       // a = aes_tbox(a,2) ^ aes_tbox(b,3) ^ aes_tbox(c,1) ^ aes_tbox(d,1)
288       // b = aes_tbox(a,1) ^ aes_tbox(b,2) ^ aes_tbox(c,3) ^ aes_tbox(d,1)
289       // c = aes_tbox(a,1) ^ aes_tbox(b,1) ^ aes_tbox(c,2) ^ aes_tbox(d,3)
290       // d = aes_tbox(a,3) ^ aes_tbox(b,1) ^ aes_tbox(c,1) ^ aes_tbox(d,2)
291       // but our {a,b,c,d} columns are:
292       // Column 1: a=input_block[127:120], b=input_block[119:112],
          ↪  c=input_block[111:104], d=input_block[103:96]
293       // Column 2: a=input_block[95:88],   b=input_block[87:80],
          ↪  c=input_block[79:72],   d=input_block[71:64]
294       // Column 3: a=input_block[63:56],   b=input_block[55:48],
          ↪  c=input_block[47:40],   d=input_block[39:32]
295       // Column 4: a=input_block[31:24],   b=input_block[23:16],
          ↪  c=input_block[15:8],    d=input_block[7:0]
296       // so we have to substitute each a,b,c,d value for each column with the
          ↪  index references above:
297
298
299       // Column 1
300       {
301         /*CM Row 1*/
302         {aes_tbox(input_block[127:120], 2) ^ aes_tbox(input_block[119:112], 3) ^
          ↪  aes_tbox(input_block[111:104], 1) ^ aes_tbox(input_block[103:96],
          ↪  1)},
303         /*CM Row 2*/
304         {aes_tbox(input_block[127:120], 1) ^ aes_tbox(input_block[119:112], 2) ^
          ↪  aes_tbox(input_block[111:104], 3) ^ aes_tbox(input_block[103:96],
          ↪  1)},
305         /*CM Row3*/
306         {aes_tbox(input_block[127:120], 1) ^ aes_tbox(input_block[119:112], 1) ^
          ↪  aes_tbox(input_block[111:104], 2) ^ aes_tbox(input_block[103:96],
          ↪  3)},
307         /*CM Row4*/
308         {aes_tbox(input_block[127:120], 3) ^ aes_tbox(input_block[119:112], 1) ^
          ↪  aes_tbox(input_block[111:104], 1) ^ aes_tbox(input_block[103:96], 2)}
309       },
310       // Column 2
311       {
312         /*CM Row 1*/
313         {aes_tbox(input_block[95:88], 2) ^ aes_tbox(input_block[87:80], 3) ^
          ↪  aes_tbox(input_block[79:72], 1) ^ aes_tbox(input_block[71:64], 1)},
314         /*CM Row 2*/
```

```
315         {aes_tbox(input_block[95:88], 1) ^ aes_tbox(input_block[87:80], 2) ^
            ↪  aes_tbox(input_block[79:72], 3) ^ aes_tbox(input_block[71:64], 1)},
316         /*CM Row3*/
317         {aes_tbox(input_block[95:88], 1) ^ aes_tbox(input_block[87:80], 1) ^
            ↪  aes_tbox(input_block[79:72], 2) ^ aes_tbox(input_block[71:64], 3)},
318         /*CM Row4*/
319         {aes_tbox(input_block[95:88], 3) ^ aes_tbox(input_block[87:80], 1) ^
            ↪  aes_tbox(input_block[79:72], 1) ^ aes_tbox(input_block[71:64], 2)}
320       },
321       // Column 3
322       {
323         /*CM Row 1*/
324         {aes_tbox(input_block[63:56], 2) ^ aes_tbox(input_block[55:48], 3) ^
            ↪  aes_tbox(input_block[47:40], 1) ^ aes_tbox(input_block[39:32], 1)},
325         /*CM Row 2*/
326         {aes_tbox(input_block[63:56], 1) ^ aes_tbox(input_block[55:48], 2) ^
            ↪  aes_tbox(input_block[47:40], 3) ^ aes_tbox(input_block[39:32], 1)},
327         /*CM Row3*/
328         {aes_tbox(input_block[63:56], 1) ^ aes_tbox(input_block[55:48], 1) ^
            ↪  aes_tbox(input_block[47:40], 2) ^ aes_tbox(input_block[39:32], 3)},
329         /*CM Row4*/
330         {aes_tbox(input_block[63:56], 3) ^ aes_tbox(input_block[55:48], 1) ^
            ↪  aes_tbox(input_block[47:40], 1) ^ aes_tbox(input_block[39:32], 2)}
331       },
332       // Column 4
333       {
334         /*CM Row 1*/
335         {aes_tbox(input_block[31:24], 2) ^ aes_tbox(input_block[23:16], 3) ^
            ↪  aes_tbox(input_block[15:8], 1) ^ aes_tbox(input_block[7:0], 1)},
336         /*CM Row 2*/
337         {aes_tbox(input_block[31:24], 1) ^ aes_tbox(input_block[23:16], 2) ^
            ↪  aes_tbox(input_block[15:8], 3) ^ aes_tbox(input_block[7:0], 1)},
338         /*CM Row3*/
339         {aes_tbox(input_block[31:24], 1) ^ aes_tbox(input_block[23:16], 1) ^
            ↪  aes_tbox(input_block[15:8], 2) ^ aes_tbox(input_block[7:0], 3)},
340         /*CM Row4*/
341         {aes_tbox(input_block[31:24], 3) ^ aes_tbox(input_block[23:16], 1) ^
            ↪  aes_tbox(input_block[15:8], 1) ^ aes_tbox(input_block[7:0], 2)}
342       }
343     };
344
345   end
346 endfunction
347
348 // AES BOX Substitutions
349 // -- Function for AES SBOX
350 function [7:0] aes_sbox(input [7:0]in);
351   begin
352   case(in)    // synopsys full_case parallel_case
353     8'h00: aes_sbox=8'h63;
```

```
608     8'hff: aes_sbox=8'h16;
609   endcase
610   end
611 endfunction
612 // -- Function for AES SBOX
613 function [7:0] aes_tbox(input  [7:0]in, input integer mult);
614   // full_tbox maps to {Srd, 2*Srd, 3*Srd}
```

```
  reg [23:0] full_tbox;
  begin
  case(in)    // synopsys full_case parallel_case
    8'h00: full_tbox=24'b101001011100011001100011;
```

```
    8'hff: full_tbox=24'b001110100010110000010110;
  endcase
  // split for required mult
  // Constant matrix:
  // 02 03 01 01
  // 01 02 03 01
  // 01 01 02 03
  // 03 01 01 02
  // where, for AES TBOX
  // 03 = [23:16]
  // 02 = [15:8]
  // 01 = [7:0]
  case (mult)
    3: aes_tbox = full_tbox[23:16];
    2: aes_tbox = full_tbox[15:8];
    1: aes_tbox = full_tbox[7:0];
  endcase
  end
endfunction

endmodule
```

6.3.5 Output Interface

```
// Define module IO
module output_interface(
  input rst_, clk,
  // input from engine_round_transformer
  input transformer_done,
  input[127:0] ciphertext,
  // outputs
  output[7:0] data_out,
  output data_ok,
  output output_read
);

// Define registers
// -- ciphertext hold register
reg [127:0] output_hold;
initial output_hold = 128'h00000000000000000000000000000000;
// -- output register
reg [7:0] output_port;
initial output_port = 8'h00;
assign data_out = output_port;
// -- output control signals
reg data_ok_r;
initial data_ok_r = 0;
assign data_ok = data_ok_r;
reg output_read_r;
initial output_read_r = 0;
assign output_read = output_read_r;
```

```
// -- internal control signals
reg[4:0] i;
initial i = 5'b00000;

// Main Logic
always @(posedge clk) begin
  // reset
  if (!rst_) begin
    output_hold <= 128'h00000000000000000000000000000000;
    output_port <= 8'h00;
    i <= 5'b00000;
    output_read_r <= 0;
  end
  // output latch logic
  else if (transformer_done && i==0) begin
    // ? add check here to see if output_read is finished
    // ? not needed; it takes 8 clock cycles to output the ciphertext
    // ? and 8 clock cycles to input the new plaintext
    // ? therefore these two tasks may be done independently and simultaneously
    // ? without any conflicts
    // copy ciphertext
    output_hold <= ciphertext;
    // set i to begin
    i <= 1;
  end
  // normal clock
  else begin
    if (i == 0) begin
      // idling
      output_read_r <= 0;
    end
    else if (i == 1) begin
      // read first byte and set output data ok
      output_port <= output_hold[127:120];
      data_ok_r <= 1;

      output_hold <= output_hold << 8;
    end
    else if (i <= 16) begin
      // sequentially shift and output bytes 2-16
      // output_hold <= output_hold << 8;
      output_port <= output_hold[127:120];
      output_hold <= output_hold << 8;
    end
    if (i == 16) begin
      output_read_r <= 1;
      // ? this needs to be done earlier due to asynchronous assign
    end
    // increment i counter to next byte
    if (i >= 1 && i < 17) begin
      i <= i + 1;
    end
    else if (i == 17) begin
      // finished outputing reset
      data_ok_r <= 0;
      output_hold <= 128'h00000000000000000000000000000000;
      output_port <= 8'h00;
      i <= 5'b00000;
```

```
86          output_read_r <= 0;
87        end
88      end
89    end
90
91    endmodule
```

6.4 Testbench

Below is the testbench code used to debug each submodule of the system and the top module. It generates the signals shown in each of the waveform displays in the RTL Simulation section.

6.4.1 Top Module

```
`timescale 1ns/1ns // define simulation timescale

// define testbench module for aes_engine
module aes_engine_tb();

// define tb registers and wires
reg clk_tb = 0;
reg rst_tb = 0;

reg[7:0] din_tb = 0;
reg[1:0] cmd_tb = 0;

wire ready_tb, dok_tb;
wire[7:0] dout_tb;

// define tb for dout
reg[127:0] cipher_out;
initial cipher_out = 128'h00000000000000000000000000000000;

// define tb->aes_engine connections
aes_engine i_uut
(
  .clk(clk_tb),
  .rst_(rst_tb),

  .din(din_tb),
  .cmd(cmd_tb),

  .interface_ready(ready_tb),

  .dout(dout_tb),
  .data_ok(dok_tb)
);

// set periodic clock pulse every 1ns
always clk_tb = #1 ~clk_tb;

// state defines
localparam C_ID = 2'b00;  // Idle state
localparam C_SP = 2'b01;  // Set plaintext state
localparam C_SK = 2'b10;  // Set key state
localparam C_ST = 2'b11;  // Start encryption

// set callback for reading data
// read data
always @(negedge clk_tb) begin
  if (dok_tb) begin
    // append one byte to cipher_out reading register
```

```
    cipher_out = { cipher_out[119:0], dout_tb };
  end
end

// set testing sequence signals
initial begin
  // reset everything
  rst_tb = 0;
  #2;
  rst_tb = 1;
  #2;

  // enter plaintext
  cmd_tb = C_SP;
  din_tb = 8'h00;
```

```
  // enter key
  cmd_tb = C_SK;
  din_tb = 8'h24;
```

```
  // Start command
  cmd_tb = C_ST;

  // wait for data to be ok to go high
  @(posedge dok_tb)
  begin
    // Idle input interface
    cmd_tb = C_ID;
    // ? OR start inputting next plaintext

    // Read output ciphertext
    // in the routine above
  end

  // Signal read finished
  // wait for data_ok to go low again
  @(negedge dok_tb) begin
    $display("----------------");
    $display("Recieved ciphertext from output_interface: ");
    $write("%02X %02X %02X %02X\n", cipher_out[127:120], cipher_out[95:88],
    ↪  cipher_out[63:56], cipher_out[31:24]);
    $write("%02X %02X %02X %02X\n", cipher_out[119:112], cipher_out[87:80],
    ↪  cipher_out[55:48], cipher_out[23:16]);
    $write("%02X %02X %02X %02X\n", cipher_out[111:104], cipher_out[79:72],
    ↪  cipher_out[47:40], cipher_out[15:8]);
    $write("%02X %02X %02X %02X\n", cipher_out[103:96],  cipher_out[71:64],
    ↪  cipher_out[39:32], cipher_out[7:0]);
    #10;
    $finish;
  end

end

endmodule
```

6.4.2 Input Interface

```
`timescale 1ns/1ns // define simulation timescale

// define testbench module for input_interface
module input_interface_tb();

// define tb registers and wires
reg clk_tb = 0;
reg rst_tb = 0;

reg[7:0] din_tb = 0;
reg[1:0] cmd_tb = 0;
reg engind = 0;

wire ready_w;
wire enginC_w;
wire[127:0] plain_tb;
wire[127:0] key_tb;

// define tb->input_interface connections
input_interface i_uut
(
  .clk    (clk_tb),
  .rst_    (rst_tb),
  .din    (din_tb),
  .cmd    (cmd_tb),
  .ready    (ready_w),
  .transformer_done  (engind),  // AES engine OK signal return
  .key_start  (enginC_w),  // AES key START signal send
  .plain_out  (plain_tb),  // -> AES engine PLAINTEXT
  .key_out  (key_tb)  // -> AES engine KEY
);
// set periodic clock pulse every 1ns
always clk_tb = #1 ~clk_tb;

// state defines
localparam C_ID = 2'b00;  // Idle state
localparam C_SP = 2'b01;  // Set plaintext state
localparam C_SK = 2'b10;  // Set key state
localparam C_ST = 2'b11;  // Start encryption
// Enter this into the command console to add a custom radix for this
// radix define InputCommand {2'b00 "ID" -color blue, 2'b01 "SP" -color yellow,
↪  2'b10 "SK" -color orange, 2'b11 "ST" -color red}
// radix define ReadyBusy {0 "BUSY" -color red, 1 "READY" -color blue, -default
↪  symbolic}

// set testing sequence signals
initial begin
  engind = 1; // AES engine done / idle

  // reset input interface
  rst_tb = 0;
  #2;
  rst_tb = 1;
  #2;

  // enter plaintext
```

```
  cmd_tb = C_SP;
  din_tb = 8'h00;
```

```
  // Idle
  cmd_tb = C_ID;
  #2;

  // enter key
  cmd_tb = C_SK;
  din_tb = 8'h24;
```

```
  // Start command
  cmd_tb = C_ST;
  engind = 0; // AES engine not done / busy
  #2;
  #2;
  #2;
  #2;
  #2;
  #2;
  #2;
  #2;
  engind = 1; // Engine done
  #2;

  // Idle
  cmd_tb = C_ID;
  #2;
  $finish;
end

endmodule
```

6.4.3 Round Keys Generator

```
`timescale 1ns/1ns // define simulation timescale

// define testbench module for engine_key_generator
module engine_key_generator_tb();

// define tb registers and wires
reg clk_tb = 0;
reg rst_tb = 0;

reg[127:0] key_in_r = 0;
reg start = 0;

wire transformer_start_w;
wire[127:0] rk0; // pre-round key
wire[127:0] rk1;
wire[127:0] rk2;
wire[127:0] rk3;
wire[127:0] rk4;
wire[127:0] rk5;
wire[127:0] rk6;
```

```
wire[127:0] rk7;
wire[127:0] rk8;
wire[127:0] rk9;
wire[127:0] rk10;

// define tb->engine_key_generator connections
engine_key_generator i_uut (
  .rst_       (rst_tb),
  .clk       (clk_tb),
  .key_in       (key_in_r),
  .key_start     (start),
  .transformer_start  (transformer_start_w),
  .round0_key     (rk0),
  .round1_key     (rk1),
  .round2_key     (rk2),
  .round3_key     (rk3),
  .round4_key     (rk4),
  .round5_key     (rk5),
  .round6_key     (rk6),
  .round7_key     (rk7),
  .round8_key     (rk8),
  .round9_key     (rk9),
  .round10_key   (rk10)
);

// set periodic clock pulse every 1ns
always clk_tb = #1 ~clk_tb;

// set testing sequence signals
initial begin
  start = 0;

  // reset module
  rst_tb = 0;
  #2;
  rst_tb = 1;
  #2;

  // set key
  key_in_r = 128'h2475A2B33475568831E2120013AA5487;
  #2;
  //start
  start = 1;
  #500;
  $finish;
end

endmodule
```

6.4.4 Round Transformer

```
`timescale 1ns/1ns // define simulation timescale

// define testbench module for aes_engine
module engine_round_transformer_tb();

```

```
// define tb registers and wires
reg clk_tb = 0;
reg rst_tb = 0;
// -- inputs
reg [127:0] plain_tb = 128'h00041214120412000C00131108231919;
reg transformer_start_tb;
initial transformer_start_tb = 0;
reg output_read_tb;
initial output_read_tb = 0;
// -- input keys
reg[127:0] round0_key_tb  = 128'h2475A2B33475568831E2120013AA5487; // pre-round
↪  key
reg[127:0] round1_key_tb  = 128'h8955B5CEBD20E3468CC2F1469F68A5C1;
reg[127:0] round2_key_tb  = 128'hCE53CD1573732E53FFB1DF1560D97AD4;
reg[127:0] round3_key_tb  = 128'hFF8985C58CFAAB96734B748313920E57;
reg[127:0] round4_key_tb  = 128'hB822DEB834D8752E479301AD54010FFA;
reg[127:0] round5_key_tb  = 128'hD454F398E08C86B6A71F871BF31E88E1;
reg[127:0] round6_key_tb  = 128'h86900B95661C8D23C1030A38321D82D9;
reg[127:0] round7_key_tb  = 128'h62833EB6049FB395C59CB9ADF7813B74;
reg[127:0] round8_key_tb  = 128'hEE61ACDEEAFE1F4B2F62A6E6D8E39D92;
reg[127:0] round9_key_tb  = 128'hE43FE3BF0EC1FCF421A35A12F940C780;
reg[127:0] round10_key_tb = 128'hDBF92E26D538D2D2F49B88C00DDB4F40;
// -- outputs
wire [127:0] ciphertext_tb;
wire transformer_done_tb;

// define tb->engine_round_transformer connections
engine_round_transformer i_uut
(
  .clk(clk_tb),
  .rst_(rst_tb),

  // inputs
  .plaintext          (plain_tb),
  .transformer_start  (transformer_start_tb),
  .output_read        (output_read_tb),
  // -- keys
  .round0_key         (round0_key_tb),
  .round1_key         (round1_key_tb),
  .round2_key         (round2_key_tb),
  .round3_key         (round3_key_tb),
  .round4_key         (round4_key_tb),
  .round5_key         (round5_key_tb),
  .round6_key         (round6_key_tb),
  .round7_key         (round7_key_tb),
  .round8_key         (round8_key_tb),
  .round9_key         (round9_key_tb),
  .round10_key        (round10_key_tb),

  // outputs
  .ciphertext          (ciphertext_tb),
  .transformer_done    (transformer_done_tb)
);

// set periodic clock pulse every 1ns
always clk_tb = #1 ~clk_tb;

// set testing sequence signals
```

```
initial begin
  // reset round transformer
  rst_tb = 0;
  #2;
  rst_tb = 1;
  #2;

  // inputs set at the beggining of program

  // start encryption rounds
  transformer_start_tb = 1;

  // wait for finish
  // wait for transofrmer done to go high
  @(posedge transformer_done_tb)
  begin
    // Idle
    #20;
    // Signal that ouput has been read
    // (simulating output_interface input signal)
    transformer_start_tb = 0;
    output_read_tb = 1;
    #10;
    output_read_tb = 0;
    #10;
    $finish;
  end

end

endmodule
```

6.4.5 Output Interface

```
`timescale 1ns/1ns // define simulation timescale

// define testbench module for output_interface
module output_interface_tb();

// define tb registers and wires
reg clk_tb = 0;
reg rst_tb = 0;

reg[127:0] cipher_in;
initial cipher_in = 128'hBC028BD3E0E3B195550D6DF8E6F18241;

reg transformer_done_r;

wire[7:0] data_out;
wire data_ok, output_read;

// define tb for dout
reg[127:0] cipher_out;
initial cipher_out = 128'h00000000000000000000000000000000;

// define tb->output_interface connections
```

```
output_interface i_uut
(
  .clk        (clk_tb),
  .rst_       (rst_tb),

  .transformer_done   (transformer_done_r),
  .ciphertext         (cipher_in),

  .data_out           (data_out),
  .data_ok            (data_ok),
  .output_read        (output_read)

);
// set periodic clock pulse every 1ns
always clk_tb = #1 ~clk_tb;

// set callback for reading data
// read data
always @(negedge clk_tb) begin
  if (data_ok) begin
    // append one byte to cipher_out reading register
    cipher_out = { cipher_out[119:0], data_out };
  end
end

// set testing sequence signals
initial begin
  // reset output interface
  rst_tb = 0;
  #2;
  rst_tb = 1;
  #2;

  // simulate aes_engine output ready
  transformer_done_r = 1;

  // wait for data_ok to go low again
  @(negedge data_ok) begin
    $display("Recieved ciphertext from output_interface: ");
    $write("%02X %02X %02X %02X\n", cipher_out[127:120], cipher_out[95:88],
    ↪  cipher_out[63:56], cipher_out[31:24]);
    $write("%02X %02X %02X %02X\n", cipher_out[119:112], cipher_out[87:80],
    ↪  cipher_out[55:48], cipher_out[23:16]);
    $write("%02X %02X %02X %02X\n", cipher_out[111:104], cipher_out[79:72],
    ↪  cipher_out[47:40], cipher_out[15:8]);
    $write("%02X %02X %02X %02X\n", cipher_out[103:96],  cipher_out[71:64],
    ↪  cipher_out[39:32], cipher_out[7:0]);
    #2;
    $finish;
  end
end

endmodule
```