

**NANENG524**  
**Testing ,Verification and Reliability -**  
**SPRING2024**  
**Final Project: Verification of a Scrambler**

**Hatem Mohamed 201-900-577**

Supervised by:

Dr. Rania Osama

Dr. Sherif Hosny

Eng. John Wafeek

April/May 2024

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Scrambler/descraler uses in MPEG-2 multiplexing . . . . .	3
<b>2</b>	<b>Scrambler's operation</b>	<b>4</b>
<b>3</b>	<b>HDL</b>	<b>5</b>
3.1	BB Scrambler . . . . .	5
3.2	Testbench SystemVerilog code . . . . .	7
3.3	Testbench explanation . . . . .	9
3.3.1	Testbench core . . . . .	9
<b>4</b>	<b>Simulation and Results</b>	<b>9</b>
4.1	Simulation setup . . . . .	9
4.2	Results . . . . .	10
4.2.1	Waveforms . . . . .	10
4.2.2	Transcript . . . . .	12
<b>5</b>	<b>Appendix: Documentation</b>	<b>13</b>
5.1	Faced Problems: . . . . .	13

# 1 Introduction

Protection of transmitted data privacy and security is critical in the field of communication systems. One important technique used to protect sensitive data is the application of scramblers. A cryptographic tool or procedure known as a scrambler modifies the structure or format of data so that unapproved parties are unable to understand it. Scramblers prevent eavesdropping attempts and improve communication channel confidentiality by rearranging the data before transmission. In this project a scrambler's functionality is being digitally verified using Hardware discription and verification language SystemVerilog and via the Universal Verification Methodology (UVM). In MPEG-2 multiplexing, multiple audio, video, and data streams are combined into a single transport stream for efficient transmission. Scrambling ensures that these streams are protected from unauthorized access or interception during transmission. By scrambling the data, it becomes unintelligible to anyone without the proper descrambling key, thus safeguarding the confidentiality of the content.

## 1.1 Scrambler/descraler uses in MPEG-2 multiplexing

Motion Picture Experts Group-2 is known as MPEG-2. It is a standard for encoding audio information that goes along with moving visuals. MPEG-2, which was created by the Moving Picture Experts Group (MPEG), is extensively utilized in digital video distribution applications such as DVDs and digital television broadcasts. It outlines the multiplexing and synchronization of numerous streams into a single transport stream for transmission, as well as the compression algorithm for audio and video data. MPEG-2 is now a fundamental component of contemporary multimedia technology, having played a key role in the shift from analog to digital television transmission.

1. **Privacy and Security:** To facilitate efficient transmission, several audio, video, and data streams are integrated into a single transport stream by MPEG-2 multiplexing. By scrambling, these streams are shielded from unwanted access and transmission interception. The data is scrambled so that anyone lacking the correct descrambling key cannot decipher it, protecting the content's confidentiality.
2. **Conditional Access Control (CAS):** In MPEG-2 multiplexing, scramblers are frequently used in tandem with CAS to regulate access to particular content according to subscription status or other factors. To guarantee that only authorized users may decode and access the content, conditional access systems rely on encryption and scrambling. Because they render content illegible for users lacking the necessary authorization, scramblers are essential to the enforcement of these access rules.
3. **Preventing Unauthorized Distribution and Piracy:** MPEG-2 multiplexed content distribution and piracy are discouraged via scrambling. It is far more difficult for unauthorized parties to intercept and replicate the content without the necessary decryption keys when the content is scrambled during transmission. This aids content providers in keeping control of their intellectual property and deters piracy.
4. **Adherence to Industry Standards:** MPEG-2 is a commonly used standard for multiplexing and compressing digital video. Scrambling is frequently used in compliance with MPEG-2 standards to guarantee compatibility and interoperability across various systems and devices. Broadcasters and content producers can make sure that their multiplexed content works with a variety of consumer devices and transmission infrastructures by following these standards.

## 2 Scrambler's operation

The operation could be easily understood as follows:

1. Initialization: The scrambler is initialized with a predefined key or seed value prior to transmission. This key is used to manage the data scrambling process and guarantee that the data can be decoded by the transmitter and receiver. In this case the initialization sequence is (100101010000000) and it shall be initiated at the start of every eight transport packets.
2. Data Scrambling: The scrambler processes the input data, which may include audio, video, or other digital information. Using the key or seed value as a basis, this technique applies a mathematical algorithm or operation to the data. The scrambler's architecture and the system's requirements determine the precise algorithm that is employed. In this case, the polynomial is  $1 + X^{14} + X^{15}$  which means that in the feedback loop, the 14th, 15th, and input bit are XORed together in order to generate this random data.
3. Transmission: The data is sent over the communication channel once it has been jumbled. Anyone intercepting the scrambled data without the correct descrambling key will see it as random or incomprehensible noise.
4. Descrambling: Using the same key or seed value as the transmitter, the descrambler is initialized at the receiving end. By applying the opposite operation to the scrambled data, the descrambler is able to recover the original content and essentially undo the scrambling process.

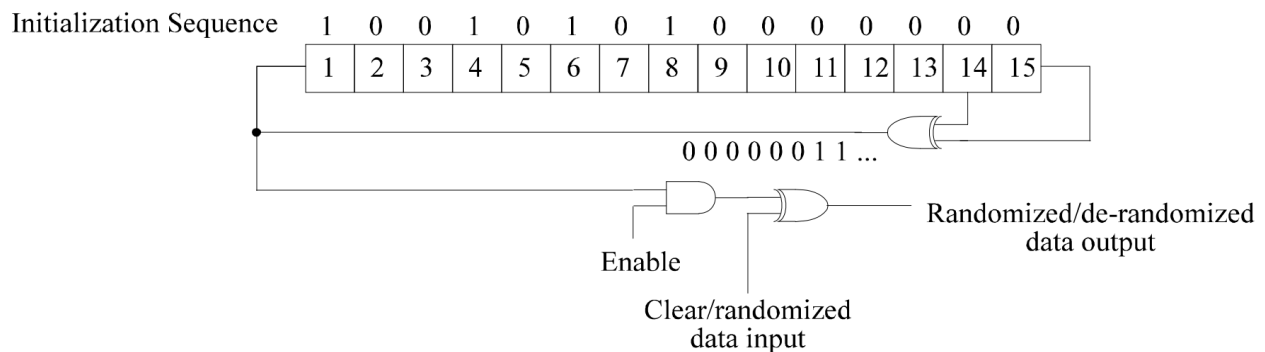


Figure 1: Scrambler/descrambler schematic diagram

The total packet length of the MPEG-2 transport multiplex (MUX) packet is 188 bytes. This includes 1 sync-word ( $47_{hex}$ ) byte. The MSB "0" of the sync-word byte  $47_{hex}$  (or 01000111) is always where the processing order on the transmitting side begins. To provide an initialization signal for the descrambler, the MPEG-2 sync byte of the first transport packet in a group of eight packets is bit-wise inverted from  $47_{HEX}$  ( $SYNC$ ) to  $B8_{HEX}$  ( $SYNC$ ).

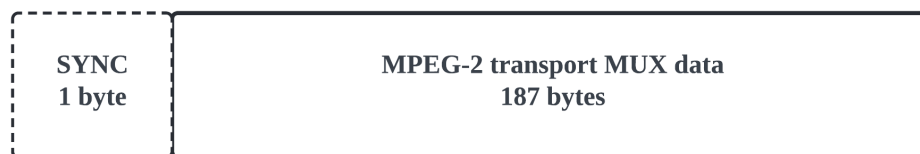


Figure 2: MPEG-2 transport MUX packet

## 3 HDL

### 3.1 BB Scrambler

This part was given as an encrypted module in order to explore it using the implemented testbench so as seen below it cannot be read.

```

1  module BB_scrambler (
2  input clk,
3  input reset_n,
4  input [1:15]initial_state,
5  input in_bit,
6  input en,
7  output reg out_bit,
8  output reg scmb_en
9  );
10 'pragma protect begin_protected
11 'pragma protect version = 1
12 'pragma protect encrypt_agent = "ModelSim" , encrypt_agent_info = "10.5"
13 'pragma protect key_keyowner = "Mentor Graphics Corporation" , key_keyname = "MGC↵
    -VERIF-SIM-RSA-2"
14 'pragma protect key_method = "rsa"
15 'pragma protect encoding = ( enctype = "base64" , line_length = 64 , bytes = 256 ↵
    )
16 'pragma protect key_block
17 GXMIvkdAQeEg7doh/EZltCufPHoU6v0h3t7ADnXbrEDkyoiYvH85w7/j+nJs/DX7
18 bV40D4X+8gRf0bYp3i0sKCIxKfhcCddoAHd12PCl5KVfJk0KGUMkNCORicwbja7V
19 xt0Q6kRKhVnFj2rKAiCsW6lJ/2E/zdWHZTXi5V2hfhCPwXUqwbzCb5j5foziMWHx
20 +U2T78a0E3sx9ZAMJVeReyg4rb7Nj7BSsbvCGcfUhVn+G50ZX+5lqN502efSvVri
21 9j8mWzeMXtFJBE4a+eCYvrCD32PQd+D6ze3WUkZxx00dGopxS7bcfn919tnN70HS
22 mjGTZYP+ebilco0gzgKFvQ==
23 'pragma protect data_method = "aes128-cbc"
24 'pragma protect encoding = ( enctype = "base64" , line_length = 64 , bytes = 2000↵
    )
25 'pragma protect data_block
26 siVB8LXmt8qgxBLsie3sbh2HX4aIlq3str6TcUi+N5YZyQ8jQtG9/Iu5PywqLyfC
27 CnC2xhEpytyvRdIH3hyTI8/1B3alHuUBHEX0SjN7oSxX9zz+oICSZmTJHdHUGJ5L
28 QblzikxfawZg0UtQGLowq3gAdHZzo6pACnnIYcAbxe0cgVquclPf+hY7S1Uk/cTQ
29 kfW0X0h1741DWxIiqjDlvB7u/zbi+jX4Vdrn4E5F03Zr8MsUw/c0+OLCETkZkBrP
30 i6Ss4bGIha5K+HWW6gVga/kzKwR9HGd+z6WOB2SIiUBMh+gXrv+n5/fNerOtVw/L
31 GNa6bGfYNl1PHDmwqgBZDALb6TOWAjsxCVetf29jFBIInwllptPWxsgSeJHJ+9w32
32 CKu6KgY4lsxVY/pgCJ0dvYvoCxsG7j6389/WVBUTHcwMkWJbvKwzd06JLVyE8neH
33 Wg96U1Cu46Pwe1G4v1mNPDuxdSccascFUdKc0bnRDvsZryZb4MvosglqAEODbUGq
34 VQ37XQjXrs49IAZTqFoeAkgJeUhDY4FQEAwLqv5bI8QaeUhZmCqKwNlCKjEEeTB0
35 1rt0FhifP2zi5NM8MzG1zMHIwy+Cz0Tfbogo1+dUvfi+wAFUWbvEW8H8PSs18pWn
36 N35Jyeb0tKJNKQ3bjC5w8U5R08iRKT3Hrjx7zfxwmWRvuXkXhua6n/udV5CZQMnG
37 Ror6NBjz3ry0z70FrlqZjdhhZZw+exyW2aMUBP4Y7RlW0Q420B42e7rspGbGRWLL
38 f9mrd0sMaxciy1PIXn0nNWy8Sq7Q10u6+lEEaM03YvwE8vzmrtj0geQkJWW1xqGU
39 FlpvYxEhXLa3hKeeeeeU17gBYyMNd+1JezGGsy10ou6WzroCoV45AI28H32HFAggb
40 5y7Mb+MEOPKkKnPzfWLL+YhGsFrFzXY/3o+7+Rhq7xP8rBAU0Y21IXmngi7T3aHZ
41 3vu4i19mKYzaQA+LW5Af2hHCZx3nGH0g2Mt4PZrDXFcTp/olq4sSGuN/U7qaxB
42 NR9QoU7T+iiuqodptYs5Sarta0Q7aIpN8h1U48zTThBeV+14w6St9FCu04zfmRRd
43 a/eJJTtpn6ngbR+7qfjQMBzISW4Lti5Lue4ElSoI4nqTymoIB561Eieu3jxpwdn
44 chdDJhcXWJXVWWHJgbbMOP/R6I93NoktqI/ZH0J1aLxDdpwDojHGlBgh6t02LC77
45 kisA79vInsuoS0hB9fA/xF3oJCZfpyNKRNMJzjQlidOTqSTZKvbybT/5jTOE1IxT
46 Cv5L379TtdxDCW6rNH0CdAc5F+1EsRy30wjP0/d2ftWMZUeXkV1bNUwvThitpuxq
47 YxlQ7XHJucDjlnRWyEMTGWlRttOpH7zpz6i8N5j2arAKdFdxkIR+ry3VgZPt5veh
48 mrN50wyRTIzZzdygbdwlqxappYozv/6Ww3PZa8Pb3v0DejIRNmb+WNQkmSYoHIkB
49 3WaXbsmub7yc9eQ6kBqRLjsDwHMz0W4ZsLfi0fn9VRwttj7UaQ0v3o4B7b6uTT1N

```

```

50 cOP0eMWpZ76hUnV5GFG+LacySjzMUxFPMIdpD7cCVw91a0IZZQIPeoyoA5np1qV/
51 6TLc/KdKqfNZrM8/ZyGXNt6J4uWapaL8HbfocM0XKfAeIMyzT1Pu0MZ5eNLNCKns
52 2EZDExj+FctjIhp7PNWfKKn0HcOL+4Kn0Cn0WoLxz1qj0FGTAY/o/mKyPWTj015D
53 xRY8FWDMad1CAph+0rSGihNJCCTGJH1R4DkwmPetF/hWTJiiIsr9Qjdr9/bKGcA+
54 RQWeRfHa80mkIUjW4jC/8/DHvJuzQw0F0SFTJ1gbmfz4iV+YnvLaEVr07drn0f3J
55 cfdN0ucJr5ivsDoa6SSGZVbo+30bpy/bt4xjGnxVqn7AiwQeM0cNk2GMCc8Vns+m
56 4qbkRvrOwyRizrt3KSaIEL/hS+18wiWMixQdosn9X/eHFzIM4ymBXrzZXwuhaI4p
57 qYCQq3TGBesjNYGL5Kz/0EU+kjypzo0nR0ZrtzgqdhhetEvRzExRpFrFZ/p1prFIV
58 CT+0hRNQ1zXWRjhG4RghcCwBdtZ87FSfIFh2bDjHA+HNTmmnCsd9cXESiyoT7LBY
59 YS0peodfw0XBkunZfM7exitJkj10fQIOQLvcBUk/OwiyAzLtfVzUK/P1oS2d2t8
60 SABqziW6WtaL5PNG0d0JmRphk3ht+U26rKp9dRZzw4vN0vv9f0PpUoMZAjdTmeuB
61 KKnsWvFCOX0ot37zDpHswRJPnbn0k2Z9n6Jxe6ymzAooczlmX4YxaN2Cux+CQbV9
62 gcxDBw9JevCrteRhu+EzhhJpfAQSmqsKRVd+G71ulYd23cr7jgdy0S21LgVqGp3k
63 MKedBI2zfqUHRXXx870CjW/I5ZtDTwGgqF2phKNqyP2eprMMb+nwxeH9m0IpwI+0
64 6CY2SBpzBW7CpuoKhSnwHVPPhSg1+RXFgXTrnfnNsKgBW3MSM3GyEIxitVI/1Gyr6
65 1ArJtKoc+ZvSSJRG10xTe6t7Fi8GJCTvrmreBZnu3AwyJKS8CA02Mq1gljhbdUgx
66 kFR0RvwkEUME+cbNorntXmp1i+a8DbAfwSyP8jhp4/dl7HYb8rPjGTv8H5EtET4E
67 vK2GyptsSMLIo43MmD0FleXKVZ6CK/7Re7vu5NyEbaw=
68 'pragma protect end_protected
69
70 endmodule

```

But by studying and relating with literature it could be seen that it is typical to what is seen in 8. With using given test vectors for input and scanning the output a proper testbench could be formulated.

### 3.2 Testbench SystemVerilog code

```

1 module scrambler_tb ();
2 reg clk;
3 reg reset_n;
4 reg [1:15] initial_state;
5 reg in_bit;
6 reg en;
7 wire out_bit;
8 wire scmb_en;
9 reg test_in [1:1504*50];
10 reg test_out [1:1504*50];
11 reg out_exp;
12 int pck;
13 int i;
14 bit ch;
15 BB_scrambler dut(
16     .clk(clk),
17     .reset_n(reset_n),
18     .in_bit(in_bit),
19     .initial_state(initial_state),
20     .en(en),
21     .out_bit(out_bit),
22     .scmb_en(scmb_en)
23 );
24 initial clk=0;
25 always begin
26     #5; clk=~clk; // clock period 10
27 end
28 task reset();
29     reset_n= 0;
30     #10;
31     reset_n=1;
32 endtask
33 // read from file
34 task read();
35     // let's read the vectors (I called it input)
36     $readmemb("Input.txt",test_in);
37     $readmemb("Output.txt",test_out);
38 endtask
39 task check (bit x, bit y, int pck, int j);
40     if(x==y) begin
41         $display("Right one :) exp= %b, true= %b, packet number %d, bit number %d↵
42             ",y,x,pck,j);
43     end
44     else begin
45         $display("Wrong one!! exp= %b, true= %b, packet number %d, bit number %d"↵
46             ,y,x,pck,j);
47     end
48 endtask
49
50 initial begin //run time
51     // reset();
52     read();
53     initial_state= 15'b1001010100000000;

```

```
54 // en=1;
55 // we would read now bro
56 for (pck=1;pck<51;pck++) begin
57     if (pck>1) begin
58         reset();
59         en=0;
60         #25
61         en=1;
62     end
63 else begin
64     reset();
65     en=0;
66     #20
67     en=1;
68 end
69 for (i =1 ;i<=1506; i++) begin
70     @ (negedge clk) begin
71         in_bit= test_in[1504*(pck-1)+i];
72         if (i>2)
73             begin
74                 check(out_bit ,out_exp,pck,i-2);
75             end
76     end
77     @ (posedge clk) begin
78         out_exp=test_out[(pck-1)*1504+i-1]; // -1 here is cause it works after en←→
79         by 1 cyclez
80     end
81 end
82 $finish;
83 end
84 endmodule
```



### 3.3 Testbench explanation

As seen in the Encrypted RTL the I/O pins could be read in order to force or read from them. With relating to the block diagram, the scrambler works with a negative reset and it works synchronously. Initial state is the value kept in each register at the beginning of every scrambling process to an MPEG-2 Packet. while **en** is the enable signal in order to do the scrambling, **out\_bit** is the randomized bit and **scmb\_en** is the signal that shows that the scrambling is happening now. The testbench is written to do the following:

1. Let the device capture 50 frames each of which is 1504 bits.
2. Test if the randomized output matches the expected one given.

And it consists of the following:

1. Three tasks: The first one is called reset in order to trigger the reset signal and reset the whole device. The task forces **reset\_n** signal to zero and after 1 cycle it goes to 1 again. The second task is used to read the text files that include the test vectors in order to force the input (**Input.txt**) and compare it with the output (**Output.txt**). The third one is used to check on output's validity.
2. One always block: The one used to generate the clock signal with period of 10ns (arbitrary).
3. Two initial blocks: The first one is used to initialize the clock in order to be generated. The second one is the core of the test bench

#### 3.3.1 Testbench core

As mentioned before the second initial block can be considered as the core of the testing process. It consists of 2 nested for loops. The upper one used to go through MPEG-2 frames from 1 to 50, the second one to scan the 1504 bits of each frame. Every time a frame is completed a Reset operation shall be performed and the Enable signal shall go to zero in order to return to the initial state of the scrambler. The input signal and the Enable signal are fed to the device on each falling clock edge. At the rising edge, the input is sampled and the output compared to the expected one. It shall be mentioned that the device was observed to react to the input after a whole clock cycle. for example if the input is 1 and the output response to this input is 0, it would take the device a complete cycle to generate that 0. As a result of the previous observation, the output needs to be compared to the previous signal from the test vector output file (**i-1**). The inner for loop loops to 1506 because the first iteration is used to initialize the process as inside them the first input takes place, but as mentioned its effect appear after one complete cycle. And the extra iteration is used to assure the last bit in the frame is generated well.

## 4 Simulation and Results

### 4.1 Simulation setup

The .do file needed to run the testbench.

```

1 vsim -voptargs=+acc work.scrambler_tb
2 add wave -position insertpoint \
3 sim:/scrambler_tb/clk \
4 sim:/scrambler_tb/reset_n \
5 sim:/scrambler_tb/initial_state \
6 sim:/scrambler_tb/in_bit \
7 sim:/scrambler_tb/en \
8 sim:/scrambler_tb/out_bit \
9 sim:/scrambler_tb/scmb_en \
10 sim:/scrambler_tb/out_exp \
11 sim:/scrambler_tb/pck \
12 sim:/scrambler_tb/ch
13 run -all

```

4.2 Results

4.2.1 Waveforms

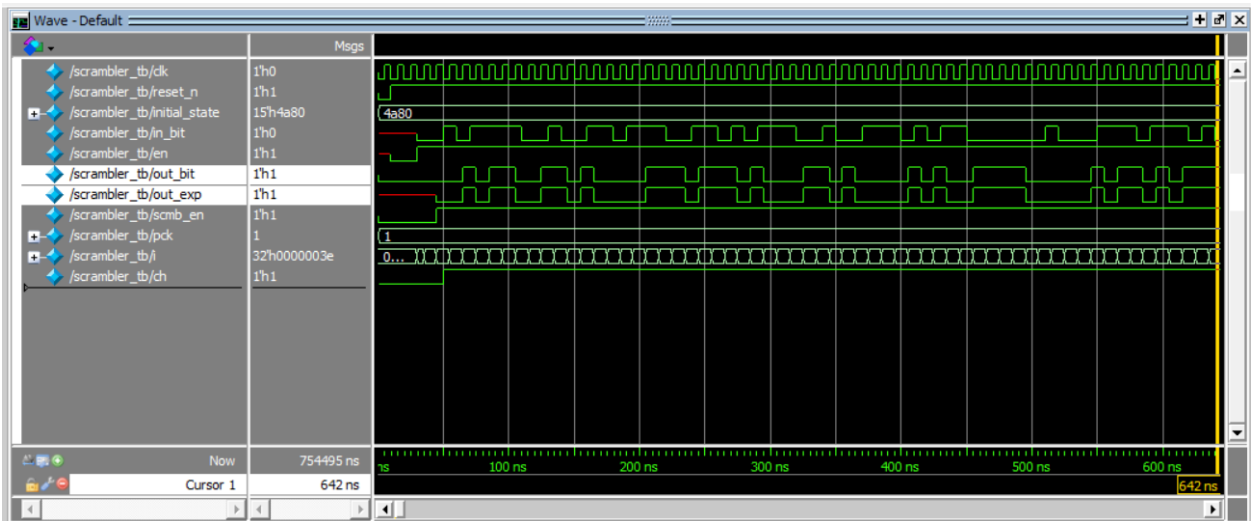


Figure 3: Waveform at the start of the simulation

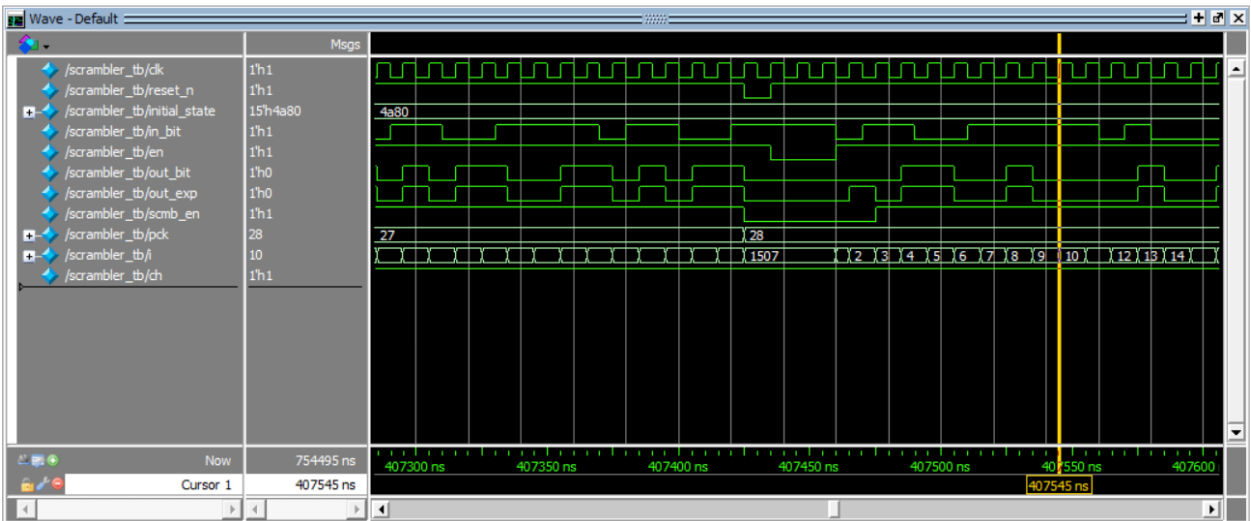


Figure 4: Waveform that depicts the packet transition

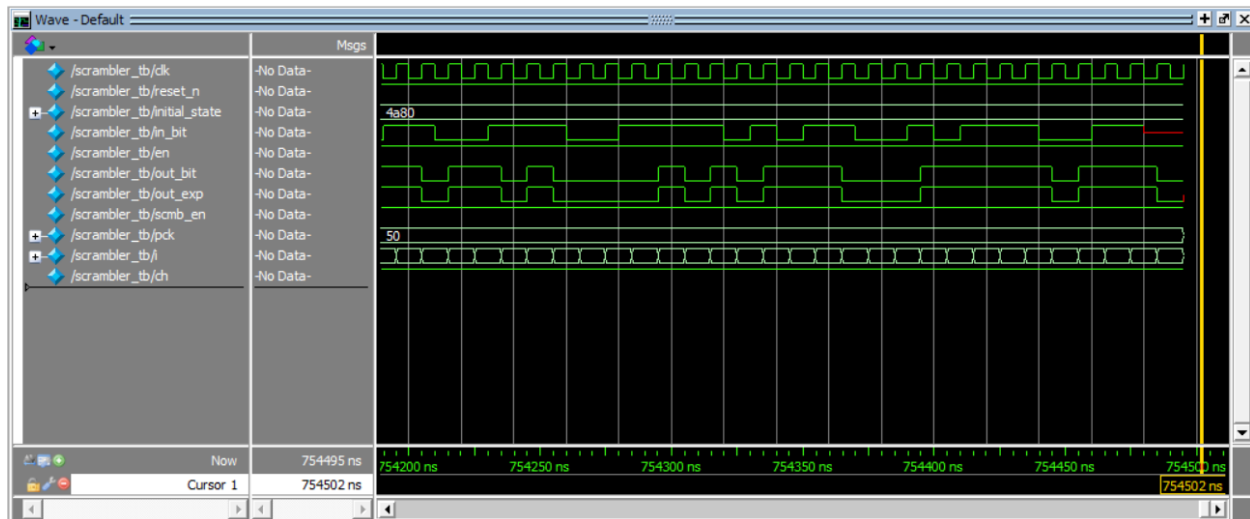
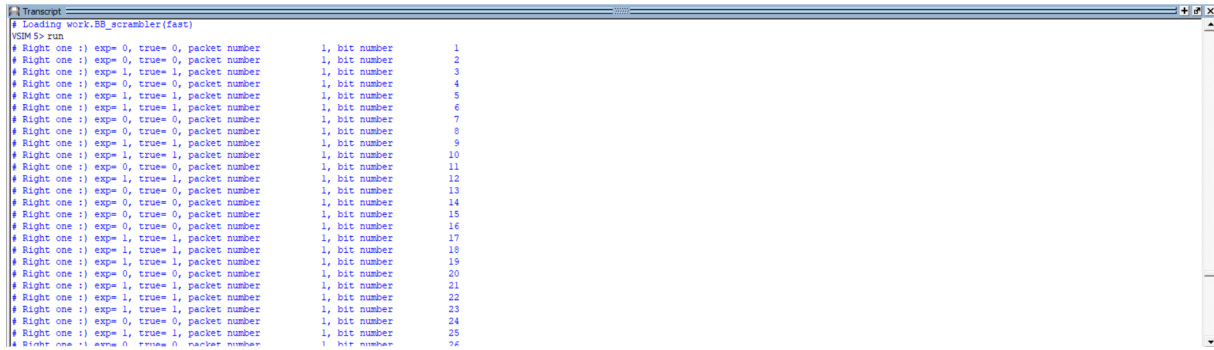


Figure 5: Waveform at the end of the simulation

## 4.2.2 Transcript

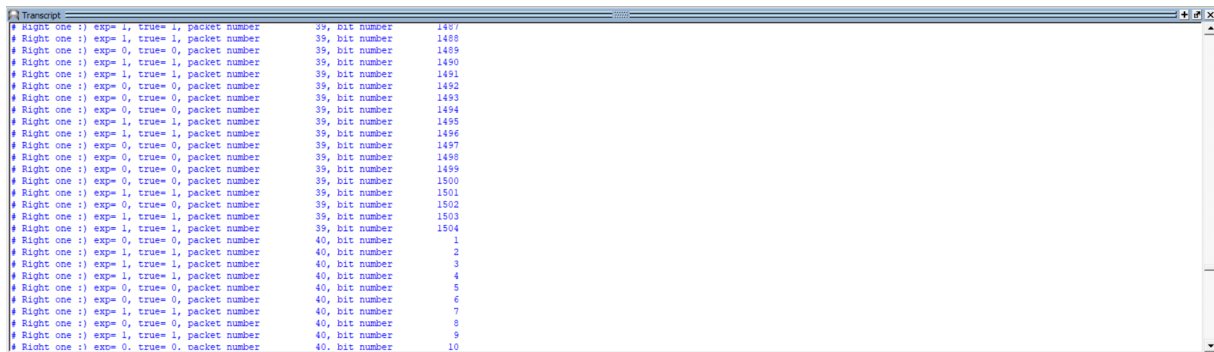


```

Transcript
# Loading work.BB_scrambler(fast)
VSIModelSim> run
# Right one : exp= 0, true= 0, packet number 1, bit number 1
# Right one : exp= 0, true= 0, packet number 1, bit number 2
# Right one : exp= 1, true= 1, packet number 1, bit number 3
# Right one : exp= 0, true= 0, packet number 1, bit number 4
# Right one : exp= 1, true= 1, packet number 1, bit number 5
# Right one : exp= 1, true= 1, packet number 1, bit number 6
# Right one : exp= 0, true= 0, packet number 1, bit number 7
# Right one : exp= 0, true= 0, packet number 1, bit number 8
# Right one : exp= 1, true= 1, packet number 1, bit number 9
# Right one : exp= 1, true= 1, packet number 1, bit number 10
# Right one : exp= 0, true= 0, packet number 1, bit number 11
# Right one : exp= 1, true= 1, packet number 1, bit number 12
# Right one : exp= 0, true= 0, packet number 1, bit number 13
# Right one : exp= 0, true= 0, packet number 1, bit number 14
# Right one : exp= 0, true= 0, packet number 1, bit number 15
# Right one : exp= 0, true= 0, packet number 1, bit number 16
# Right one : exp= 1, true= 1, packet number 1, bit number 17
# Right one : exp= 1, true= 1, packet number 1, bit number 18
# Right one : exp= 1, true= 1, packet number 1, bit number 19
# Right one : exp= 0, true= 0, packet number 1, bit number 20
# Right one : exp= 1, true= 1, packet number 1, bit number 21
# Right one : exp= 1, true= 1, packet number 1, bit number 22
# Right one : exp= 1, true= 1, packet number 1, bit number 23
# Right one : exp= 0, true= 0, packet number 1, bit number 24
# Right one : exp= 1, true= 1, packet number 1, bit number 25
# Right one : exp= 0, true= 0, packet number 1, bit number 26

```

Figure 6: The transcript at the start of the simulation

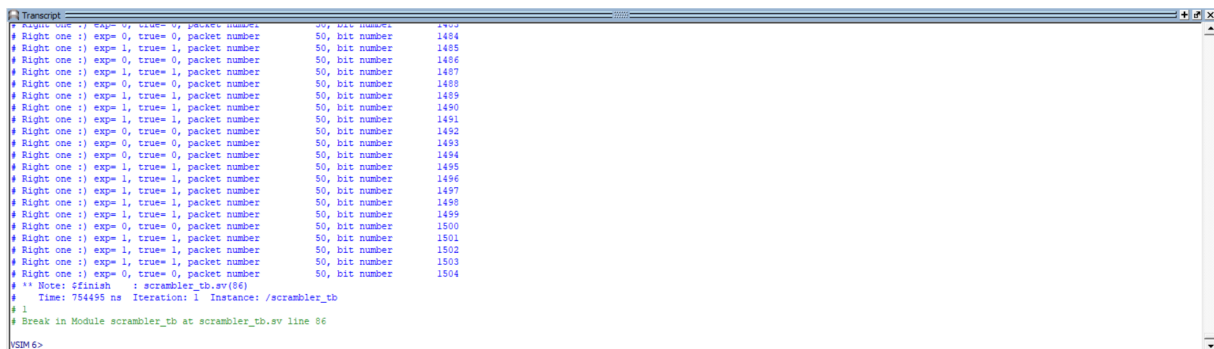


```

Transcript
# Right one : exp= 1, true= 1, packet number 39, bit number 1487
# Right one : exp= 1, true= 1, packet number 39, bit number 1488
# Right one : exp= 0, true= 0, packet number 39, bit number 1489
# Right one : exp= 1, true= 1, packet number 39, bit number 1490
# Right one : exp= 1, true= 1, packet number 39, bit number 1491
# Right one : exp= 0, true= 0, packet number 39, bit number 1492
# Right one : exp= 0, true= 0, packet number 39, bit number 1493
# Right one : exp= 0, true= 0, packet number 39, bit number 1494
# Right one : exp= 1, true= 1, packet number 39, bit number 1495
# Right one : exp= 1, true= 1, packet number 39, bit number 1496
# Right one : exp= 0, true= 0, packet number 39, bit number 1497
# Right one : exp= 0, true= 0, packet number 39, bit number 1498
# Right one : exp= 0, true= 0, packet number 39, bit number 1499
# Right one : exp= 0, true= 0, packet number 39, bit number 1500
# Right one : exp= 1, true= 1, packet number 39, bit number 1501
# Right one : exp= 0, true= 0, packet number 39, bit number 1502
# Right one : exp= 1, true= 1, packet number 39, bit number 1503
# Right one : exp= 1, true= 1, packet number 39, bit number 1504
# Right one : exp= 0, true= 0, packet number 40, bit number 1
# Right one : exp= 1, true= 1, packet number 40, bit number 2
# Right one : exp= 1, true= 1, packet number 40, bit number 3
# Right one : exp= 1, true= 1, packet number 40, bit number 4
# Right one : exp= 0, true= 0, packet number 40, bit number 5
# Right one : exp= 0, true= 0, packet number 40, bit number 6
# Right one : exp= 1, true= 1, packet number 40, bit number 7
# Right one : exp= 0, true= 0, packet number 40, bit number 8
# Right one : exp= 1, true= 1, packet number 40, bit number 9
# Right one : exp= 0, true= 0, packet number 40, bit number 10

```

Figure 7: The transcript at packet transition



```

Transcript
# Right one : exp= 0, true= 0, packet number 50, bit number 1487
# Right one : exp= 1, true= 1, packet number 50, bit number 1488
# Right one : exp= 0, true= 0, packet number 50, bit number 1489
# Right one : exp= 1, true= 1, packet number 50, bit number 1490
# Right one : exp= 0, true= 0, packet number 50, bit number 1491
# Right one : exp= 1, true= 1, packet number 50, bit number 1492
# Right one : exp= 1, true= 1, packet number 50, bit number 1493
# Right one : exp= 0, true= 0, packet number 50, bit number 1494
# Right one : exp= 1, true= 1, packet number 50, bit number 1495
# Right one : exp= 1, true= 1, packet number 50, bit number 1496
# Right one : exp= 1, true= 1, packet number 50, bit number 1497
# Right one : exp= 1, true= 1, packet number 50, bit number 1498
# Right one : exp= 0, true= 0, packet number 50, bit number 1499
# Right one : exp= 0, true= 0, packet number 50, bit number 1500
# Right one : exp= 1, true= 1, packet number 50, bit number 1501
# Right one : exp= 1, true= 1, packet number 50, bit number 1502
# Right one : exp= 1, true= 1, packet number 50, bit number 1503
# Right one : exp= 0, true= 0, packet number 50, bit number 1504
# ** Note: $finish : scrambler_tb.sv(86)
# Time: 754495 ns Iteration: 1 Instance: /scrambler_tb
# Break in Module scrambler_tb at scrambler_tb.sv line 86
VSIModelSim>

```

Figure 8: The transcript at the end

## 5 Appendix: Documentation

### 5.1 Faced Problems:

1. The fact that the input value affects the following output with a separation of one cycle made it difficult to find the most optimised way to compare the output to the expected one. It was solved using extra iterations and the shifted index.
2. the transition between packets was not smooth and there was always a problem. The reason for that problem was then discovered to be that the inner for loop always ends at a positive clock edge, thus upon resetting and enabling/disabling the enable signal is triggered on the positive edge, when it was planned to trigger at the negative edge so extra half cycle was added and seen in the **#25** line.

## References

- [1] Mentor Graphics. "QuestaSim 10.7" Wilsonville, OR: Mentor Graphics, 2020.
- [2] ETSI EN 300 744, EUROPEAN STANDARD, V1.6.2 (2015-10).