NYU-ECE-GY 6463
Final Project: Processor Design

FINAL REPORT-TEAM 16

ANKIT KUMAR(AK7311), PRAHARSH PARASHARA(PP1968),
NING LI(NL1802), KARTHIK SRIRAM(KS5189)

DATE OF PROJECT PRESENTATION: 12/16/2018

DUE DATE: 11:55 PM, 12/15/2018

Table of Contents

Objective

The objective of this project was to synthesize and create the layout of the 32-bit processor and implement it on an FPGA. The steps involved multiple tasks such as writing machine code and storing it into a file, modifying the VHDL files as well as adding the components, modifying the files required for them, verifying it using testbenches, and finally implementing the RC5 encryption, decryption, and round key generation programs on the FPGA.

The limitations of this project however, were that we are strictly expected to adhere to the capacity of Xilinx tools and the FPGA board used.

Introduction

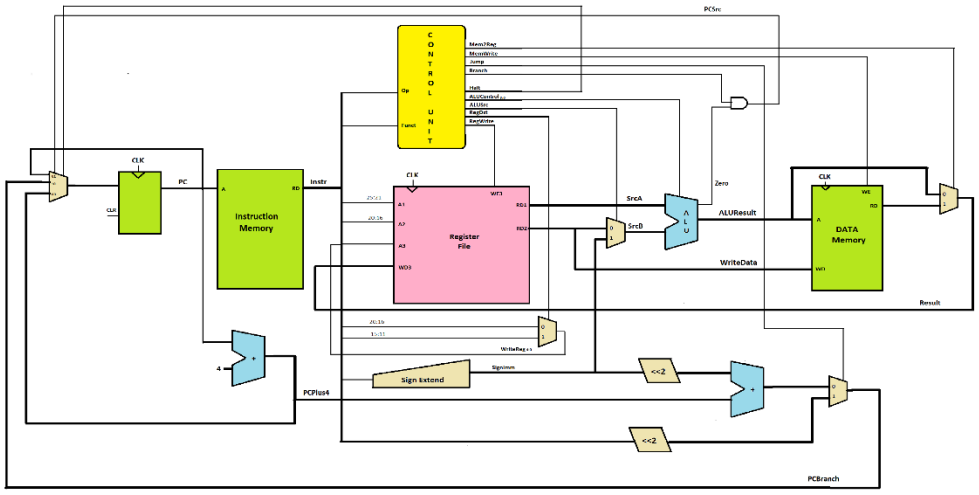
The motivation of this project was to understand multiple concepts that had to be implemented towards building something as challenging as a functioning central processing unit. Be it VHDL coding, assembly language it's machine code conversion or using Xilinx tools to project the layout of this complex 32-bit processor, we had to follow through with the entire process to engineer and design it. The engineering and design challenges that one might face is to understand the principles behind VHDL coding and testing it. Challenges such as being able to handle tools as Xilinx Vivado and HLS, that requires skill and perseverance can be identified in several portions of implementing this CPU. Also, to be able to

understand the working and functionality of every individual component and the prerequisites/ requirements to design it can be challenging and requires one to be able to go through the learning curve of the entire concept.

Background

Fundamentally, to begin with the design of the processor, the pre-work to be done is to understand the purpose and functionality of all its components which includes the memory, the ALU, the PC, the control unit, data memory and its instruction set. The flip-flops and the tri-states that are necessary must be anticipated. The first step is to design the VHDL code for all the components and concepts that were implemented onto it. Then, the assembly language code to perform operations is analyzed and converted into machine code and stored into the file ‘’. The 32-bit processor being larger in size, therefore requires extension of all the various components and they are stored in the file ‘’. Then, the functionality of this file is tested using Xilinx Vivado, where the synthesis and implementation is done, that verifies if the file performs the desired operations that has programmed as instructions by the user. Once done so, the designer then performs functional validation and verification using functional and timing simulation, after which the 32-bit processor layout is finally generated as a bitstream to be implemented on the FPGA board. Additionally, the RC5 encryption, decryption, and round key generation programs are verified using testbenches and then the processor is used to perform these functions on the FPGA board.

Block Diagram



Processor Components and Interface Analysis

From the given project manual, the processor components identified were:

• Program counter (PC) register: This is a 32 -bit register that contains the address of the next instruction to be executed by the processor.

• Decode Unit: This block takes as input some or all of the 32 bits of the instruction and computes the proper control signals to be utilized for other blocks. These signals are generated based on the type and the content of the instruction being executed.

• Register File: This block contains 32 32-bit registers. The register file supports two independent register reads and one register write in one clock cycle. 5 bits are used to address the register file.

• ALU: This block performs operations such as addition, subtraction, comparison, etc. It uses the control signals generated by the Decode Unit, as well as the data from the registers or from the instruction directly. It computes data that can be written into one of the registers (including PC). You will implement this block by referring to the instruction set.
• Instruction and Data Memory (Word-addressable or Byte-addressable): The instruction memory is initialized to contain the program to be executed. The data memory stores the data and is accessed using load word and store word instruction.[1]

Processor Interface and Port Mapping:

- Reset is BTNC
- By default, BTNR is the clock but internal clock can be used by pressing BTNL and BTNU simultaneously
- Program counter is displayed on the seven-segment display by default
- To display data memory and register:
  1) Address is specified using SW (7:3) & BTND
  2) BTNL is pressed for displaying register file and BTNU is used for data memory
- MODES are specified using SW (2:0).
- INPUT depending on the mode is specified using SW (15:8). Only 32 MSB can be specified for ukey, plain text and cipher text

Design and Simulation

The processor component VHDL codes were written out and each of them were verified. They were all added to the FPGA top module after that and the assembly codes were executed on them. The modes are “001” which represents key generation and expansion, “010” which performs encryption and “100” which performs decryption.

The functional and timing simulation was performed after synthesis and implementation and their screenshots are attached below:

Functional Simulation:

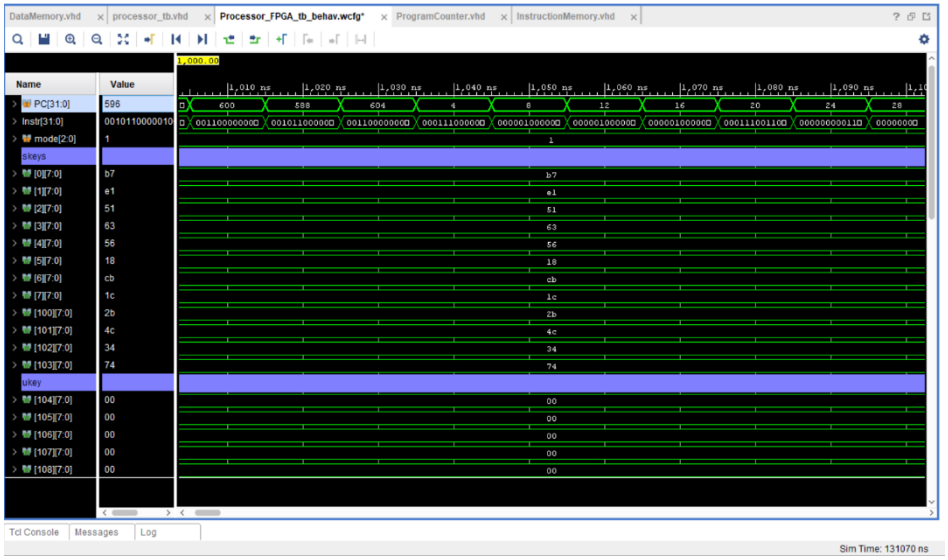


Fig.1: Mode ‘001’-Calculating skey values for the given ukey value:x”00000000000000000000000000000000”.

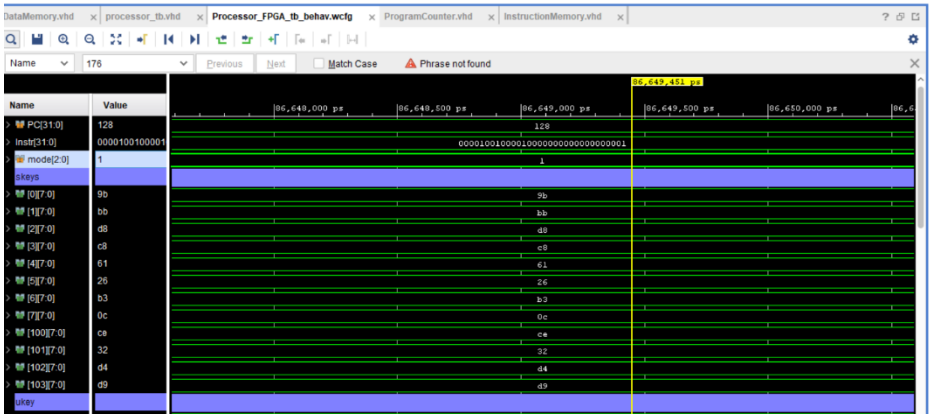


Fig.2: Mode ‘001’-Generated skey values for the given ukey value.

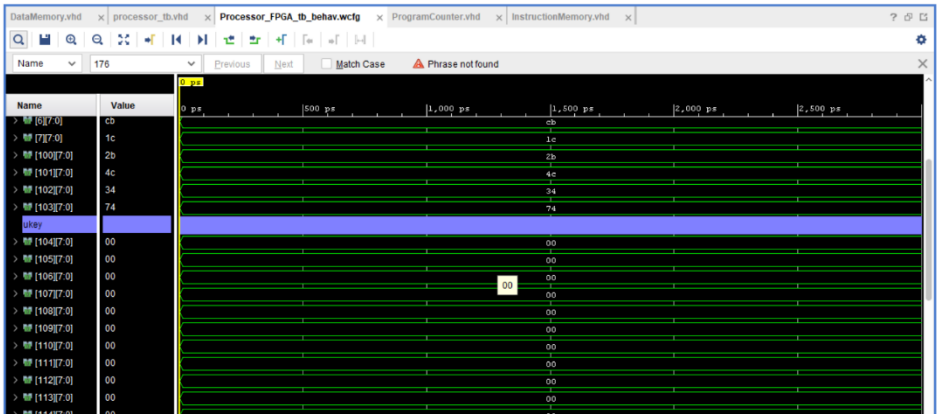


Fig 3: Preserved “ukey” value.

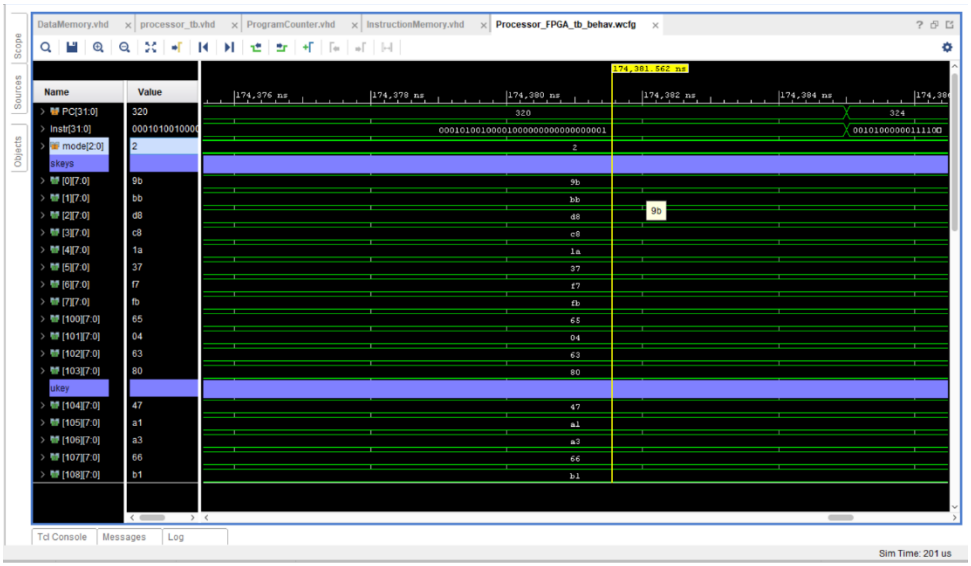


Fig 4: Mode ‘010’-Encryption Program Counter generated.

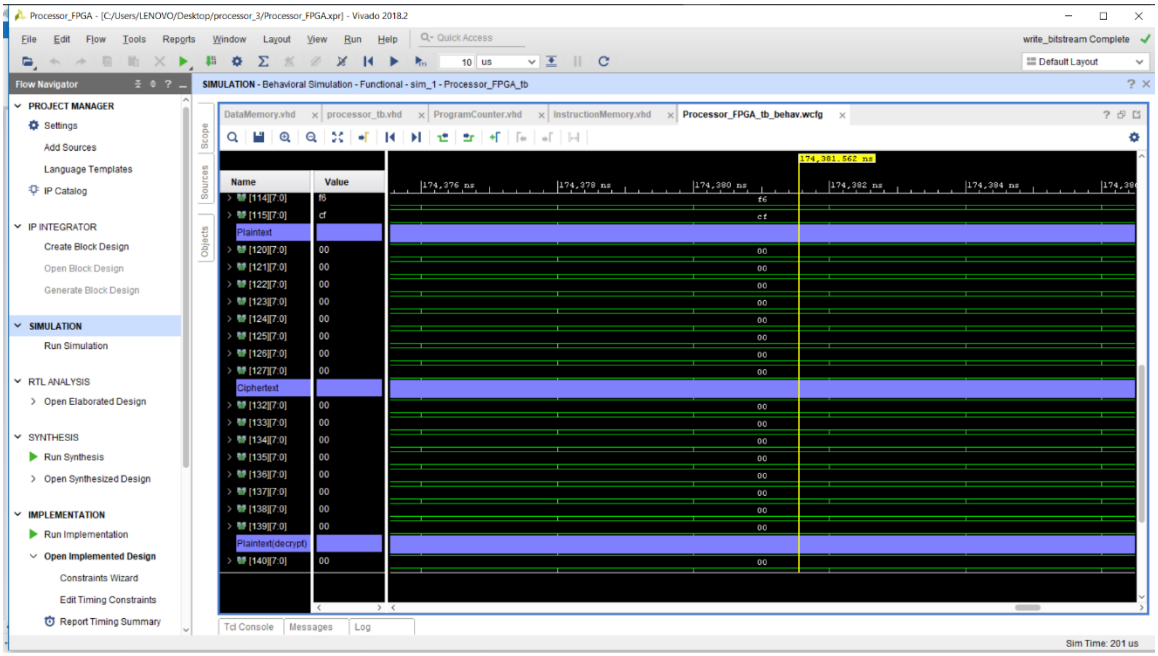


Fig 5: Mode ‘010’-Encryption Input value-1

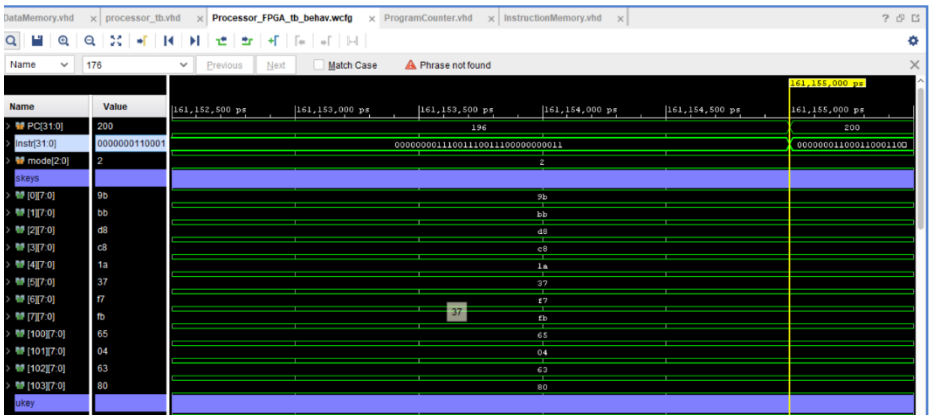


Fig 6: Mode ‘010’-Encryption Program Counter displaying process completion.

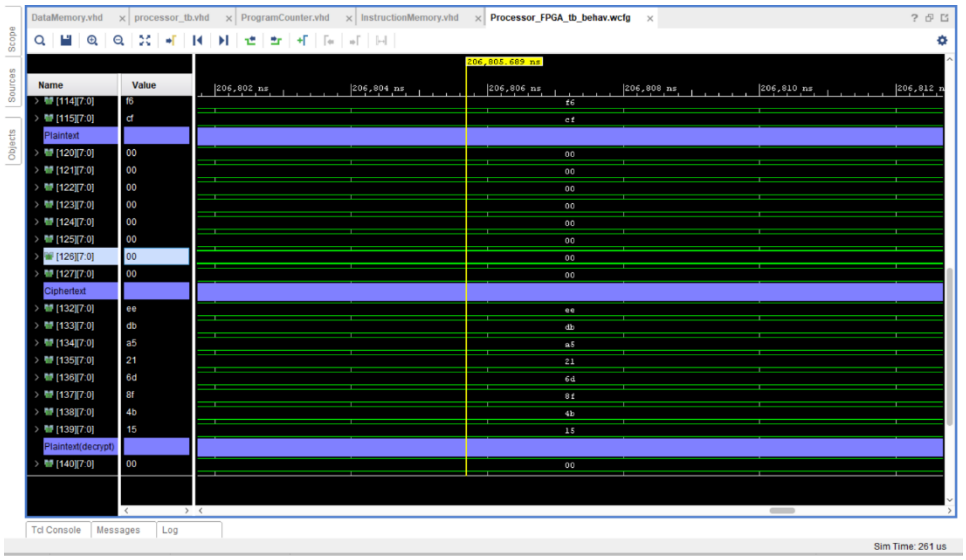


Fig 7: Mode ‘010’-Encryption Program Counter Output-1 generated.

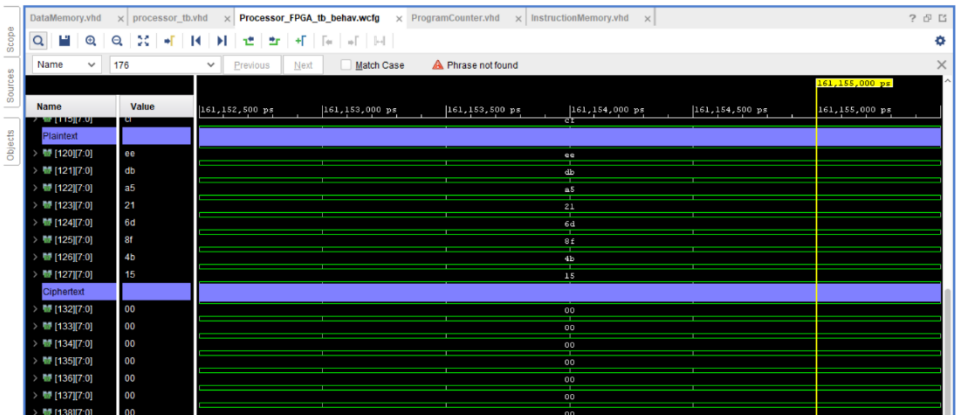


Fig 8: Mode ‘010’-Encryption Input-2.

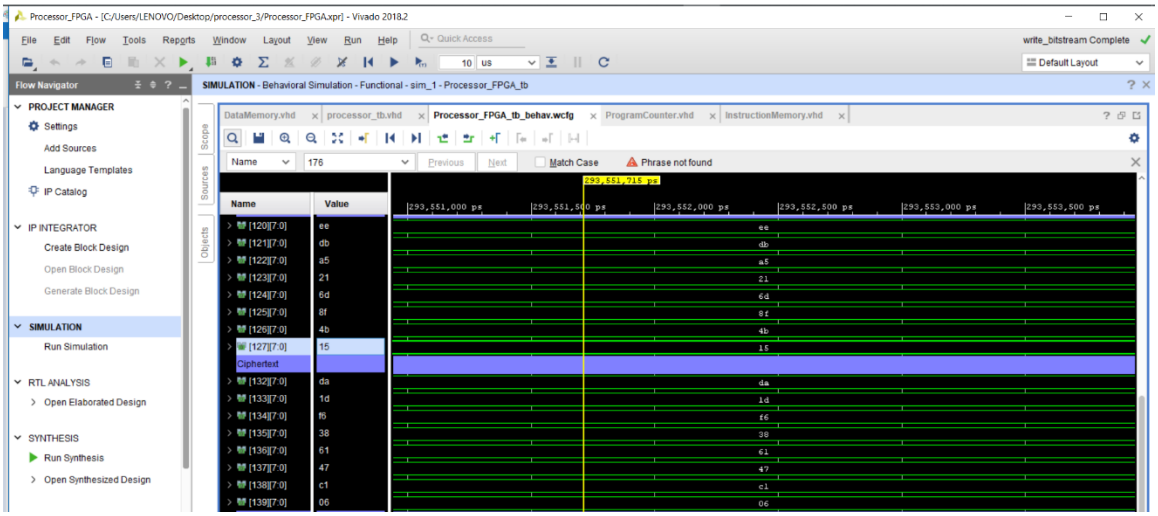


Fig 9: Mode ‘010’-Encryption Output-2 generated.

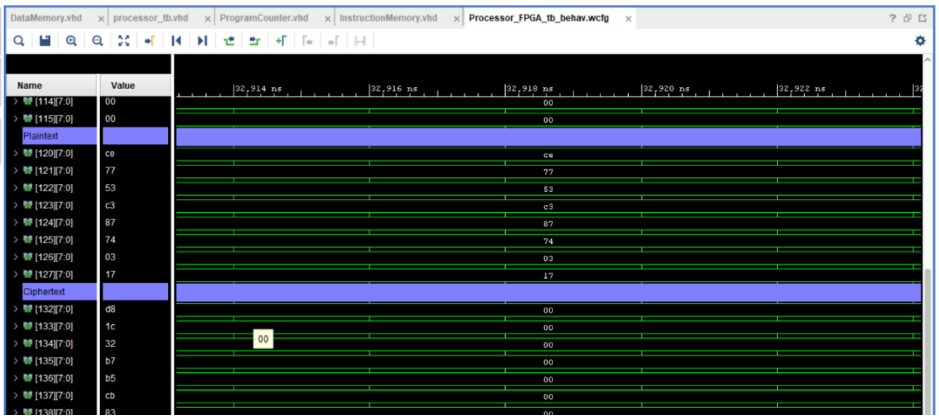


Fig 10: Mode ‘010’-Encryption Input-3.

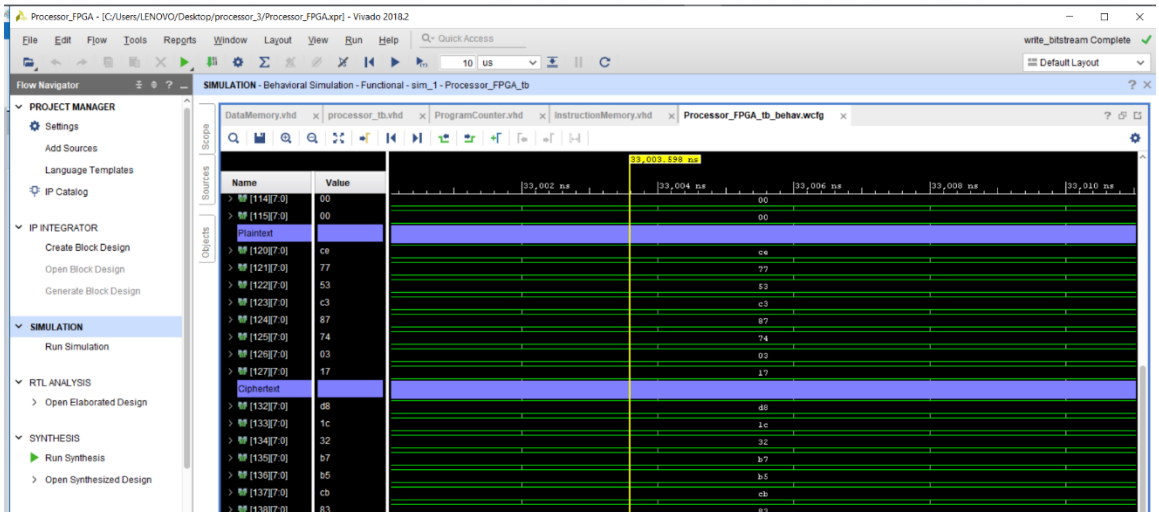


Fig 11: Mode ‘010’-Encryption Output-3.

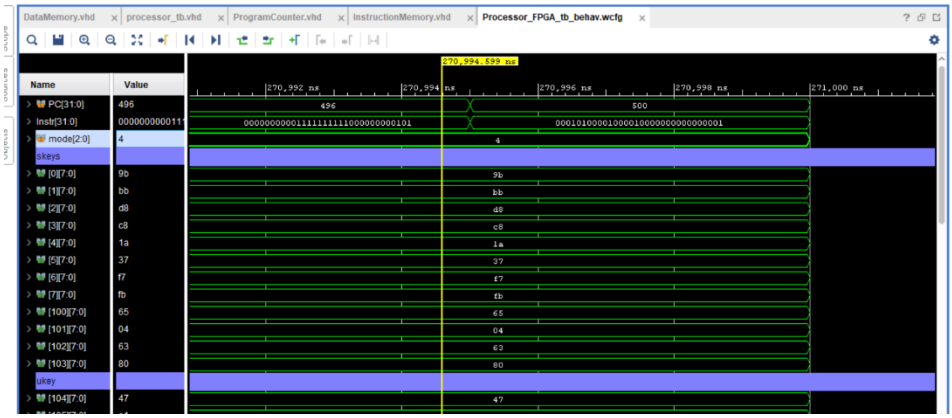


Fig 12: Mode ‘100’-Dencryption Program Counter generated.

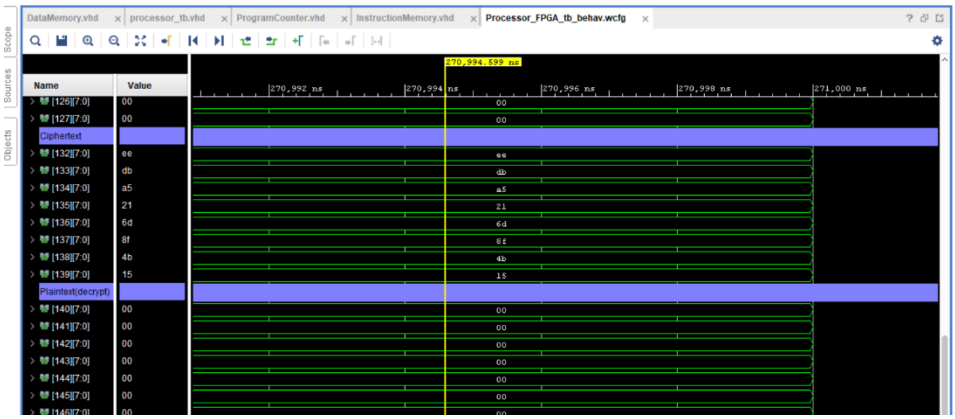


Fig 13: Mode ‘100’-Dencryption Input given.

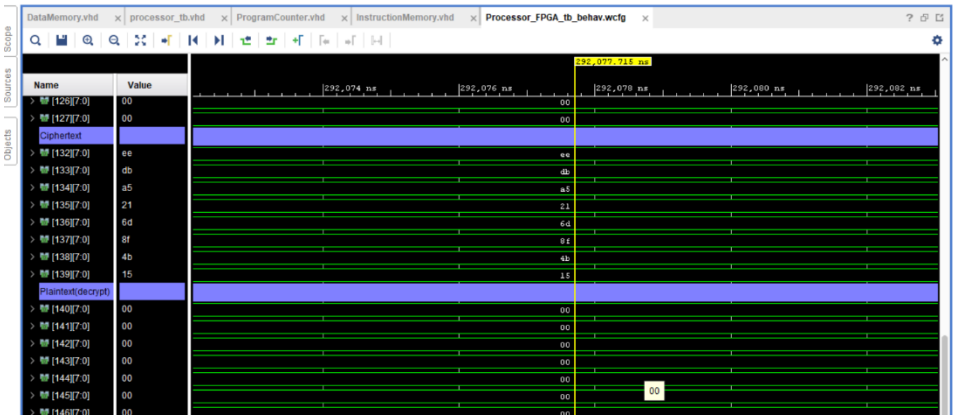


Fig 14: Mode ‘100’-Dencryption Output generated.

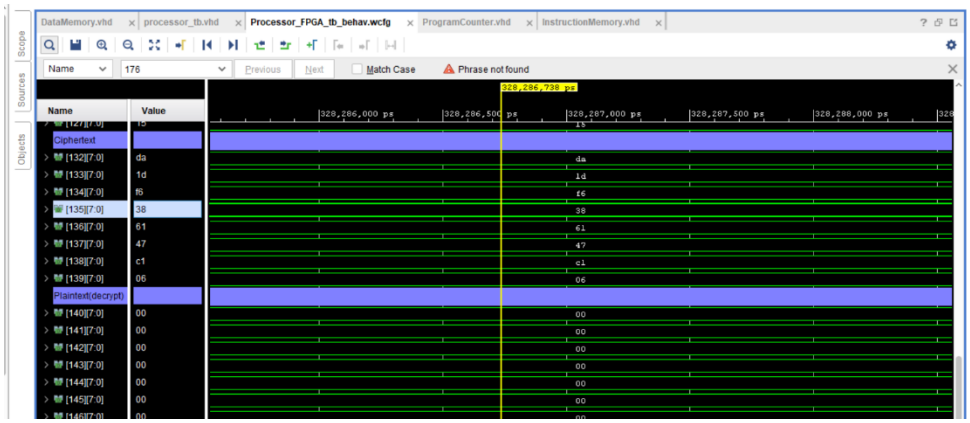


Fig 15: Mode ‘100’-Dencryption Input-2.

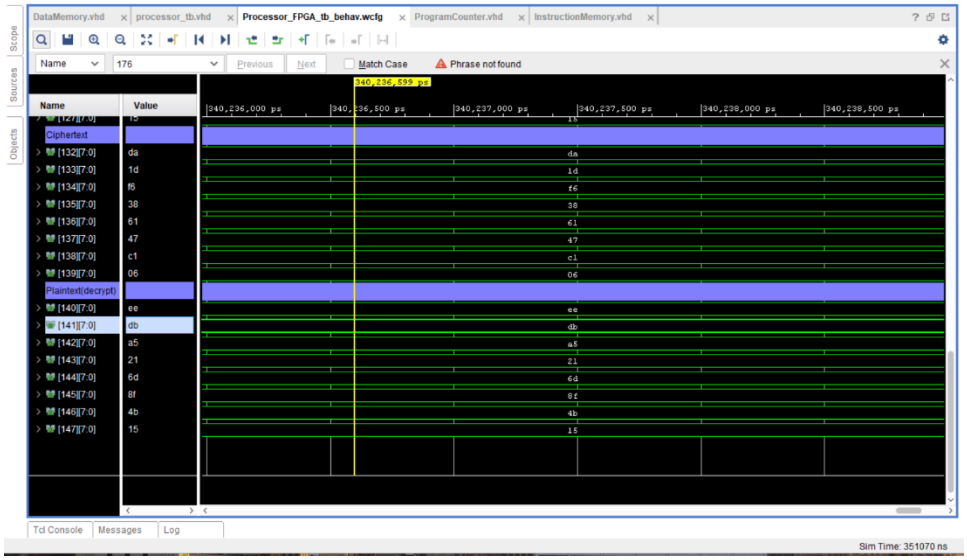


Fig 16: Mode ‘100’-Dencryption Output-2.

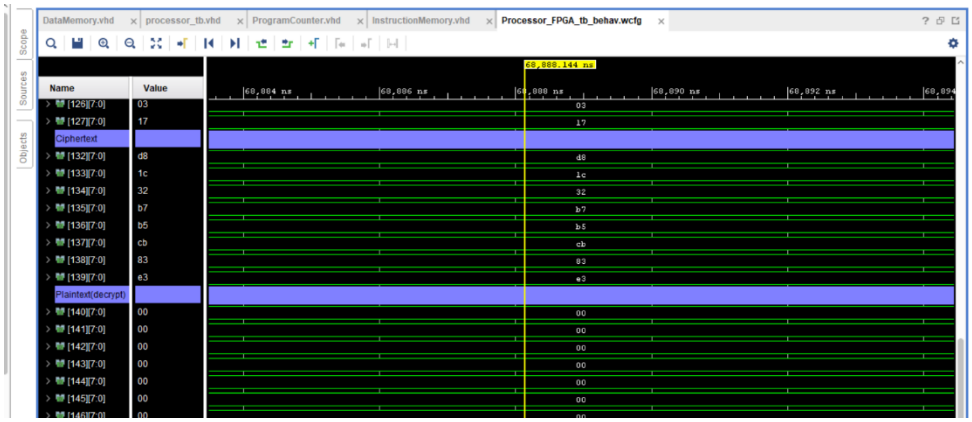


Fig 17: Mode ‘100’-Dencryption Input-3.

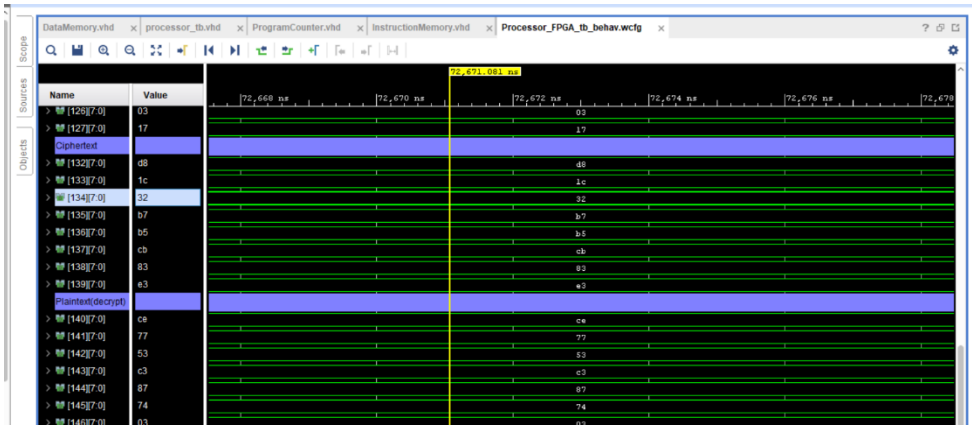


Fig 18: Mode ‘100’-Dencryption Output-3.

Timing Simulation:

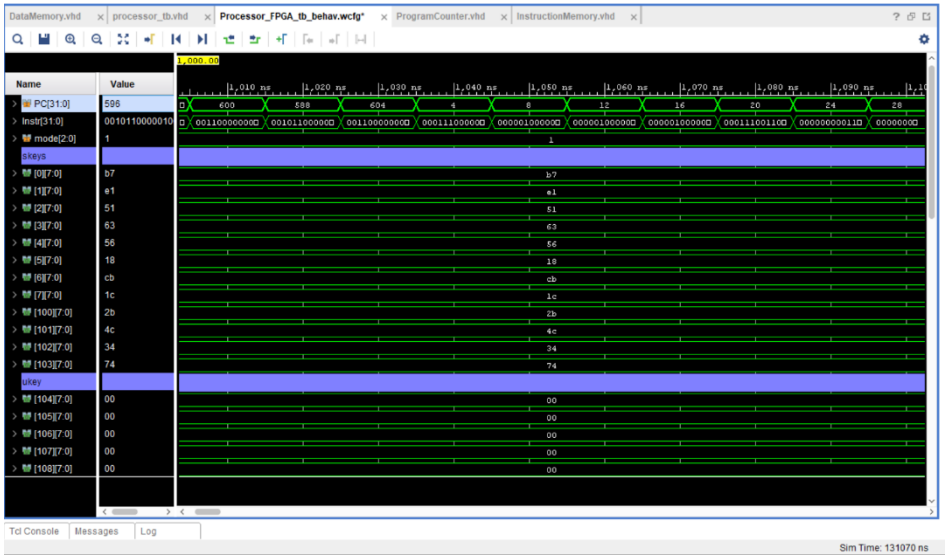


Fig.19: Mode ‘001’-Calculating skey values for the given ukey value:x”000000000000000000000000000000000”.

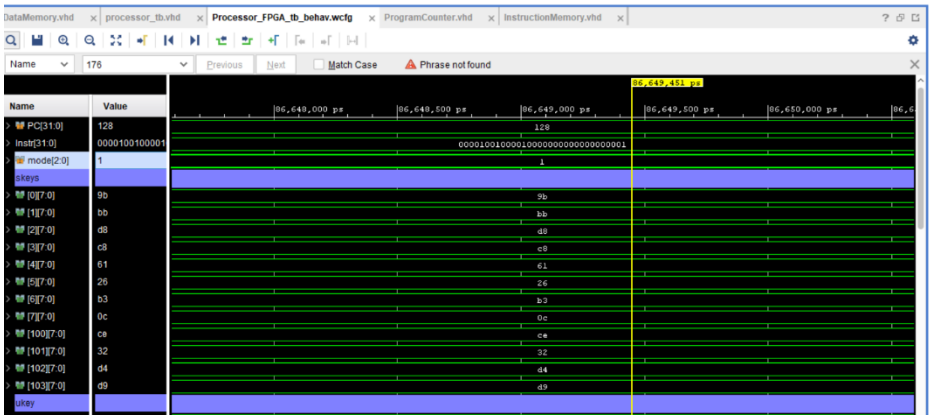


Fig.20: Mode ‘001’-Generated skey values for the given ukey value.

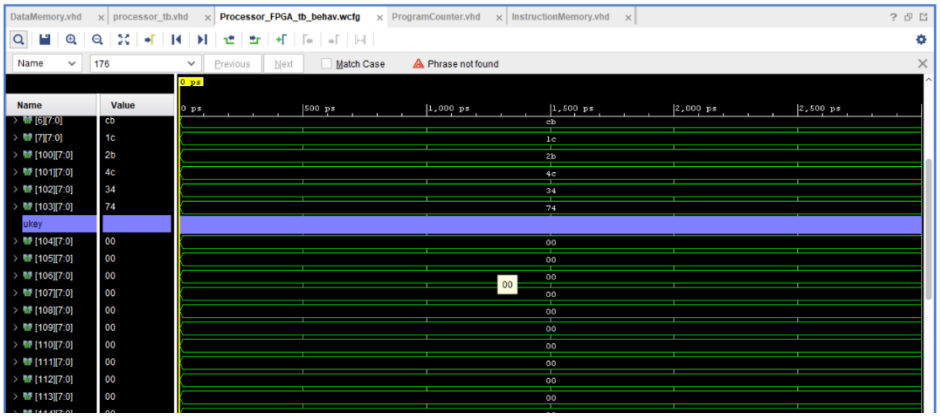


Fig 21: Preserved “ukey” value.

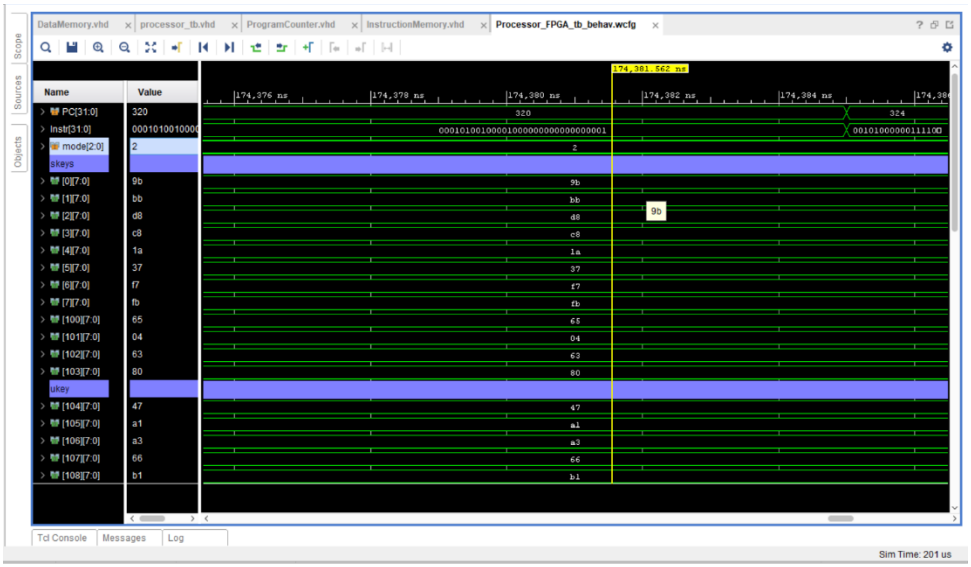


Fig 22: Mode ‘010’-Encryption Program Counter generated.

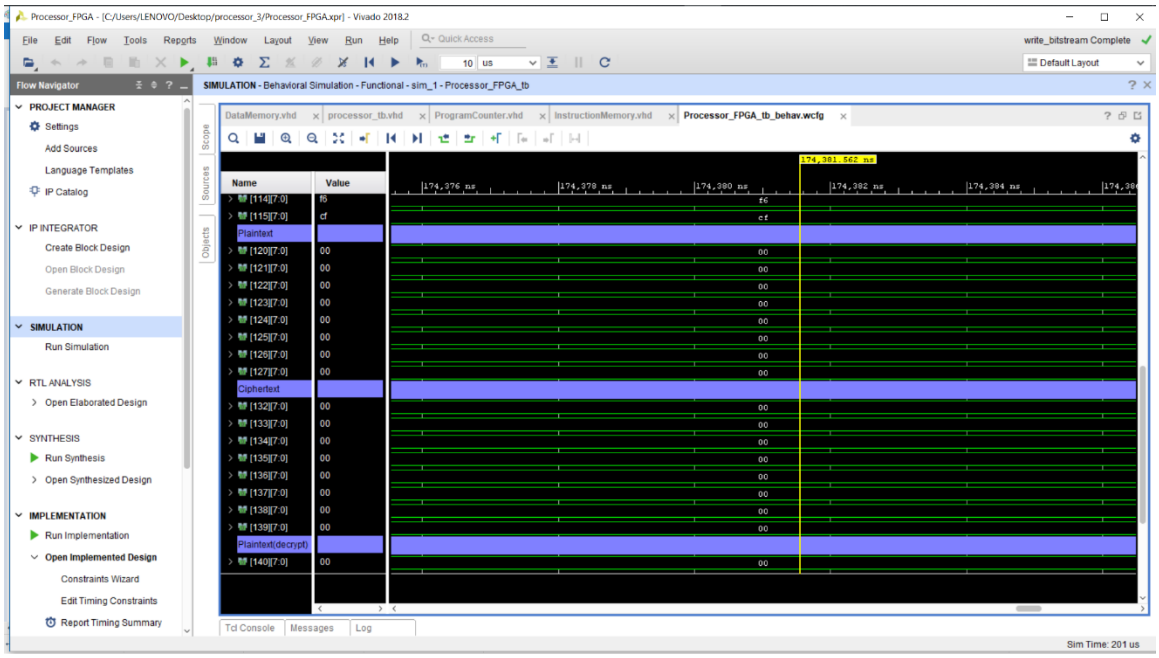


Fig 23: Mode ‘010’-Encryption Input value-1

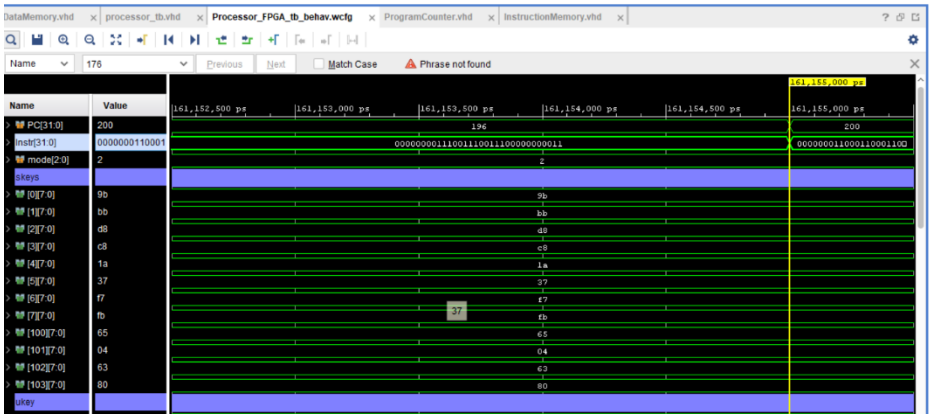


Fig 24: Mode ‘010’-Encryption Program Counter displaying process completion.

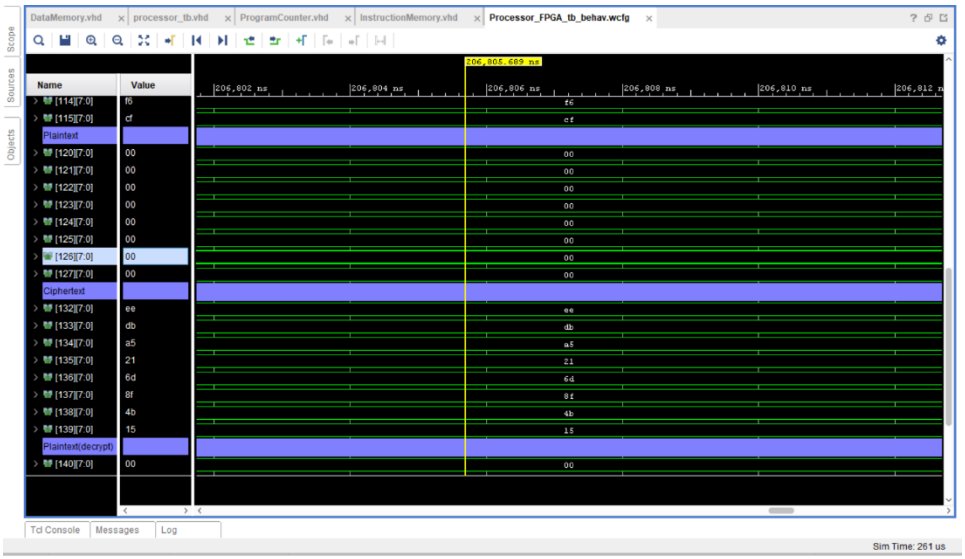


Fig 25: Mode ‘010’-Encryption Program Counter Output-1 generated.

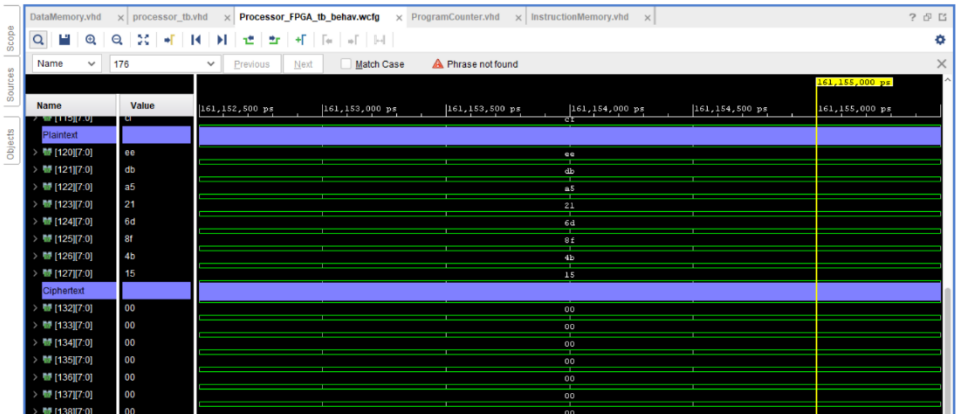


Fig 26: Mode ‘010’-Encryption Input-2.

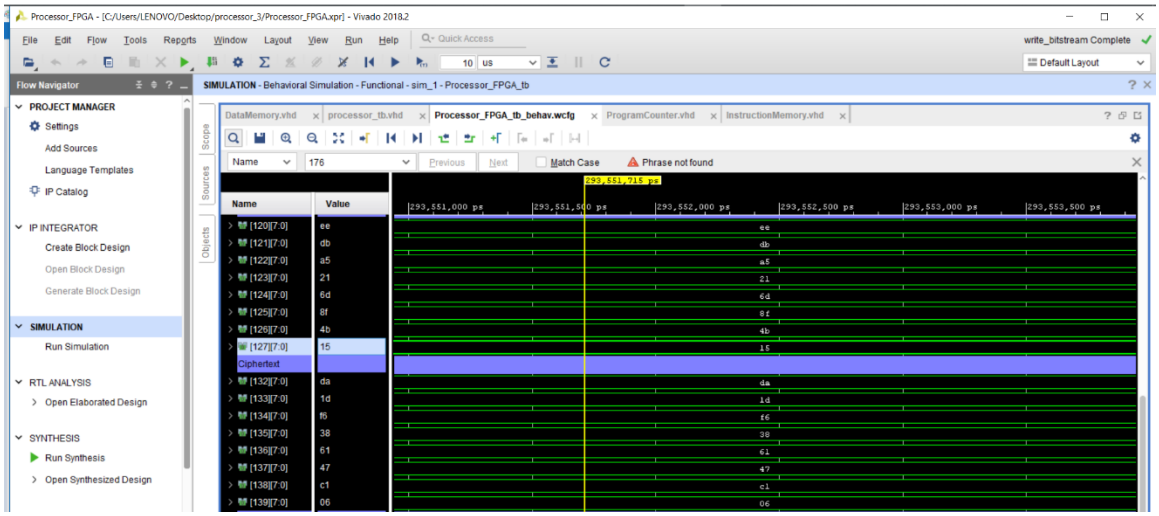


Fig 27: Mode ‘010’-Encryption Output-2 generated.

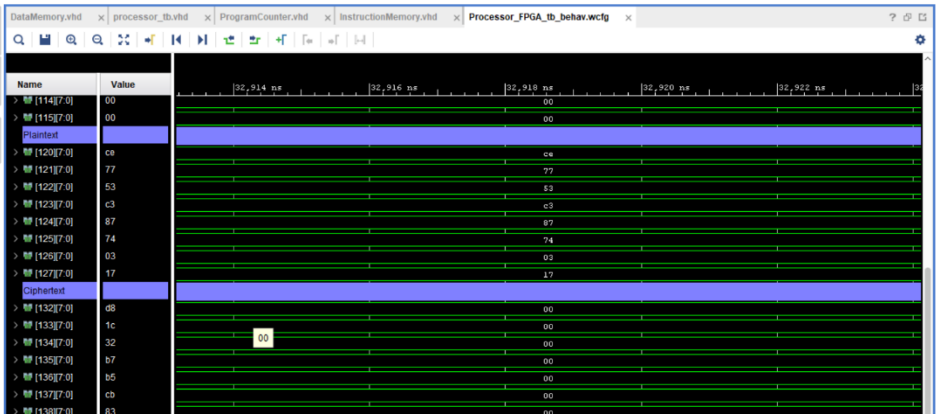


Fig 28: Mode ‘010’-Encryption Input-3.

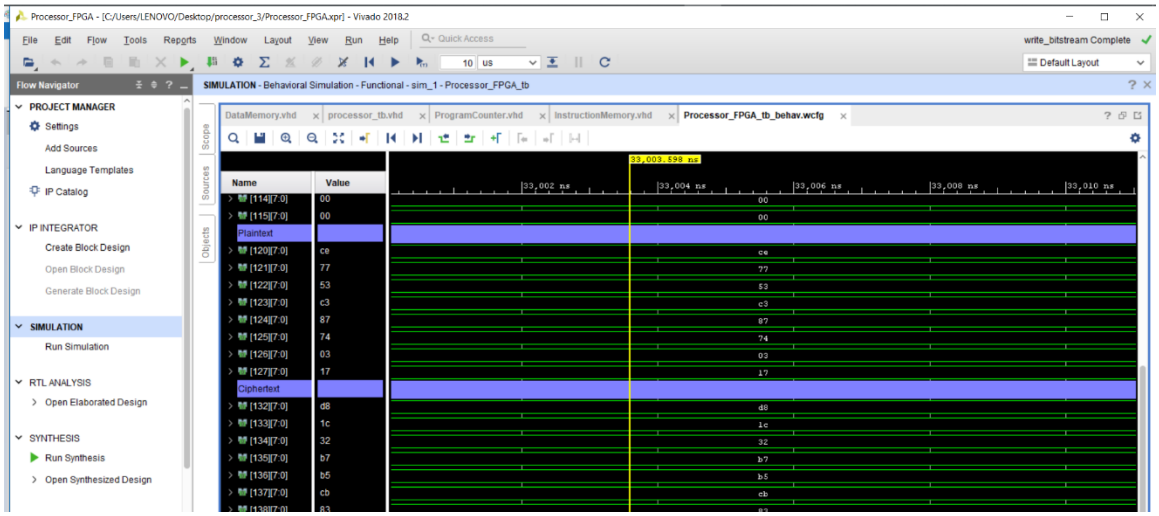


Fig 29: Mode ‘010’-Encryption Output-3.

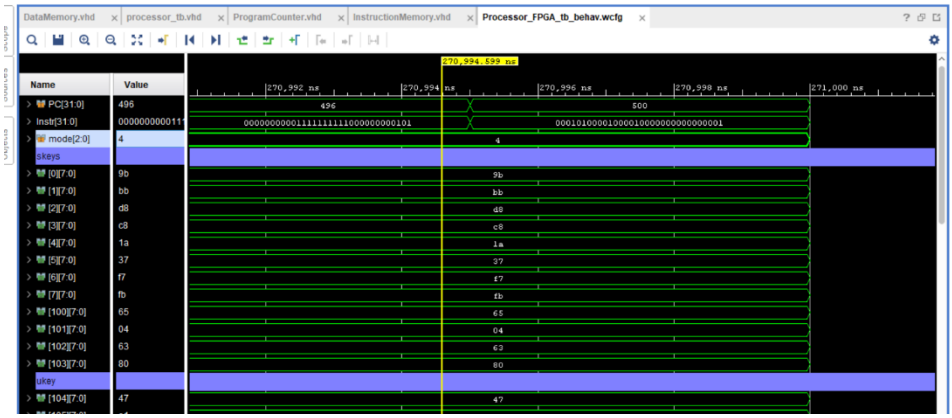


Fig 30: Mode ‘100’-Dencryption Program Counter generated.

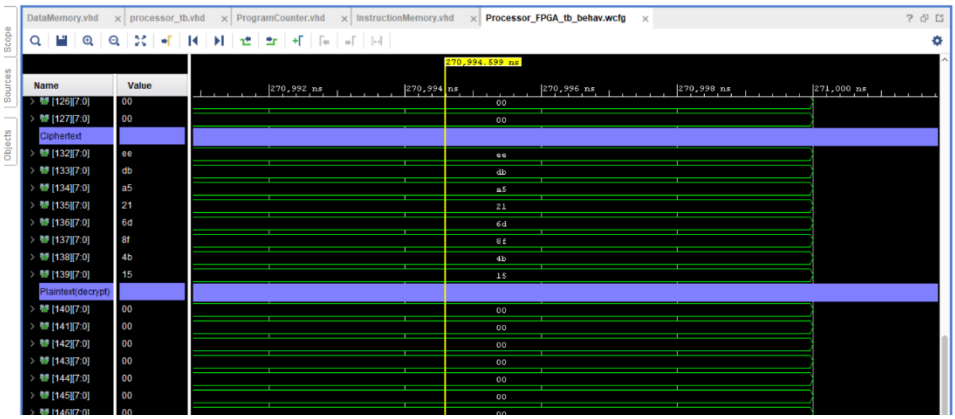


Fig 31: Mode ‘100’-Dencryption Input given.

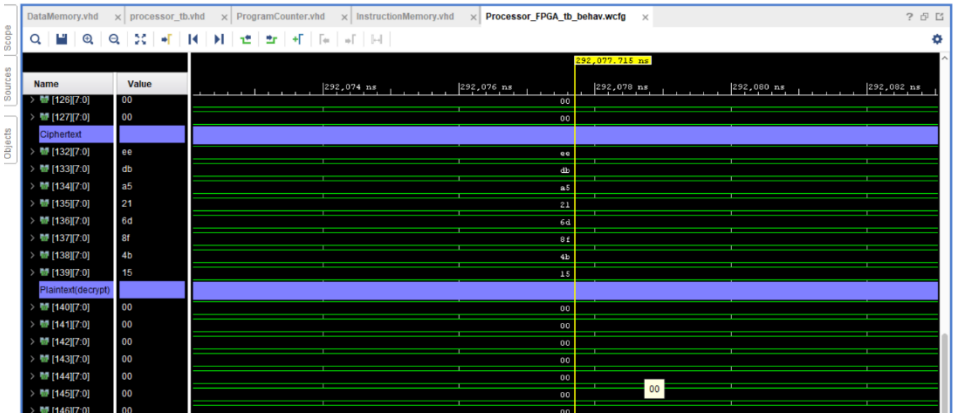


Fig 32: Mode ‘100’-Dencryption Output generated.

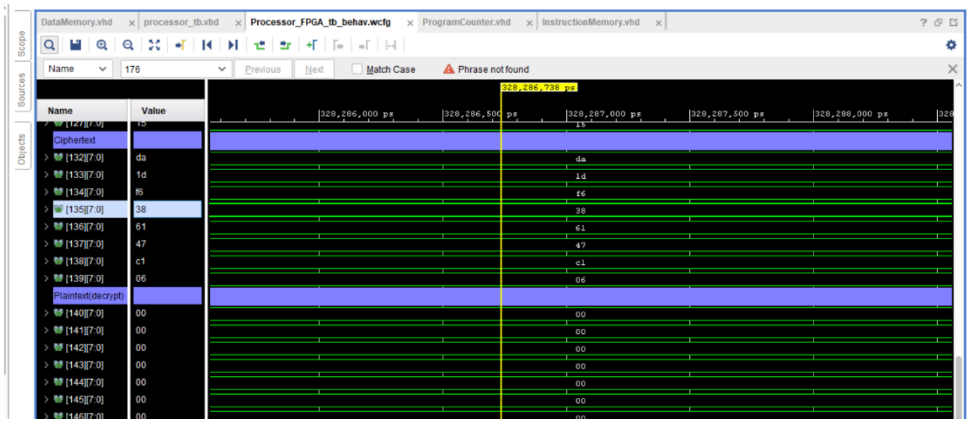


Fig 33: Mode ‘100’-Dencryption Input-2.

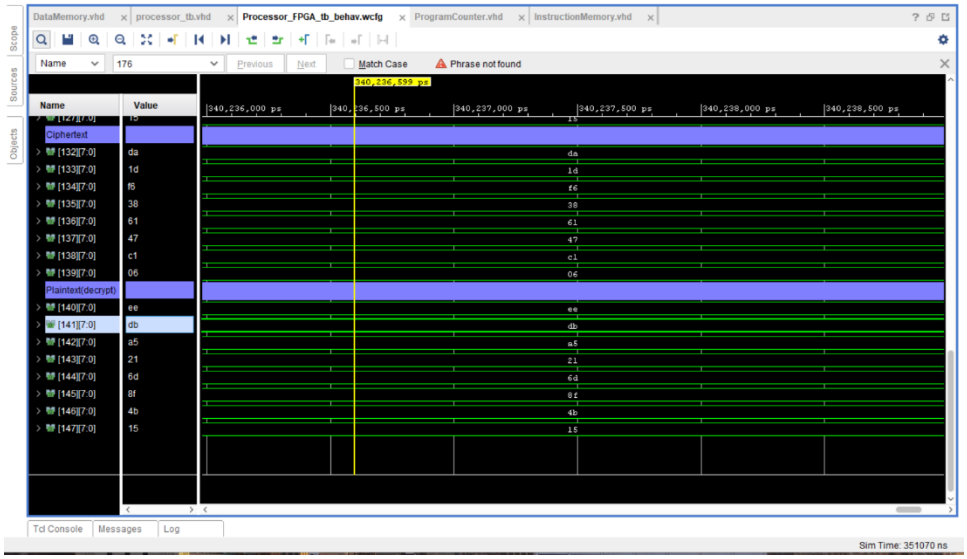


Fig 34: Mode ‘100’-Dencryption Output-2.

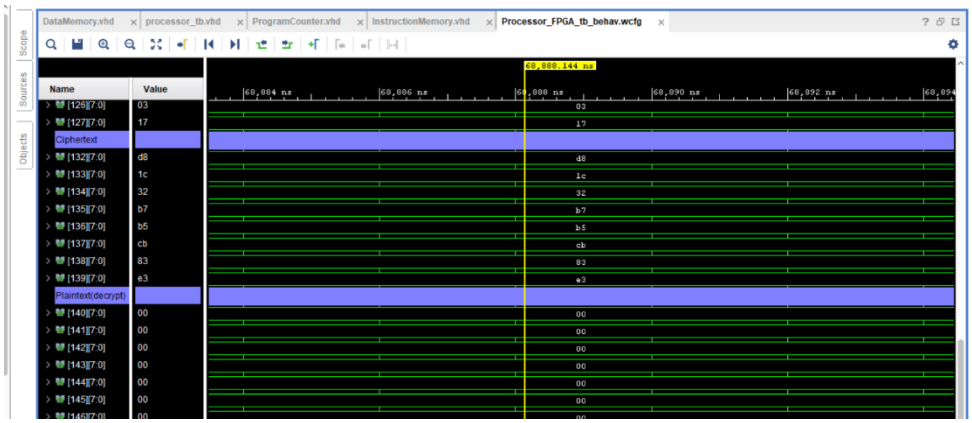


Fig 35: Mode ‘100’-Dencryption Input-3.

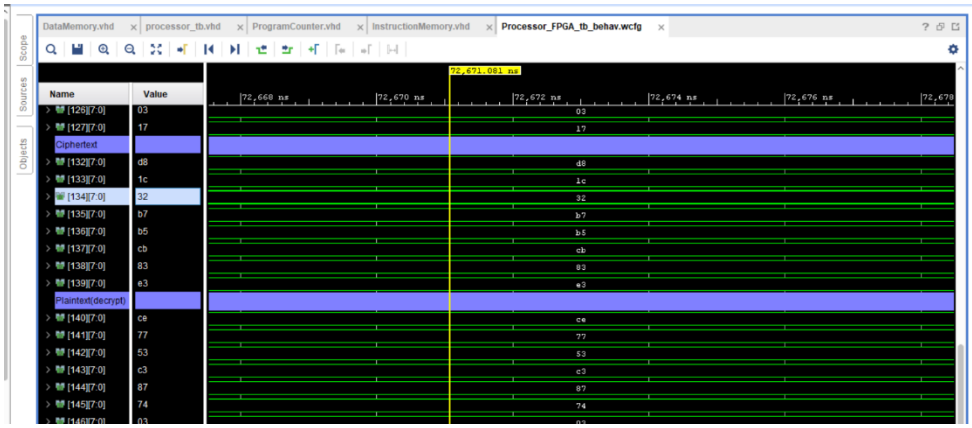


Fig 36: Mode ‘100’-Dencryption Output-3.

Observation on Processor Performance and Area

The performance of the processor after running each assembly code was primarily measured using the clock cycles taken, which are described as: 12,276ns for key expansion/round key generation, 2,786ns for encryption, and 3,175ns for decryption for the input given as “CE7753C387740317” and ukey as “x’00000000000000000000000000000000’.

The resource utilization of post synthesis and post place and route are:

Post synthesis:

Utilization **Post-Synthesis** | Post-Implementation

Graph | **Table**

Resource	Estimation	Available	Utilization...
LUT	5999	63400	9.46
FF	4964	126800	3.91
IO	54	210	25.71
BUFG	2	32	6.25

Post Place-and-route:

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization...
LUT	5998	63400	9.46
FF	4964	126800	3.91
IO	54	210	25.71
BUFG	2	32	6.25

The timing summary generated post synthesis and post place and route are:

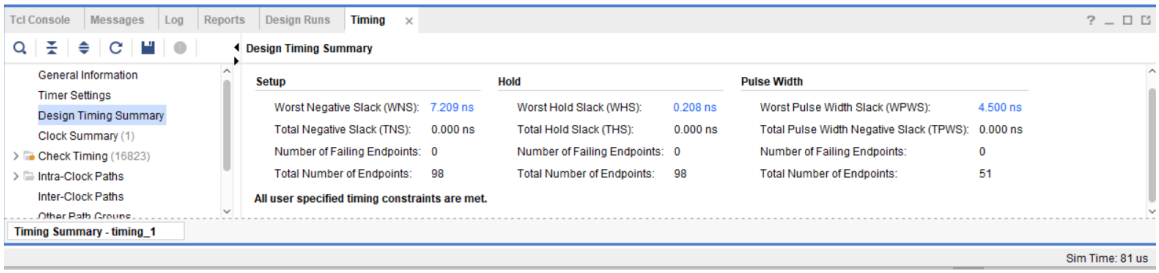


Fig 37: Design Timing Summary Report

Clock constraint given: 10ns

Minimum period: 2.791ns

Maximum frequency gives the speed as: 358.29 MHz

Latency: 6079ns-average.

Encryption, Decryption, and Round Key Generation

The assembly flow of encryption, decryption, and round key generation can be explained by analyzing the code written deeply.

First, we see the INTIAL STATE which is:

- ALL THE REGISTERS HAVE VALUE $(0000000000000000)_{32}$
- DMEM HAS THE INITIAL SKEY VALUES WHICH ARE GENERATED USING THE MAGIC CONSTANTS Pw= 0xB7E15163 AND Qw= 0x9E3779B9

The explanation is done as:

- Round-key generation:

```
1   "00110000","00000000","00000000","10010011",
2   "00011100","00011111","00000000","00100000",
3   "00000100","00001010","00000000","00011010",
4   "00000100","00001011","00000000","00000100",
5   "00000100","00001100","00000000","01001110",
6   "00011100","11000011","00000000","00000000",
```

In the chunk of code from 1-6 we can see different values being loaded in the different register.

We can see that R31 is being loaded with value from the data memory location 32. In the subsequent line of code different values are being stored in register 10,11 and 12 respectively.

In the next chunk of code, the implementation of A = A + S [0]; B = B + S [1]; takes place as shown by the following chunk

```
7   "00000000","01100001","01001000","00000001",
8   "00000001","00100010","01001000","00000001",
```

Here in the 8TH line the register R9 is loaded with the value A+B+S(0). Next, we implemented the shift left operation as shown

```
 9  "00000001","00111111","11110000","00000101", --*and $9, $31, $30
10  "00010101","00101001","00000000","00000001", --shl $9, $9, 1
11  "00101000","00011110","00000000","00000001", --*beq $30, $0, 1 (s
```

The shift left operation is done as follows: in the 9 th instruction R31 is and with R9 and the first digit of the and is stored in R30. Then R9 is shifted left by one bit. Then following that,

the value of R30 is compared with the value in R0 and if the value is equal then the program counter skips one block of code. If not, then the following code is implemented where the R9(shifted value) is added with on bit.

```
12  "00000101","00101001","00000000","00000001", --*addi $9, $9, 1
```

Then after this the following command restored the lost bit in R30

```
13  "00000001","00111111","11110000","00000101", --*and $9, $31, $30
```

```
14  "00010101","00101001","00000000","00000001", --shl $9, $9, 1
15  "00101000","00011110","00000000","00000001", --*beq $30, $0, 1 (
16  "00000101","00101001","00000000","00000001", --*addi $9, $9, 1
17  "00000001","00111111","11110000","00000101", --*and $9, $31, $30
18  "00010101","00101001","00000000","00000001", --shl $9, $9, 1
19  "00101000","00011110","00000000","00000001", --*beq $30, $0, 1 (
20  "00000101","00101001","00000000","00000001", --*addi $9, $9, 1
21  "00000000","00001001","00001000","00000001", --add $0, $9, $1
```

In the above code again, multiple shift lefts are done and the value in R9 is changed post every shift left.

```
22  "00100000","11000001","00000000","00000000", --sw $6, $1, 0
23  "00000000","00100010","01000000","00000001", --add $1, $2, $8
24  "00001101","00001000","00000000","00011111", --andi $8, $8, 31
25  "00011100","11100100","00000000","00011010", --lw $7, $4, 26
26  "00000000","10000001","01001000","00000001", --add $4, $1, $9
27  "00000001","00100010","01001000","00000001", --add $9, $2, $9
28  "00101000","00001000","00000000","00000110", --beq $0, $8, 6 (
```

In the above command the ukey (0) is stored in R4 and subsequently the values of A and B are summed with key (0). In the next command A+B+ukey (0) is left shifted by one using the logic of SHL implemented above.

```
30  "00010101","00101001","00000000","00000001", --shl $9, $9, 1
31  "00101000","00011110","00000000","00000001", --*beq $30, $0, 1
32  "00000101","00101001","00000000","00000001", --*addi $9, $9, 1
```

In the set of commands below i and j counter values are uploaded in register R6 and R7 respectively.

```
34  "00101100","00001000","00000000","00001100", --bne $0, $8, 12
35  "00000000","00001001","00010000","00000001", --add $0, $9, $2
36  "00100000","11100010","00000000","00011010", --sw $7, $2, 26
37  "00000100","11000110","00000000","00000001", --addi $6, $6, 1
38  "00000100","11100111","00000000","00000001", --addi $7, $7, 1
```

In the following chunk of code does the looping and the generation of the rest of the Skey values and implements the complete round key generation

```
39   "00101100","11001010","00000000","00000001",
40   "00000000","11000110","00110000","00000011",
41   "00101100","11101011","00000000","00000001",
42   "00000000","11100111","00111000","00000011",
43   "00000100","10100101","00000000","00000001",
44   "00101100","10101100","00000000","00000001",
45   "00110000","00000000","00000000","10011010",
46   "00110000","00000000","00000000","00000101",
47   "00110000","00000000","00000000","00011011",
```

- ENCRYPTION

```
48  "00000000","10100101","00101000","00000011", --*sub $5, $5, $5  --
49  "00000000","11000110","00110000","00000011", --*sub $6, $6, $6
50  "00000000","11100111","00111000","00000011", --*sub $7, $7, $7
51  "00000001","10001100","01100000","00000011", --*sub $12, $12, $12
```

The above code is used to set the value of different registers to 0 so the if the modules is reset in between then the output doesn't produce any errors.

```
52  "00011100","00000001","00000000","00011110", --*lw $0, $1, 30
53  "00011100","00000010","00000000","00011111", --*lw $0, $2, 31
54  "00011100","00011111","00000000","00100000", --*lw $0, $31, 32
55  "00011100","00000011","00000000","00000000", --lw $0, $3, 0
56  "00011100","00000100","00000000","00000001", --lw $0, $4, 1
57  "00000000","00100011","00001000","00000001", --add $1, $3, $1
58  "00000000","01000100","00010000","00000001", --add $2, $4, $2
59  "00000100","00001100","00000000","00001100", --addi $0, $12, 12
```

In the above code R3 is loaded with S (0), R (4) is loaded with S (1), R1 is loaded with A+S(0) and R2 is loaded with B+S(1). Next R12 is stored with the value 12 for running the loop for the given function:

for i = 1 to 12 do

A = ((A xor B) <<< B) + S [2×i];

B = ((B xor A) <<< A) + S [2×i + 1];

```
60  "00000000","00100010","00100000","00000111", --or $1, $2, $4
61  "00000000","00100010","00101000","00000101", --and $1, $2, $5
62  "00000000","10000101","00100000","00000011", --sub $4, $5, $4
63  "00001100","01000101","00000000","00011111", --andi $2, $5, 31
```

In the above command A XOR B is implemented using OR, AND and SUB. In the line the last 5 bits of B are put in R5.

```
64  "00000100","11000110","00000000","00000001", --addi $6, $6, 1
```

This command is used to input the value of i (for generating the Skey [2*i] and Skey[

2*I +1])

```
65  "00101000","00000101","00000000","00000110",
66  "00000000","10011111","11110000","00000101",
67  "00010100","10000100","00000000","00000001",
68  "00101000","00011110","00000000","00000001",
69  "00000100","10000100","00000000","00000001",
70  "00001000","10100101","00000000","00000001",
71  "00101100","00000101","00000000","00010101",
```

The above code is used to implement the SHIFT function for

Register A required in the given function. And the code below shows the SHIFT function for Register B

```
72  "00010100","11000111","00000000","00000001", --SHL $6, $7, 1
73  "00011100","11100011","00000000","00000000", --LW $7, $3, 0
74  "00000000","10000011","00001000","00000001", --ADD $4, $3, $1
75  "00000000","00100010","00100000","00000111", --OR $1, $2, $4
76  "00000000","00100010","00101000","00000101", --and $1, $2, $5
77  "00000000","10000101","00100000","00000011", --sub $4, $5, $4
78  "00001100","00100101","00000000","00011111", --andi $1, $5, 31
79  "00101000","00000101","00000000","00000110", --beq $0, $5, 6 (
80  "00000000","10011111","11110000","00000101", --*and $4, $31, $
81  "00010100","10000100","00000000","00000001", --shl $4, $4, 1
82  "00101000","00011110","00000000","00000001", --*beq $30, $0, 1
83  "00000100","10000100","00000000","00000001", --*addi $4, $4, 1
84  "00001000","10100101","00000000","00000001", --subi $5, $5, 1
85  "00101100","00000101","00000000","00001000", --bne $0, $5, 8
```

In the following code the value S[2×i + 1] is implemented and the values of A and B are stored back in the data memory.

```
86  "00000100","11100111","00000000","00000001", --addi $7, $7, 1
87  "00011100","11100011","00000000","00000000", --lw $7, $3, 0
88  "00000000","10000011","00010000","00000001", --add $4, $3, $2
89  "00101101","10000110","00000000","00000101", --bne $12, $6, 5 (
90  "00100000","00000001","00000000","00011110", --sw $0, $1, 30
91  "00100000","00000010","00000000","00011111", --sw $0, $2, 31
```

```
92  "00110000","00000000","00000000","10011101",
93  "00110000","00000000","00000000","01000000",
94  "00110000","00000000","00000000","01001110",
95  "00110000","00000000","00000000","00111011",
```

These commands are for switching in between the different modes.

- Decryption

```
96  "00000001","10001100","01100000","00000011", --*sub $12, $12, $12
97  "00000011","10111101","11101000","00000011", --*sub $29, $29, $29
98  "00000100","00011101","00000000","00100000", --*addi $0, $29, 32
99  "00011100","00000001","00000000","00011110", --*lw $0, $1, 30
100 "00011100","00000010","00000000","00011111", --*lw $0, $2, 31
101 "00011100","00011111","00000000","00100000", --*lw $0, $31, 32
102 "00000100","00001100","00000000","00001100", --addi $0, $12, 12
103 "00010101","10001101","00000000","00000001", --shl $12, $13, 1
104 "00000101","10101110","00000000","00000001", --addi $13, $14, 1
105 "00011101","11000011","00000000","00000000", --lw $14, $3, 0
106 "00000000","01000011","00010000","00000011", --sub $2, $3, $2
107 "00001100","00100100","00000000","00011111", --andi $1, $4, 31
```

So, the above code first clears the values in R12 and R29, so if any sudden reset is done, we don’t obtain erroneous outputs.

```
"00000011","10100100","00100000","00000011", --*sub $29, $4, $4
"00101011","10100100","00000000","00000110", --beq $29, $4, 6 (
"00000000","01011111","11110000","00000101", --*and $2, $31, $30
"00010100","01000010","00000000","00000001", --shl $2, $2, 1
"00101000","00011110","00000000","00000001", --*beq $30, $0, 1 (
"00000100","01000010","00000000","00000001", --*addi $2, $2, 1
"00001000","10000100","00000000","00000001", --subi $4, $4, 1
"00101100","00000100","00000000","00011101", --bne $0, $4, 29 (
"00000000","00100010","00101000","00000111", --or $1, $2, $5
"00000000","00100010","00110000","00000101", --and $1, $2, $6
"00000000","10100110","00010000","00000011", --sub $5, $6, $2
"00000000","01100011","00011000","00000011", --sub $3, $3, $3
"00011101","10100011","00000000","00000000", --lw $13, $3, 0
"00000000","00100011","00001000","00000011", --sub $1, $3, $1
"00001100","01000100","00000000","00011111", --andi $2, $4, 31
```

In the above code the function ((B-S(25))>>>A) XOR A) is implemented using OR, AND and SUB. Also the following code shows the function ((A-S(24))>>>B) XOR B) which is implemented using OR, AND and SUB .

```
123  "00000011","10100100","00100000","00000011", --*sub $29, $4, $4
124  "00101011","10100100","00000000","00000110", --beq $29, $4, 6 (o
125  "00000000","00111111","11110000","00000101", --*and $1, $31, $30
126  "00010100","00100001","00000000","00000001", --shl $1, $1, 1
127  "00101000","00011110","00000000","00000001", --*beq $30, $0, 1 (
128  "00000100","00100001","00000000","00000001", --*addi $1, $1, 1
129  "00001000","10000100","00000000","00000001", --subi $4, $4, 1
130  "00101100","00000100","00000000","00001111", --bne $0, $4, 15 (j
131  "00000000","00100010","00101000","00000111", --or $1, $2, $5
132  "00000000","00100010","00110000","00000101", --and $1, $2, $6
133  "00000000","10100110","00001000","00000011", --sub $5, $6, $1
134  "00001001","10001100","00000000","00000001", --subi $12, $12, 1
135  "00101101","10000000","00000000","00001011", --bne $12, $0, 11 (
136  "00000000","01100011","00011000","00000011", --sub $3, $3, $3
137  "00011100","00000011","00000000","00000001", --lw $0, $3, 1
138  "00000000","01000011","00010000","00000011", --sub $2, $3, $2
```

The following commands are used to check whether the loop has run 12 times, if not then the loop runs for one more iteration. if the loop is complete then the values of A and B is stored into the data memory. The last line halt which is replaced with jump.

```
134  "00001001","10001100","00000000","00000001", --subi $12, $12, 1
135  "00101101","10000000","00000000","00001011", --bne $12, $0, 11 (jmp
136  "00000000","01100011","00011000","00000011", --sub $3, $3, $3
137  "00011100","00000011","00000000","00000001", --lw $0, $3, 1
138  "00000000","01000011","00010000","00000011", --sub $2, $3, $2
139  "00000000","01100011","00011000","00000011", --sub $3, $3, $3
140  "00011100","00000011","00000000","00000000", --lw $0, $3, 0
141  "00000000","00100011","00001000","00000011", --sub $1, $3, $1
142  "00100000","00000001","00000000","00011110", --sw $0, $1, 30
143  "00100000","00000010","00000000","00011111", --sw $0, $2, 31
144  "00110000","00000000","00000000","10100000", --hal REPLACED WITH JU
```

The last bit the code is used for switching between the different modes, by changing the values of the registers R16, R17 and R18. This switching is also explained in the port mapping as mentioned in the project report.

```
145  "00110000","00000000","00000000","01101100", --jmp 108 (beq $29,
146  "00110000","00000000","00000000","01111011", --jmp 123 (beq $29,
147  "00110000","00000000","00000000","01100110", --jmp 102 (shl $12,
148  "00101100","00010000","00000000","00000011", --**bne $0, $16, 3
149  "00101100","00010001","00000000","00000011", --**bne $0, $17, 3
150  "00101100","00010010","00000000","00000011", --**bne $0, $18, 3
151  "00110000","00000000","00000000","10010011", --**jmp instr 147
152  "00110000","00000000","00000000","00000001", --**jmp to key expa
153  "00110000","00000000","00000000","00101111", --**jmp to encrypti
154  "00110000","00000000","00000000","01011111", --**jmp to decrypti
155  "00101100","00010000","00000000","00000001", --**bne $16, $0, 1
156  "00110000","00000000","00000000","10010011", --**jmp instr 147
157  "00110000","00000000","00000000","10011010", --**jmp 154loop fai
158  "00101100","00010001","00000000","00000001", --**bne $17, $0, 1
159  "00110000","00000000","00000000","10010011", --**jmp instr 147
160  "00110000","00000000","00000000","10011101", --**jmp 157 loop fa
161  "00101100","00010010","00000000","00000001", --**bne $18, $0, 1
162  "00110000","00000000","00000000","10010011", --**jmp instr 147
163  "00110000","00000000","00000000","10100000"); --**jmp 160 loop f
```

Verification using Testbench

The outputs of the encryption, decryption, and round key generation was verified using a testbench file that automatically reads inputs from a file and compares the outputs with the mentioned outputs in another file. Through this method, it enabled us not only to verify 1000 test cases for functional simulation and 1000 cases for timing simulation, but

also enabled us to fix any errors that were generated during the production of the generated outputs.

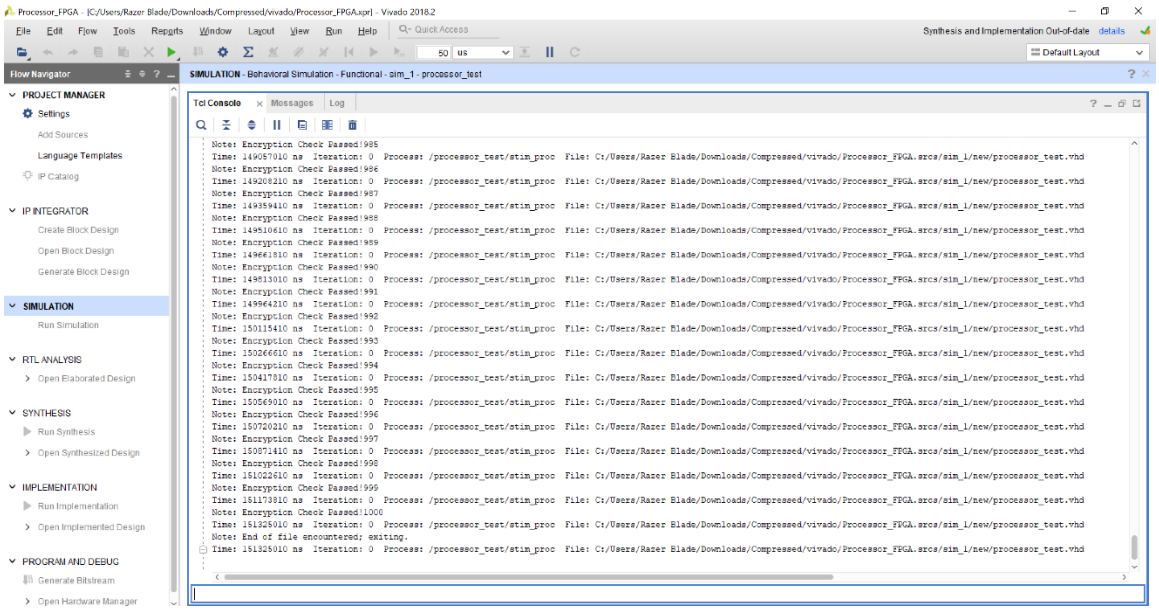


Fig 38: Encryption Check Passed! For 1000 test cases- Functional Simulation.

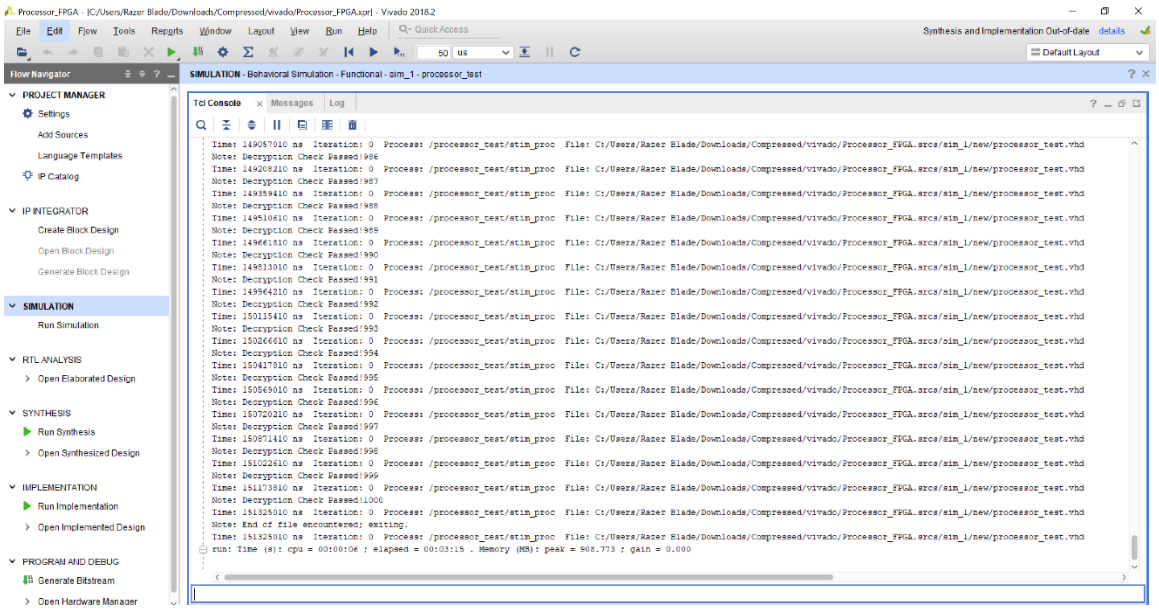


Fig 39: Decryption Check Passed! For 1000 test cases- Functional Simulation.

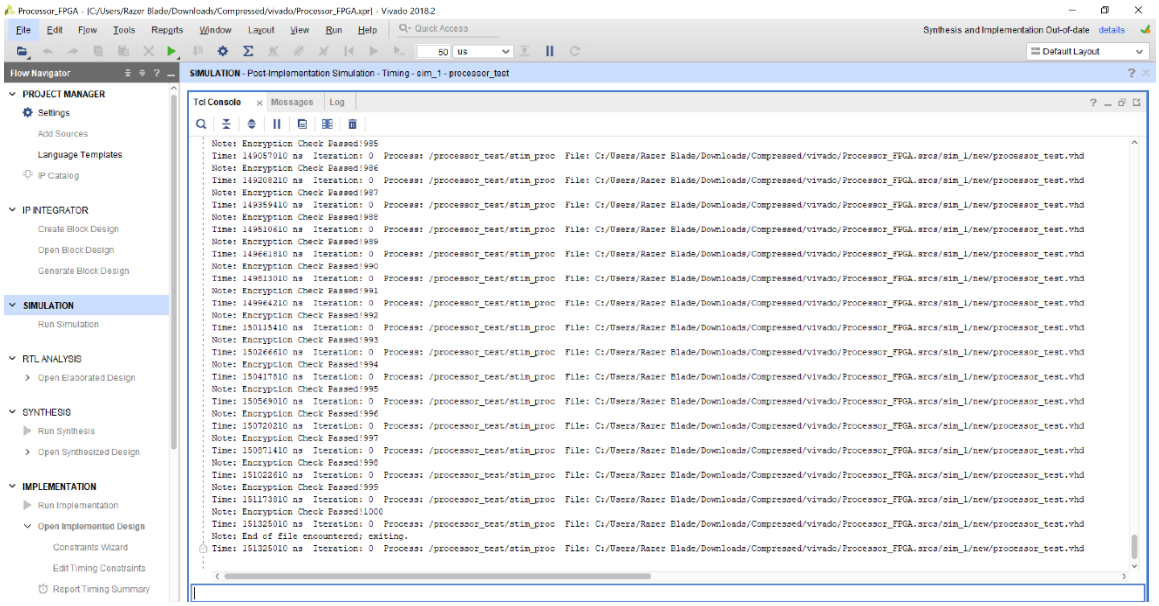


Fig 40: Encryption Check Passed! For 1000 test cases-Timing Simulation.

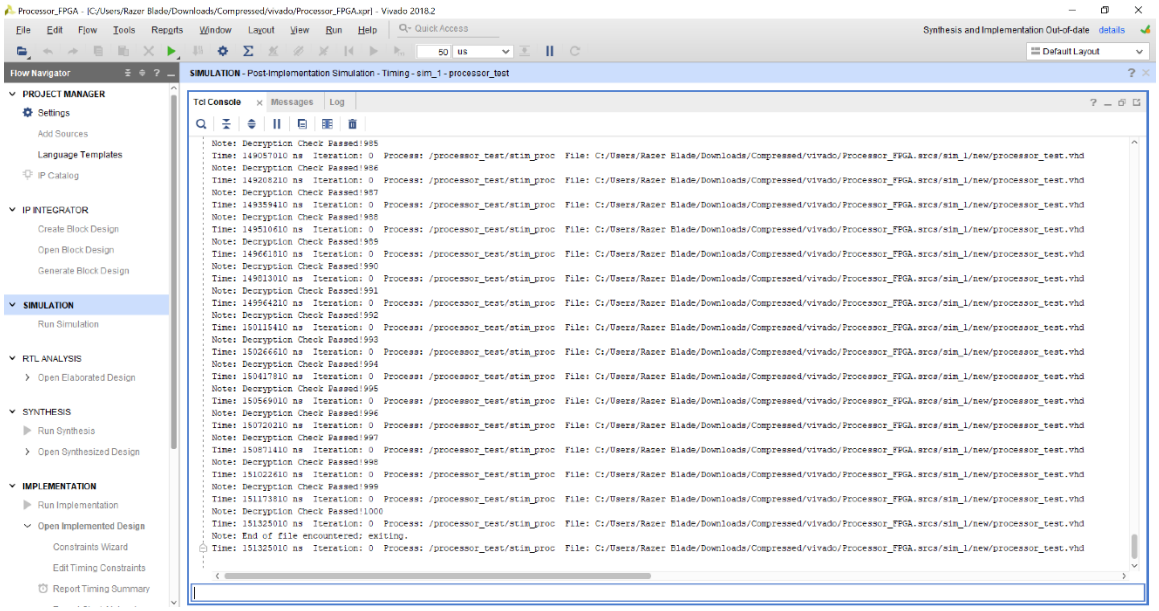


Fig 41: Decryption Check Passed! For 1000 test cases-Timing Simulation.

Corner Cases covered in the timing simulation and their observations:

1) Different input key and plain text covering both 0s and 1 s for each bit.
   Explanation/Observation: Encryption passed was observed for all the cases.
2) When mode is changed from Key generation to encryption mode while key generation is running
   Explanation/Observation: In our implementation when mode is changed the instructions of Key generation are still executed first and after complete execution encryption mode starts.
3) When reset is pressed while decryption is running
   Explanation/Observation: Program counter goes back to initial instruction, then goes to the mode selection part and re-runs the decryption.
4) Input for encryption is changed while its running
   Explanation/Observation: If the program counter was at the load instruction and then input was changed, then the new input is encrypted, if the program counter was after the load instructions part then the change of inputs doesn’t matter, and the previous input is encrypted.

Deliverables

The deliverables for this project include the vhdl code files: ’ALU’, ‘Control Unit’,
’Data Memory’,’Dmem’,’Hex2led’,’IMem’,’Instruction Memory’,’ProgramCounter’,’Registerfile’,’Registers’,’SignExtend’.
The testbench file:’Processor_test.vhd’
XDC file: ‘Processor.xdc’.
Bit file: ‘Processor_FPGA.bit.
and the following video links for each phase:

1. https://youtu.be/wLkuqIcMZu0
2. https://youtu.be/L_Ha_YsN2jo

Final Video Link: https://youtu.be/_H4sY1U5ZLs

Conclusion

Therefore, we implemented a 32-bit processor using VHDL, and used it to execute the RC5 encryption, decryption, and round key generation on an FPGA. The design and interface were performed, and the simulation was done on Xilinx Vivado to acquire the expected results. The performance and area analysis provided all utilization details, and the verification of this design was done using an automated testbench.

Future Scope

Addition of pipelined base design using C/C++ and Vivado HLS can be done and also, the practicality can be improved using interfacing and designing the processor with multiple

cores, extend its capacity to 64-bits (or more) and provide more operations and functions. A minor addition would be to also add decoder to convert the assembly 1s and 0s.

References

1. Project instructions by Professor Ramesh Karri.
2. https://github.com/mmdtoycar/ELGY6463/blob/master/SUPPORT%20PROGRAM/TESTVECTORS/rc5_1k_test_vectors.txt

THANK YOU