COL215 HW Assignment 3 - AES Decryption

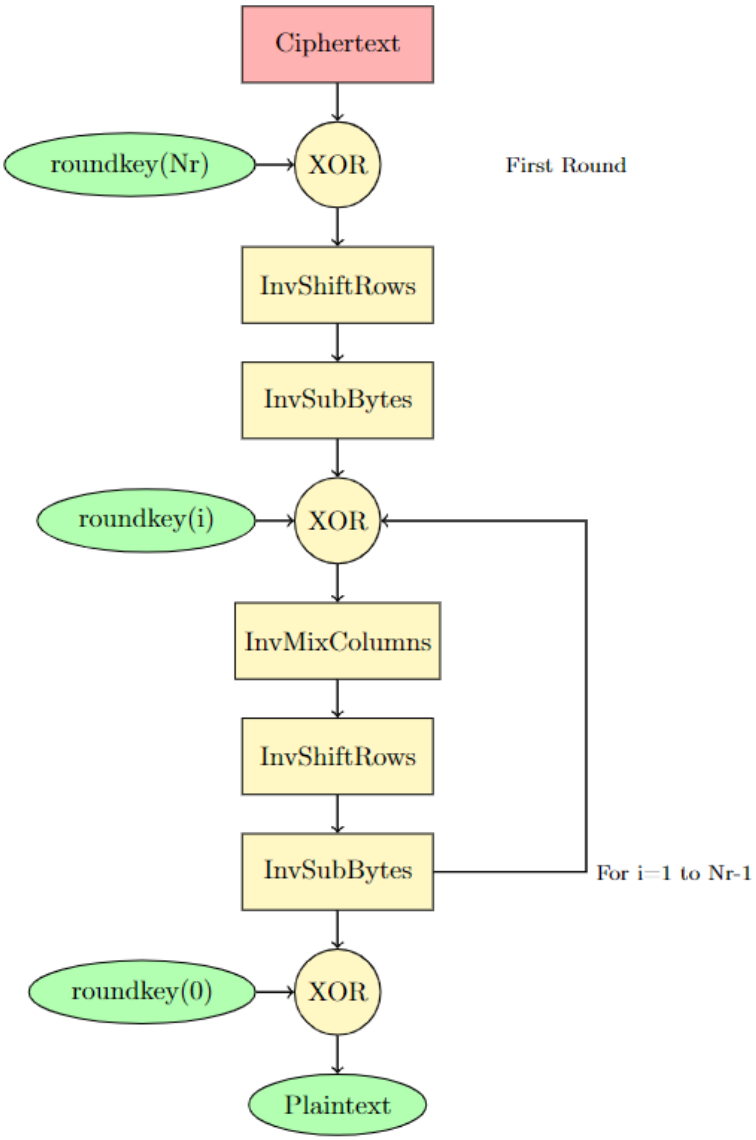
Submission By :

Yash Rawat **2023CS50334**

Priyanshi Gupta **2023CS10106**

Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

October 26, 2024



1 Assignment Problem Statement

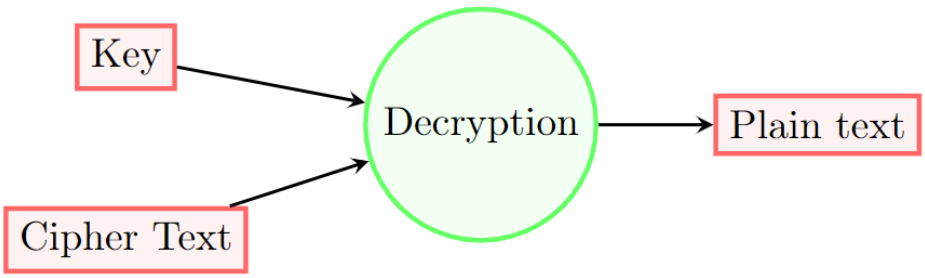
1.1 Introduction

To design an implementation of **AES Decryption Unit** involving various memory elements such as ROM, RAM and logical unit.

1.2 Problem Description

Given a cipher text and round keys, we have to perform an AES decryption operation to obtain plain text. Then the plain text has has to be displayed on Seven Segment Display as shown below :

Figure 1: Overview of task: Decryption



The following details are known :

- Input cipher text is of the size in multiples of 128 bits and will be provided in the COE file (8 bit binary unsigned). The input file should be stored in 1-D array format in the block RAM, starting from address 0 (0000_{16}) in row major format.
- All the keys will be provided via round key file (The implementation of the round key generation logic is not required)

Note that the first part of assignment mainly deals with designing the individual components (rather than connecting them together) which are :

1. Memory Logic (RAM/ROM)
2. Logical Operations involved in AES Decryption

2 Design of Memory Modules

2.1 Use of IP Block Memory

An IP (Intellectual Property) Block Memory in Vivado refers to pre-configured memory modules present on the FPGA board. These blocks, allow efficient storage and retrieval of data directly on the FPGA.

We can instantiate these memory blocks using pre-configured `COE` files, which allow us to load our test case data onto the board, which can be used throughout the implementation.

2.2 Brief of ROM and RAM Design

2.2.1 ROM - Read Only Memory

ROM Design involves the following details :

- Implementation be used for storing and accessing the round key data, input cipher text as well as performing lookup for `Inv_Sub_Byte` operation.
- Using Generic keyword we can vary the `address_width` as well as `data_width` and instantiate a component of `IP` block that has been pre-loaded onto the board.
- Only read operations have to be supported (hence the name Read Only Memory)

2.3 RAM - Random Access Memory

RAM Design involves the following details :

- Implementation be used to store and fetch data for intermittent calculations of AES Decryption.
- Supports dynamic sizing at time of instantiation (using Generic Keyword)
- For synchronous design, any read or write operation will only be performed of `rising_edge` of standard `clk`. If `we` (write enable) is turned `1`, data at `din` is assigned to `addr` of RAM, otherwise we just assign the data at `addr` to `dout`
- The data of a RAM is stored in an `std_logic_vector(DATA_WIDTH-1 downto 0)` of `array(0 to 2**ADDR_WIDTH -1)`

3 Design of Modules for Decryption

The hardware blocks employ a generic keyword and are highly customisable for future scaling. At this stage, we are not interconnecting the components; thus, we have utilised inputs for the entire 128 bits. To facilitate future correctness verification, we have ensured that the same blocks can be efficiently employed with minor adjustments, thereby preventing resource overuse on the FPGA board. For instance, we can directly implement gf_256_multiply in our code later on.There four modules that are implemented are as follows.

3.1 Bitwise_XOR

The bitwise XOR module in AES decryption component performs the AddRoundKey operation by conducting a bit-by-bit XOR operation between a 128-bit data block and a 128-bit round key. This module is simple yet crucial, producing an output where each bit is 1 only if the corresponding input bits differ. It is implemented as a fully combinational circuit (no clock required) and is used multiple times during decryption. The XOR operation's key feature is its reversibility: applying the same key twice returns the original data.

54	49	36	65
68	73	42	4B
31	41	79	65
73	31	74	79

$\oplus$

00	69	57	06
00	1A	62	39
58	32	0A	00
00	11	11	0D

=

54	20	61	63
68	69	20	72
69	73	73	65
73	20	65	74

(d) XOR operation

3.2 Inv_Row_Shift

This operation serves as the inverse of the ShiftRows function utilised in encryption. Each row of the state is cyclically shifted to the right by a specified number of positions. The initial row is preserved, the subsequent row is displaced one position to the right, the third row is displaced by two positions, and the fourth row is displaced by three positions. This reinstates the original row configuration, reversing the transposition applied during the encryption phase.

63	F9	5B	6F
A2	AA	12	63
67	63	6A	23
D7	63	82	82

InvRowShift

63	F9	5B	6F
63	A2	AA	12
6A	23	67	63
63	82	82	D7

(b) InvRowShift operation

3.3 Inv_Sub_Bytes

In this step, each byte in the state is substituted with its corresponding inverse value from the inverse S-box.This process undoes the non-linear substitution executed during the SubBytes step of encryption, thereby reinstating the original byte values prior to substitution.

63	F9	5B	6F
63	A2	AA	12
6A	23	67	63
63	82	82	D7

InvSubbytes

00	69	57	06
00	1A	62	39
58	32	0A	00
00	11	11	0D

(c) InvSubbytes operation

3.4 Inv_Mix_Columns

The VHDL code implements a sequential GF(2^8) array multiplier for 4x4 matrices used in AES operations. It uses a state machine with three states (IDLE, COMPUTING, COMPLETED) to control the multiplication process, where each element of the result matrix is computed by multiplying and XORing corresponding elements from input matrices array_a and array_b. The module operates on a clock signal and begins processing when the start signal is received. It uses three counters (row, col, k) to traverse through the matrices, performing one GF(2^8) multiplication per clock cycle using a sub-component gf256_multiply. The results are accumulated using XOR operations and stored in an internal 128-bit vector. The module includes debug ports for monitoring intermediate values and signals completion through the done port when all computations are finished.

3.5 gf256_multiply

This module implements Galois Field multiplication for AES operations.GF(2^8) multiplication is a key component in AES's MixColumns/InvMixColumns transformations.It takes two 8-bit inputs ('a' and 'b') and uses irreducible polynomial $x^8 + x^4 +$

0E	0B	0D	09
09	0E	0B	0D
0D	09	0E	0B
0B	0D	09	0E

*

8B	0C	68	DA
42	70	43	4E
6D	30	00	D7
D5	1F	8A	EE

=

63	F9	5B	6F
A2	AA	12	63
67	63	6A	23
D7	63	82	82

(a) InvMixColumns operation

x^3+x+1 (0x11B) to carry out the multiplication.It returns an 8-bit result in GF(2^8). The multiplication is performed using a shift-and-add algorithm:

1. For each bit in the second operand (y)
2. If the bit is 1, XOR the running product with the shifted first operand (x)
3. After each iteration, shift the first operand left
4. If the shifted value would overflow, XOR with the reduction polynomial

This is similar to regular binary multiplication but includes the reduction step to keep results within GF(2^8). The constant checking and reduction with polynomial 0x11B ensures the result stays within the finite field.

3.6 AES_AddRoundKey

This module performs a simple bitwise XOR operation between two 8-bit inputs (input1 and input2) when the enable signal (en) is high.

3.7 AES_Key

This VHDL module implements a key lookup function for AES by using a ROM_Key component to map 8-bit input addresses to corresponding key values in an S-Box lookup table. When enabled, it accepts an input data vector, uses it as an address to fetch the corresponding key value from ROM, and outputs the result.

3.8 AES_Round

This VHDL code implements a single round of AES decryption with four main transformations. The module uses a finite state machine (FSM) to control these operations sequentially, processing 128-bit data blocks and generating the decrypted output when done.

3.9 AES_RoundLogic

This VHDL code implements an AES round for decryption in a hardware design. It includes several components that perform the core AES transformations.

3.10 register

This VHDL code defines an N-bit register with synchronous reset and enable. When the clock has a rising edge, the register outputs either a reset value (all zeros) or the input data based on the enable signal.

3.11 AES_Text

This VHDL code implements a component that maps an 8-bit input address to an 8-bit output value using a ROM-based S-box lookup for AES encryption. It passes the input address to the ROM and outputs the corresponding value when enabled.

3.12 AES_Controller

This is the wrapper module which controls the decryption process.

3.13 AES_Display

This module implements a top-level module that connects an controller component (handling AES decryption) with a seven-segment display component .

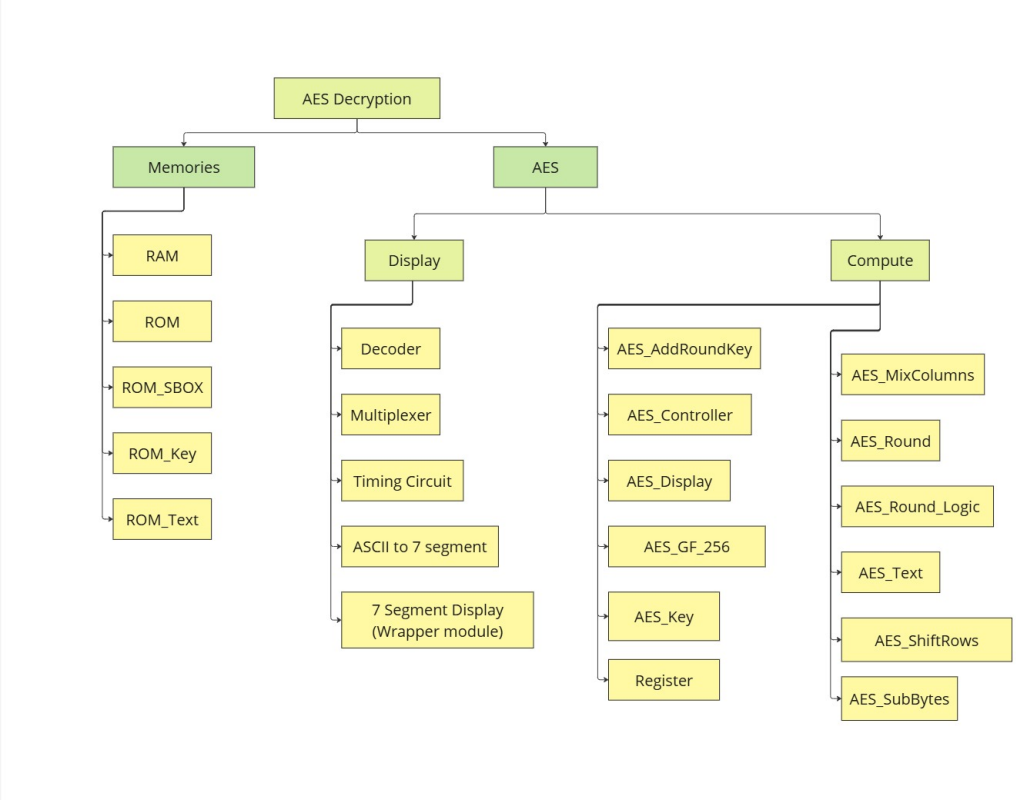


Figure 1: Project Structure

4 Design of Finite State Machine

1. An FSM (Finite State Machine) in AES decryption using FPGA memories is a pivotal component that carries out the decryption process with precision and efficiency. The FSM meticulously manages state transitions and the sequence of operations required to decrypt data.

2. The decryption process involves multiple stages, including initial key addition, inverse substitution bytes (InvSubBytes), inverse shift rows (InvShiftRows), inverse mix columns (InvMixColumns), and final key addition. Each of these stages must be executed in a specific order to ensure the integrity of the decrypted data.

3. The FSM begins by loading the necessary data and keys from the Block RAM, utilizing the FPGA's high-speed memory capabilities. It then transitions through various states, each representing a distinct phase of the AES decryption algorithm.

4. For instance, in the InvSubBytes state, the FSM fetches data from memory, applies the inverse S-box transformation, and stores the result back in the registers or RAM. Similarly, during the InvShiftRows and InvMixColumns states, the FSM carefully reads from and writes to memory, applying the respective transformations to the data.

5. To handle the timing and synchronization of these memory operations, the FSM incorporates WAIT states. These states account for the setup and hold times of read and write address lines, as well as the data read access time specified in the FPGA's datasheet.

6. The FSM coordinates with control signals to manage the flow of data between the compute unit and memory. It continuously monitors input flags, to determine when an operation is complete and to change the operation mode.

5 Cyclic Display of Text

The features of our display implementation are as follows

1. The system displays scrolling text on the Basys 3 board's four 7-segment displays. It converts hexadecimal values (0-F) to ASCII characters and displays them continuously.

2. The system handles both uppercase and lowercase letters identically (e.g., 'f' and 'F' show as 'F'), and shows "-" for any out-of-range characters.

3. The plaintext can include spaces, and all text scrolls cyclically across the display, showing four characters at a time. The implementation builds upon the existing VHDL code from HW Assignment 2 for the display control.

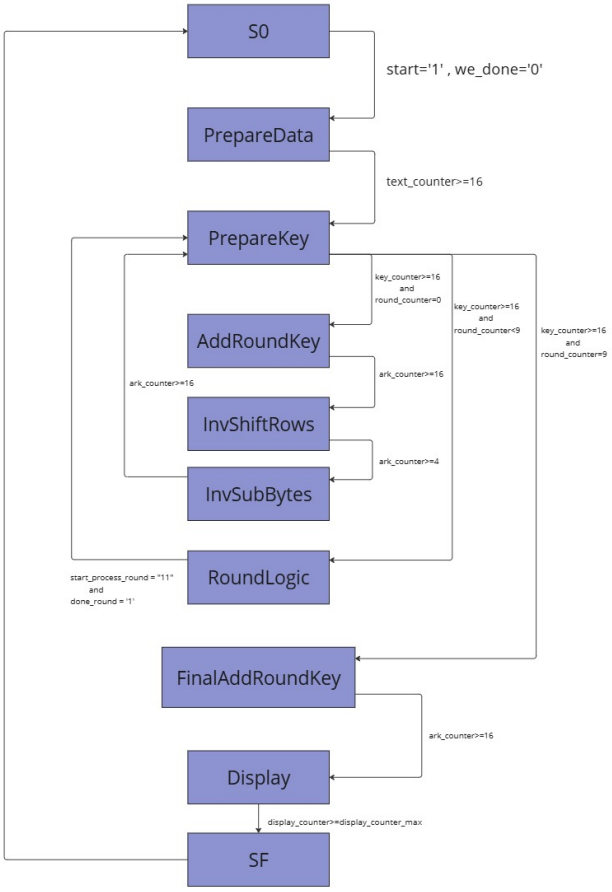

S0
start='1' , we_done='0'
PrepareData
text_counter>=16
PrepareKey
key_counter>=16
and
round_counter=0
key_counter>=16
and
round_counter<9
key_counter>=16
and
round_counter=9
AddRoundKey
ark_counter>=16
ark_counter>=16
InvShiftRows
ark_counter>=4
InvSubBytes
start_process_round = "11"
and
done_round = '1'
RoundLogic
FinalAddRoundKey
ark_counter>=16
Display
display_counter>=display_counter_max
SF

Figure 2: Fin

6 Simulation Snapshots

6.1 Simulation of ROM and ROM_INV_SBOX

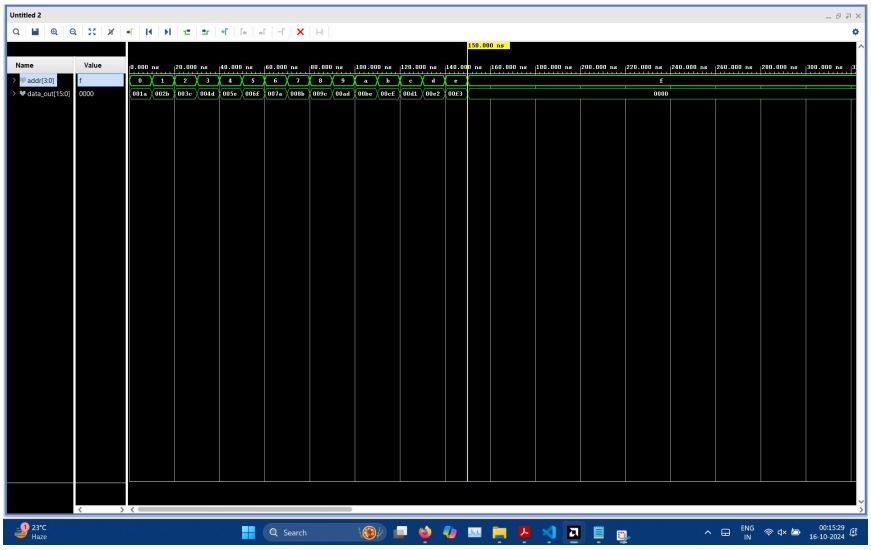


Figure 3: Testing ROM - Implementation

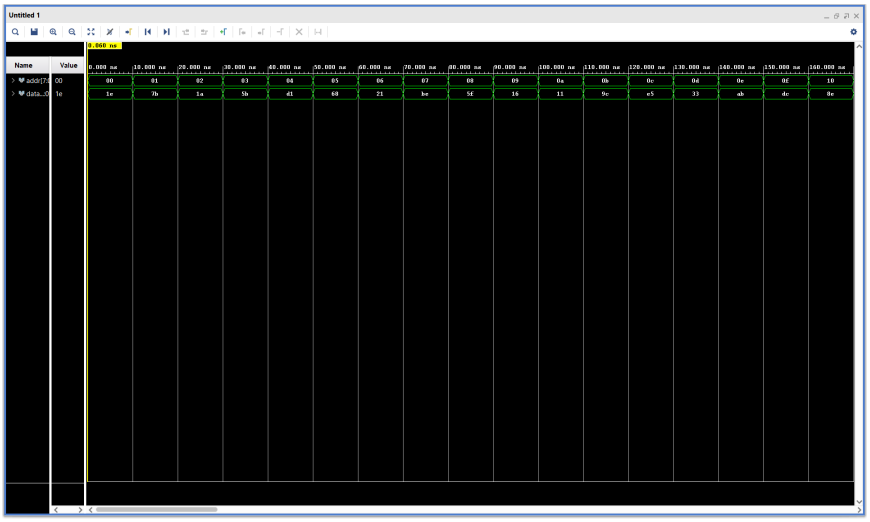


Figure 4: Testing ROM INV_SBOX - Using Generic Block

The above `INV_SBOX` uses data from the custom generated `coe` file as mentioned below :

COL215_assignments > HW Assignments > HW Assignment 3 > Local_Files_Yash > COE_Files > rom.coe

```
memory_initialization_radix=16;
memory_initialization_vector=
1E, 7B, 1A, 5B, D1, 68, 21, BE, 5F, 16, 11, 9C, E5, 33, AB, DC,
8E, F4, E6, 38, 3F, 4F, 72, 57, 25, 05, 34, A2, B6, E8, 5F, F2,
08, 0B, 8B, 2D, BE, 84, 50, 94, 2D, BA, DF, BB, 94, 48, 93, 56,
7D, 5B, 41, FD, 6C, EF, 6F, C6, FA, 53, 35, A9, 1E, 5B, 3C, 8A,
45, 35, 8A, 9F, 42, 1E, 51, 8C, A4, 9C, 7D, 43, C1, 4C, DA, 63,
A1, B4, CA, 5C, D5, EC, 5D, 0E, 1F, 96, 3D, F9, 59, 7D, B8, 46,
CF, 27, F0, A9, 52, F7, C0, D9, 8D, EE, 6F, AA, DD, 5B, 0E, EE,
69, 7E, 72, B4, B2, 82, 3C, 5F, E1, CA, 83, 7A, 8F, 8E, 82, A1,
67, 83, FB, B6, CE, 29, F1, FA, F4, 80, 50, 94, 38, C1, F9, 4C,
D3, 91, 35, AC, 13, 47, 48, EE, DE, 40, 97, 7E, 70, 5D, 3F, CB,
30, 36, B5, 28, 31, D4, E1, 24, F4, 6B, 2C, B8, DE, DE, 05, 96,
01, 55, CF, C1, 9C, 28, 68, 3E, 79, 50, F8, D7, 82, 8A, 2F, 5D,
D6, A2, AE, EF, 5B, 34, 03, F4, E7, A6, 5F, 31, 8A, F1, B1, 56,
00, 6B, 74, F0, C6, C6, C0, 2A, C9, 4C, 5B, 0C, B5, D7, E1, 28,
DE, 87, EC, 8D, A5, 2C, 93, C2, FE, 41, F6, DC, EB, DD, FB, 26,
43, 5B, BB, 78, 30, 10, 7C, 3A, 61, 3E, 39, 52, 78, B9, 59, DF;
```

Figure 5: COE File data for test simulation

As is visible the ROM is able to fetch the correct value based on the input address, which will be required during later stages of implementation.

6.2 Simulation of RAM

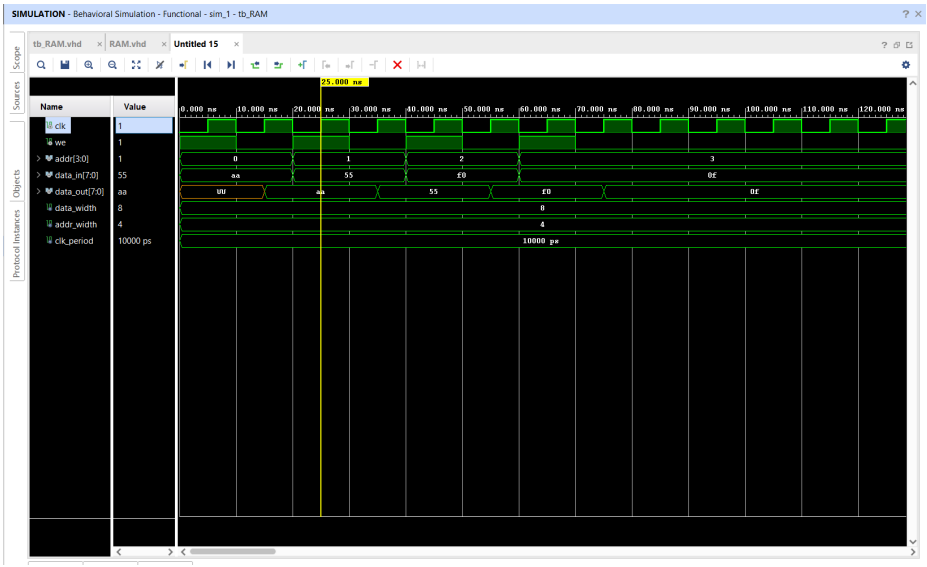


Figure 6: Testing RAM implementation

As is visible by the yellow marker, on the `rising_edge` (at `25 ns`) of `clk`, when `we` (write enable) is `1`, current `data_in` is written to the `data_addr`, which is visible on the next `rising_edge` (at `35 ns`) on `data_out` (when `we` is `0`), thus giving expected behaviour.

6.3 Simulation of XOR

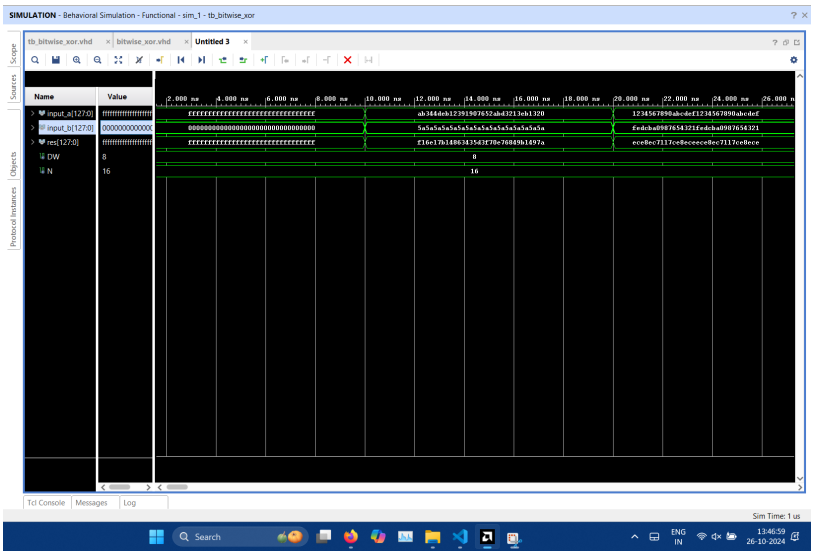


Figure 7: Testing XOR implementation

Generic block allows us to have flexibility over the lengths of inputs , which is useful at later stages of implementation.

6.4 Simulation of Inv_Sub_Bytes

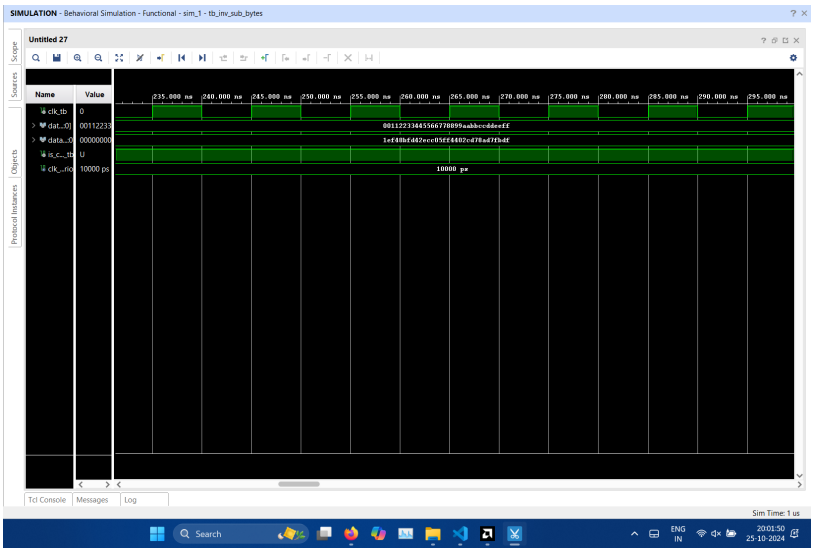


Figure 8: Testing Inv_Sub_Bytes Implementation

Note that the implementation uses `INV_SBOX` data from the previously mentioned `coe`, hence the waveform matches the expected output

6.5 Simulation of Inv_Row_Shift

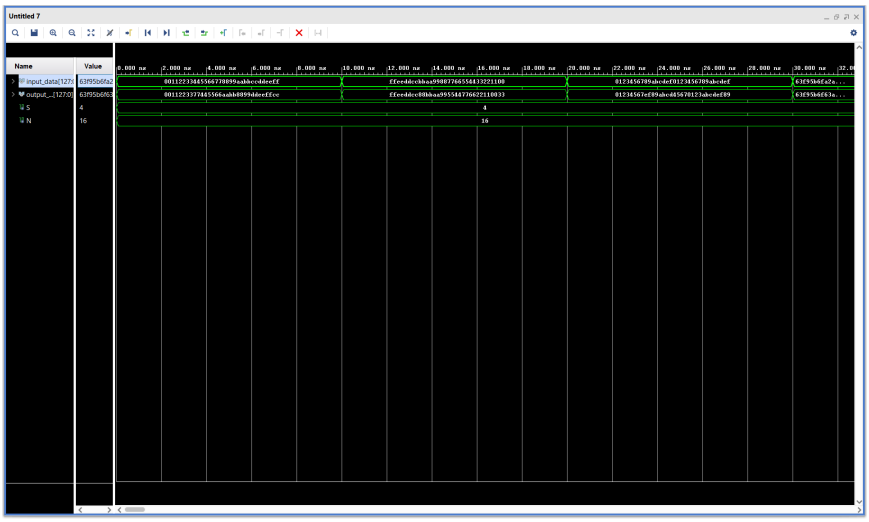


Figure 9: Testing Inv_Row_Shift Implementation

The implementation's waveform matches the expected output as the 4 set of 4 bytes receive 0,1,2,3 rotations which the desired outcome.

6.6 Simulation of Inv_Mix_Columns

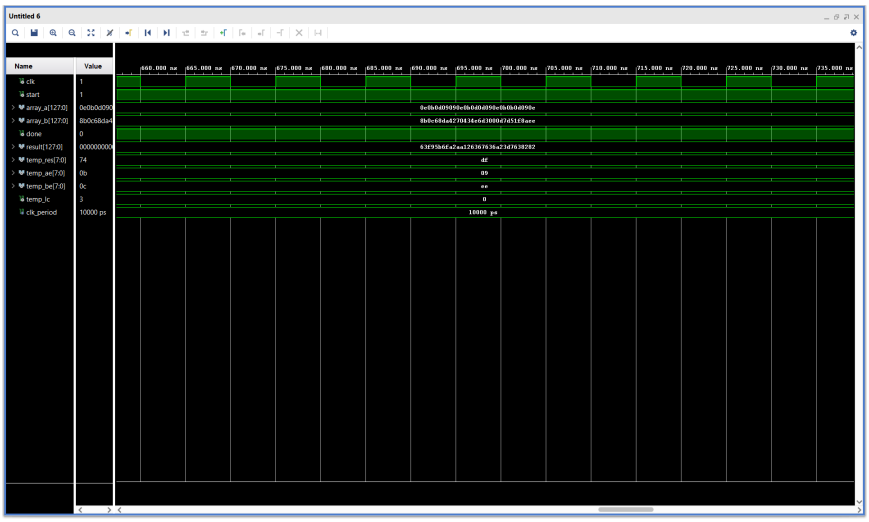


Figure 10: Testing Inv_Row_Shift Implementation

The implementation takes the input as given in the handout and the waveform matches the expected output as per the handout, verifying it's implementation. Another image has been attached to show intermediate calculation steps below :

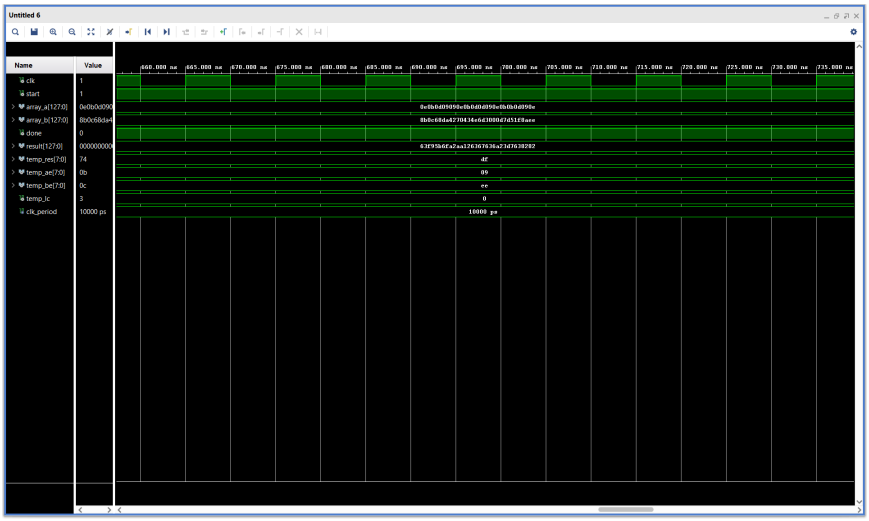
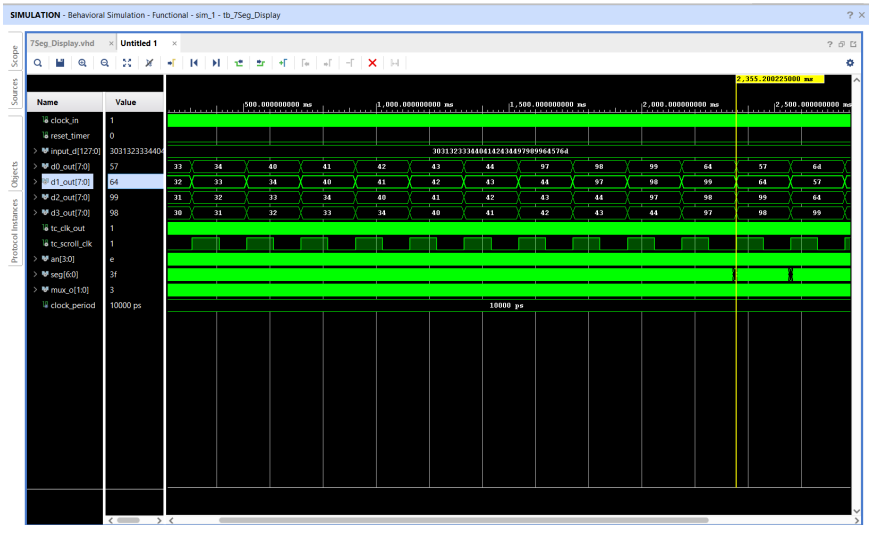


Figure 11: Testing Inv_Row_Shift Implementation - Calculation steps

6.7 Simulation of Scrolling Output on Seven Segment Displays After Decryption



6.8 Simulation of the FSM

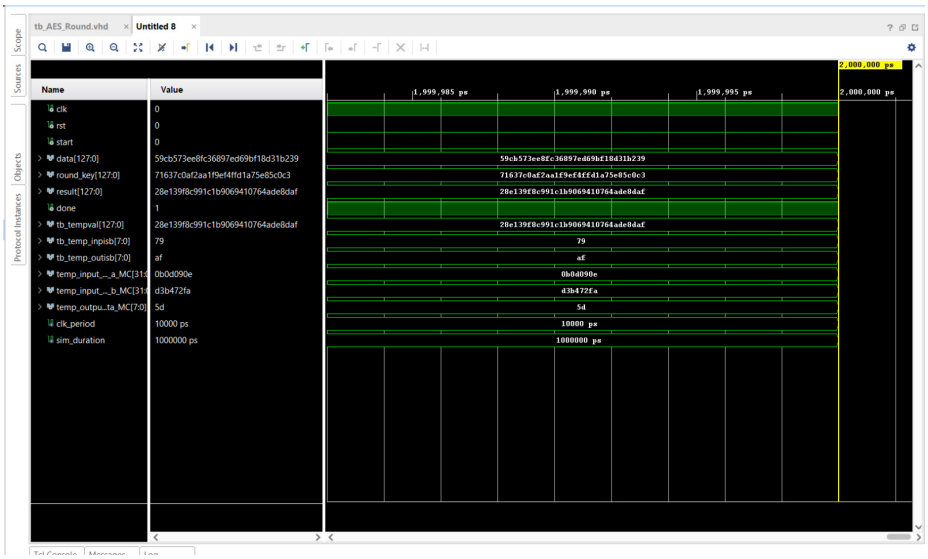


Figure 12: Single Round FSM simulation

6.9 Simulation of Scroll Display Module

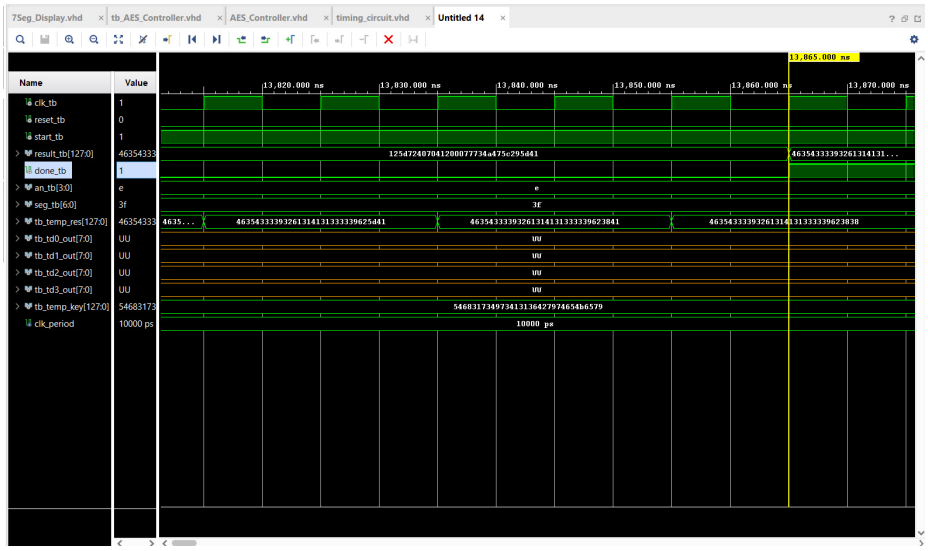


Figure 13: Decryption of one 128 bit input block in 13 microseconds

As is visible, Fig:10 shows the overall scrolling output (`d_0`,`d_1`,`d_2`,`d_3`) denotes the `ASCII` value of character, which decrypted and to be displayed on the respective 4 seven segment displays. It is sequentially changed in order to produce a scrolling effect. Note that Fig: 11 shows handling of loop around and Fig: 12 shows edge-triggered cyclic change.

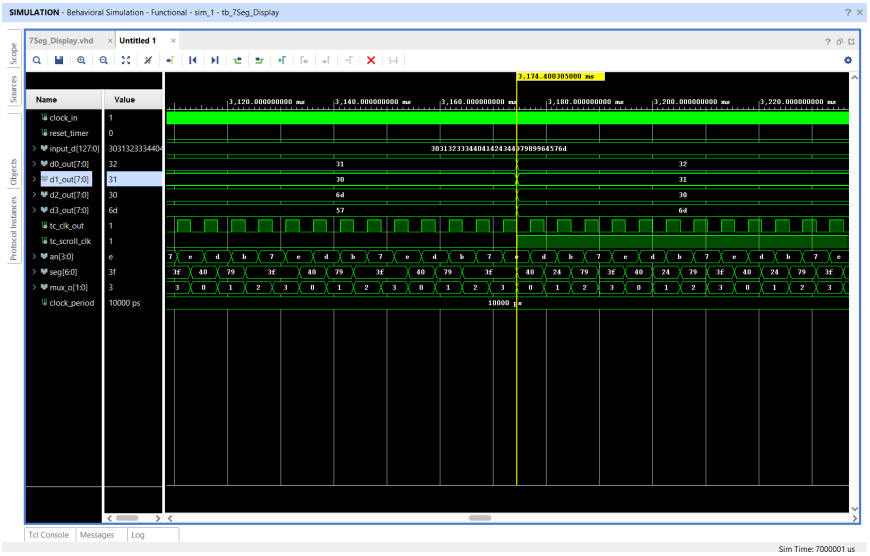


Figure 14: Testing Scrolling Output Implementation - Looping of output

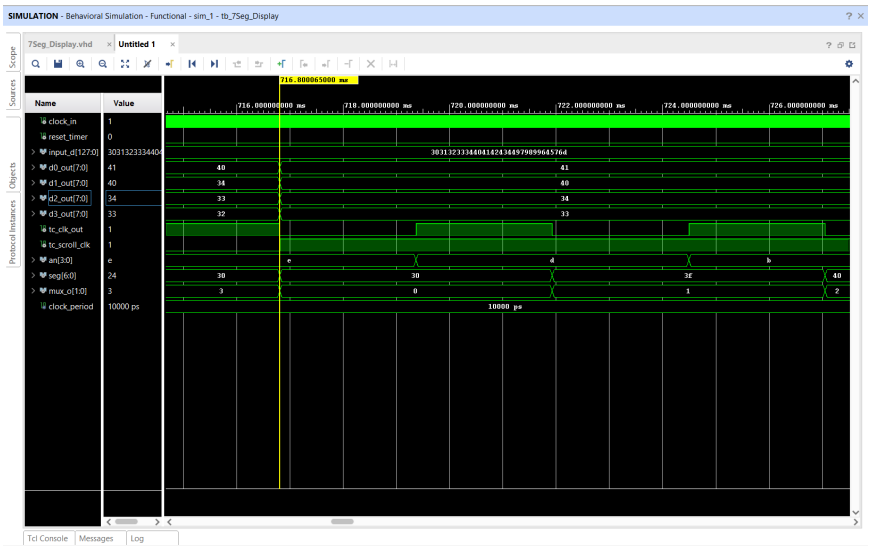


Figure 15: Testing Scrolling Output Implementation - Transition based on scroll_clk

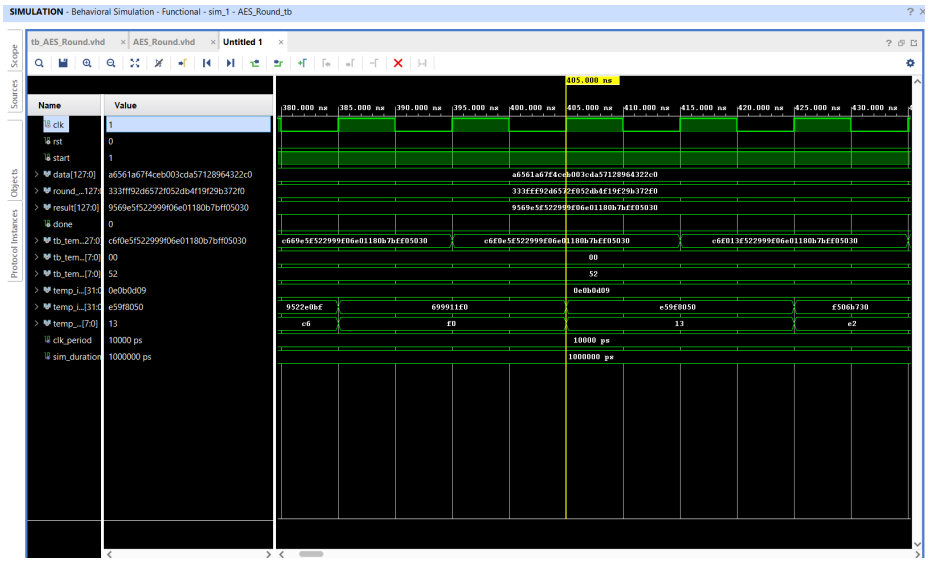


Figure 16: Waveform - Decryption Process

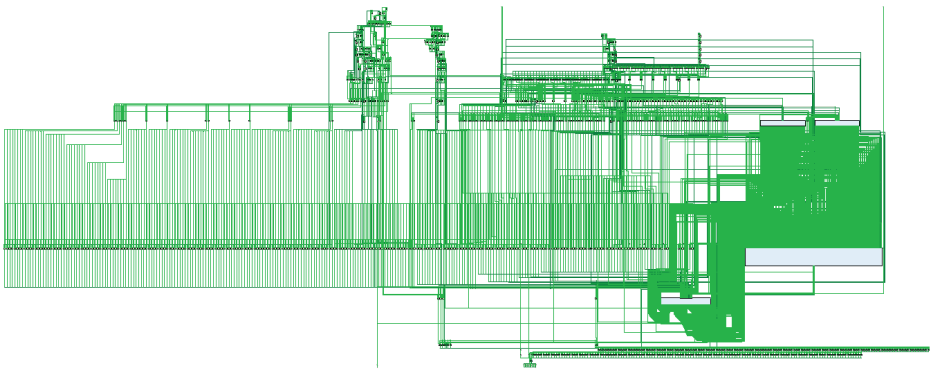
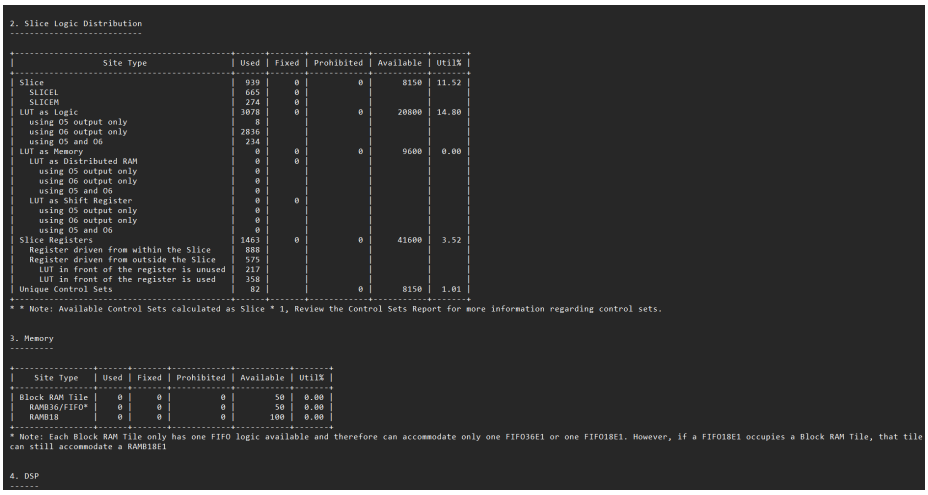


Figure 17: Schematic of final Top Module (Synthesized)



2. Slice Logic Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	939	0	0	8150	11.52
SLICEL	665	0			
SLICEM	274	0			
LUT as Logic	3078	0	0	20800	14.80
using O5 output only	8				
using O6 output only	2836				
using O5 and O6	234				
LUT as Memory	0	0	0	9600	0.00
LUT as Distributed RAM	0	0			
using O5 output only	0				
using O6 output only	0				
using O5 and O6	0				
LUT as Shift Register	0	0			
using O5 output only	0				
using O6 output only	0				
using O5 and O6	0				
Slice Registers	1463	0	0	41600	3.52
Register driven from within the Slice	888				
Register driven from outside the Slice	575				
LUT in front of the register is unused	217				
LUT in front of the register is used	358				
Unique Control Sets	82		0	8150	1.01

* * Note: Available Control Sets calculated as Slice * 1, Review the Control Sets Report for more information regarding control sets.

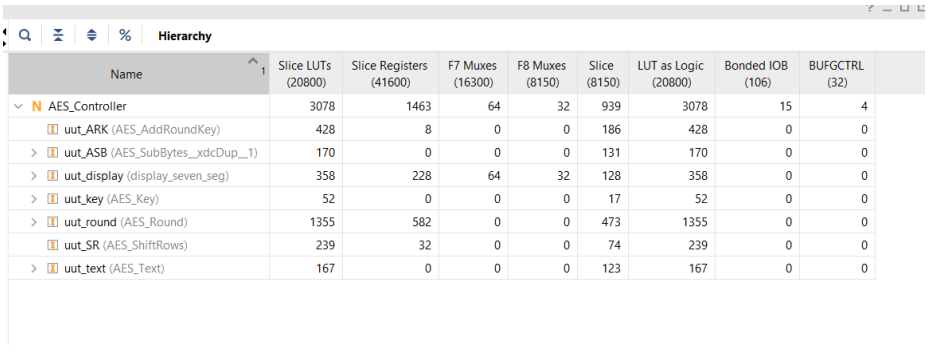
3. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	0	0	0	50	0.00
RAMB36/FIFO*	0	0	0	50	0.00
RAMB18	0	0	0	100	0.00

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

4. DSP

Figure 18: Resource Utilisation Table



Hierarchy

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	Bonded IOB (106)	BUFGCTRL (32)
AES_Controller	3078	1463	64	32	939	3078	15	4
uut_ARK (AES_AddRoundKey)	428	8	0	0	186	428	0	0
uut_ASB (AES_SubBytes__xdcDup__1)	170	0	0	0	131	170	0	0
uut_display (display_seven_seg)	358	228	64	32	128	358	0	0
uut_key (AES_Key)	52	0	0	0	17	52	0	0
uut_round (AES_Round)	1355	582	0	0	473	1355	0	0
uut_SR (AES_ShiftRows)	239	32	0	0	74	239	0	0
uut_text (AES_Text)	167	0	0	0	123	167	0	0

Figure 19: Resources Utilisation across modules