ECE 385
Spring 2020

Experiment #9

SOC with Advanced Encryption Standard in SystemVerilog

Ishaan Datta, Lukas Dumasius
Lab ABE/Tuesday 3pm
Jiaxuan Liu

Introduction

The objective of this lab was to combine much of our previous knowledge of HDL design in SystemVerilog and interfacing and running operations with the NIOS II to implement the AES (Advanced Encryption Standard) across software (for the encryption) and SystemVerilog hardware (for the decryption) to create an IP or Intellectual Property core. The first week or part of this lab involved the implementation of standard AES 128-bit encryption on the software part of IP core. Then, in week or part two, we implemented the reverse- AES 128-bit decryption in SystemVerilog that is able to decrypt the encoded message from software.

Written Description and Diagrams of the AES encryptor/decryptor

Written description of the software encryptor

The overall encryption algorithm must be understood or defined to implement in software. This encryption process, implemented in AES fashion, is a cipher that turns a user message into garbled data that cannot be comprehended by a third party. This is a basic common security feature, primarily for data storage and transfer.

The user inputted "Plaintext" message that we want to be encrypted is passed to the cipher algorithm, which scrambles the data, along with the appropriate "Cipher Key" which we implemented according to the AES algorithm. This means that we follow a block cipher approach, as an algorithm that performs operations on a set number of bits, denoted as fixed blocks one at a time. The AES algorithm is a "symmetric"-key block cipher algorithm because it uses the same cipher key set for encoding (encryption) and decoding (decryption). This approach is based on a slight modification to the Rijndael algorithm as AES has standardized 128 bit data blocks. So, we implement an approach with AES algorithm using cipher keys of 128-bit length.

The AES encryption algorithm is implemented with a State logic[127:0] which stores intermediate results during the process of the algorithm. It is organized in sets of 4x4 Byte, 16 Byte sets in a column-major order matrix. This is split into four "Words" logic [31:0] grouped as sets of 32 bits = 4 Bytes of data from a given current column in "State". Our AES algorithm takes in the "Plaintext" message and the Cipher Key, expanded by KeyExpansion() which generates "Round Keys" logic[127:0] used for encryption along with the "Key Schedule" logic[1407:0] that holds the the Round Keys specifically generated from the Key.

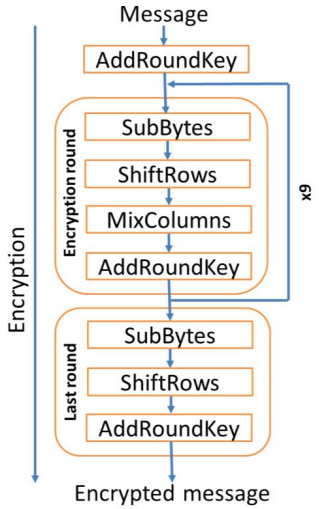


Figure 2 AES encryption algorithm flow.

Then, we go into a series of 10 looping rounds where we run modules SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey() (further explained below) which then

outputs the “Ciphertext”, or encoded message, which will eventually go to the hardware decryption part to decode the original message. AddRoundKey() XORs every cell with the Key for every state. The SubBytes() transformation is simplified from a Rijndael’s finite field calculation to a “Substitution Box” lookup table. The entire state is updated with the corresponding value from the lookup table. Then, each row in States is shifted by ShiftRows() which follows a pattern of doing a left circular shift to each successive row increasing the offset by 1 each time. Meaning, the first row is not shifted at all and the second row is shifted by 1 bit and so on. This is passed to MixColumns() which performs an XOR using the current state to linearly combine by multiplying the current state four Byte word matrix into a new word. After the looping completes the appropriate number of times, the encoded version of the message is outputted.

Written description of the hardware decryptor

The decryption is implemented in hardware and receives the ciphertext, encoded message, and uses the key to decode the original message. It works similar to encryption, but must loop in the opposite direction to inverse the encryption modules to function as an inverse cipher algorithm.

The AES algorithm state controller kept track of which state in the decryption function we are in so that successive operations are executed properly. It loops through the InvShiftRows, InvSubBytes, AddRoundKey, and InvMixColumns then exits to finish decryption and return the decoded message at completion.

AddRoundKey XORs the current state matrix part with the KeySchedule set, as mentioned above, InvShiftRows does the inverse or reverse operation, so it circularly shifts right each row of the state successively by one more. InvMixColumns and InvSubBytes were already provided to inverse their functions.

```
byte state[4,Nb]
state = in
AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
for round = Nr-1 step -1 downto 1
    InvShiftRows(state)
    InvSubBytes(state)
    AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    InvMixColumns(state)
end for
InvShiftRows(state)
InvSubBytes(state)
AddRoundKey(state, w[0, Nb-1])
out = state
```

Decryption Pseudo-code loop snippet

Written description of the hardware/software interface

This is the IO module that interfaces between the software encryption and hardware decryption. It takes the encrypted message and passes to hardware module. There are 16 loops for reading the next 8 bits of the message at a time until the message is received and we leave that state. Once we are ready for decryption and signal is set active, we again loop through and read the next 8 bits at a time for the decrypted message to pass the entire contents.

This module takes inputs AVL_READ, AVL_WRITE, AVL_CS, AVL_BYTE_EN, AVL_ADDR, and AVL_WRITEDATA along with output AVL_READDATA and EXPORT_DATA. AVL_READDATA and EXPORT_DATA are used for LED display and output. In the 16 registers for the Avalon interface, r0 through r3 takes key input data from AVL_WRITEDATA and r8 through r11 store the decrypted "Word" passed from msg_dec. 8 and 15 are used as start and done registers.

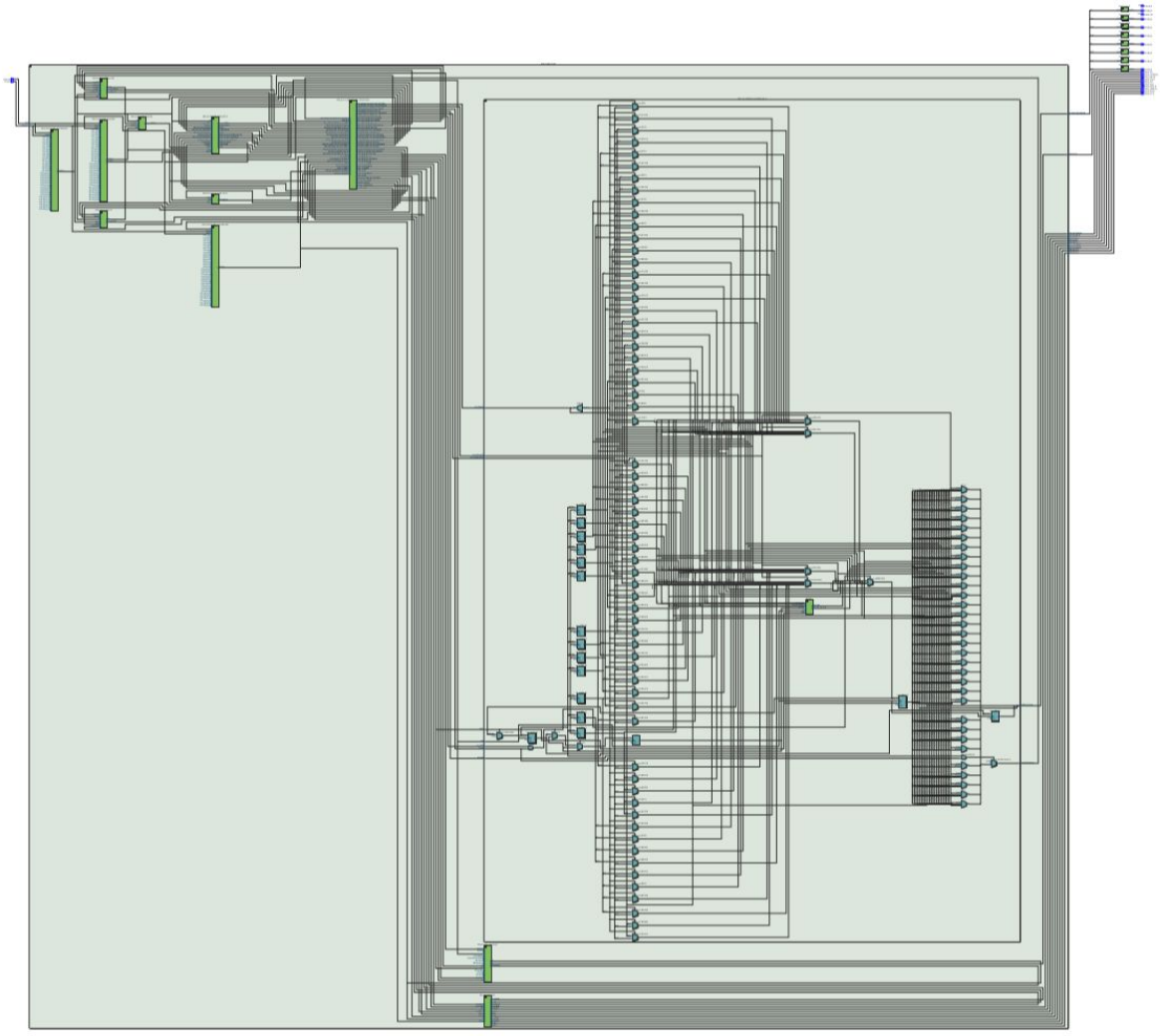
```
module avalon_aes_interface (
    // Avalon Clock Input
    input logic CLK,

    // Avalon Reset Input
    input logic RESET,

    // Avalon-MM Slave Signals
    input  logic AVL_READ,
    input  logic AVL_WRITE,
    input  logic AVL_CS,
    input  logic [3:0] AVL_BYTE_EN,
    input  logic [3:0] AVL_ADDR,
    input  logic [31:0] AVL_WRITEDATA,
    output logic [31:0] AVL_READDATA,

    // Exported Conduit
    output logic [31:0] EXPORT_DATA
);
```

The interface is set up in such a way where access is managed between the software and hardware components. Reading the encrypted message from software and reading the 4 blocks of decrypted message from the hardware are only executed with the appropriate "load" signal active.

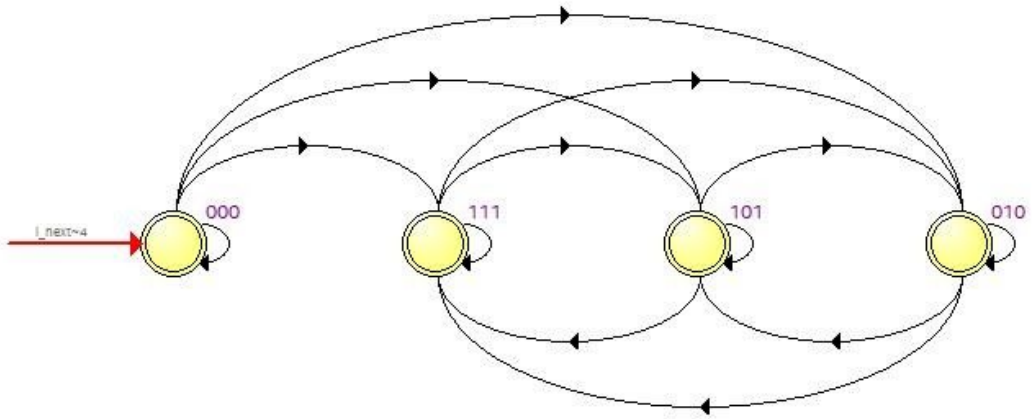
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1x	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2x	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3x	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4x	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5x	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6x	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7x	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8x	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9x	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
ax	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
bx	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
cx	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
dx	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
ex	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
fx	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Table 2 Rijndael S-box.

Block diagram



State Diagram of AES decryptor controller



Module Descriptions

- Module: SubBytes.sv
Inputs: clk, [7:0] in
Outputs: [7:0] out
Description & Purpose: This module contains the SubBytes and inverse SubBytes modules that perform the lookup table operation after reading the appropriate bits from the state matrix and passing the corresponding new state value.

- Module: lab9_top.sv
Inputs: CLOCK_50, [1:0] KEY
Outputs: [7:0] LEDG, [17:0] LEDR, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [6:0] HEX6, [6:0] HEX7, [12:0] DRAM_ADDR, [1:0] DRAM_BA, DRAM_CAS_N, DRAM_CKE, DRAM_CS_N, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
Description & Purpose: This is the top level module which instantiates many of the other modules including hexdrivers and how we display on LEDs. This allows our software encryption and hardware decryption to communicate and we can connect logic or signal connections between modules to maintain the state, key, and other data for decryption.

- Module: KeyExpansion.sv
Inputs: clk, [127:0] Cipherkey
Outputs: [1407:0] KeySchedule
Description & Purpose: This module uses the given key to write a KeySchedule form Round Keys which is used during decryption (and encryption similarly). These will be the keys used during decryption as it steps through the states and loops through the successive decoding.

- Module: avalon_aes_interface.sv
Inputs: CLK, RESET, AVL_READ, AVL_WRITE, AVL_CS, [3:0] AVL_BYTE_EN, [3:0] AVL_ADDR, [31:0] AVL_WRITEDATA
Outputs: [31:0] AVL_READDATA, [31:0] EXPORT_DATA
Description & Purpose: This is the interface that allows our hardware and softwre to pass dat ato each other. This module waits for encrypted message to be written in blocks or decrypted message as appropriate and manages writing permission correctly. This allows us to retrieve the final output value.

- Module: AES.sv
Inputs: CLK, RESET, AES_START, [127:0] AES_KEY, [127:0] AES_MSG_ENC
Outputs: AES_DONE, [127:0] AES_MSG_DEC
Description & Purpose: This module functions as the actual decryptor that gets interfaced the encoded message and steps through all of the decryption states and after looping it finally outputs the final product. We use the enable and select bits for our muxes that connect to the state machine to perform the operation.

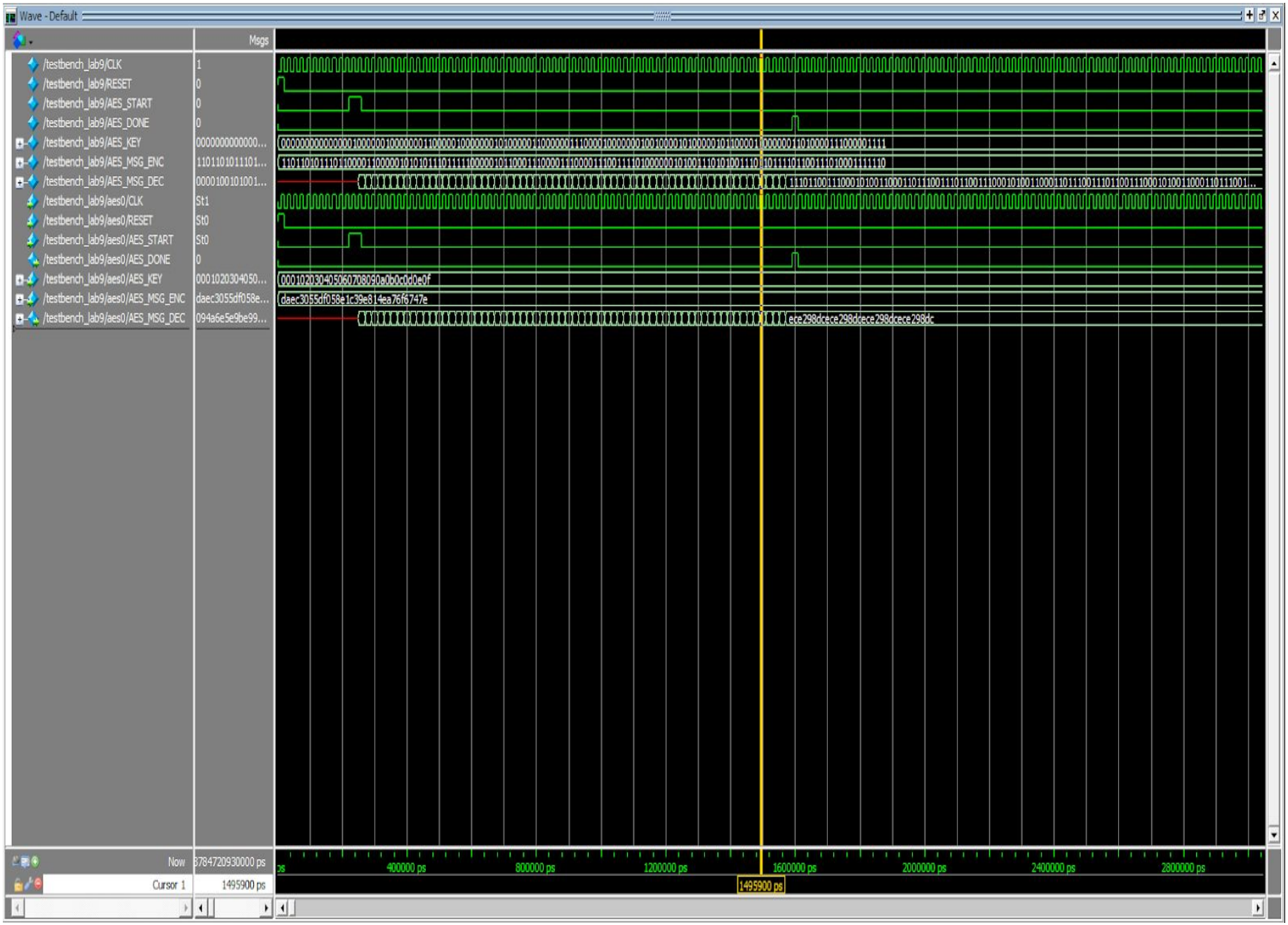
- Module: InvMixColumns.sv
Inputs: [31:0] in
Outputs: [31:0] out
Description & Purpose: This is one of the essential operations for decryption that get looped through. It performs the inverse operation of MixColumns() by performing the inverse matrix multiplication of the “Word” and the designated matrix to pass the new updated “Word”.

- Module: InvShiftRows.sv
Inputs: [127:0] data_in
Outputs: [127:0] data_out
Description & Purpose: This module takes in a block of a “Word” and performs the inverse to the ShiftRows() function in the encryption. Thus, as detailed above, it performs a right circular shift on each row, successively increasing the offset by 1 bit each time.

- Module: AddRoundKey.sv
Inputs: [127:0] state, [127:0] key
Outputs: [127:0] out

Description & Purpose: This module takes the two inputs, state and key, and XORs then to write to the “out” output. This operation is needed in the decryption (and similarly in encryption).

Annotated Simulation of the AES decryptor (next page)


Wave - Default
Msgs
/testbench_lab9/CLK
/testbench_lab9/RESET
/testbench_lab9/AES_START
/testbench_lab9/AES_DONE
/testbench_lab9/AES_KEY
/testbench_lab9/AES_MSG_ENC
/testbench_lab9/AES_MSG_DEC
/testbench_lab9/aes0/CLK
/testbench_lab9/aes0/RESET
/testbench_lab9/aes0/AES_START
/testbench_lab9/aes0/AES_DONE
/testbench_lab9/aes0/AES_KEY
/testbench_lab9/aes0/AES_MSG_ENC
/testbench_lab9/aes0/AES_MSG_DEC
1
0
0
0
00000000000000...
11011010111011...
00001001010011...
St1
St0
St0
0
00010203040506...
daec3055df058e...
094a6e5e9be99...
000102030405060708090a0b0c0d0e0f
daec3055df058e1c39e814ea76f6747e
ece298dcece298dcece298dcece298dc
Now
8784720930000 ps
Cursor 1
1495900 ps
400000 ps
800000 ps
1200000 ps
1600000 ps
2000000 ps
2400000 ps
2800000 ps
1495900 ps

At the beginning of the simulation around time 0, we see that the AES_Message is set to ‘daec3055df058e1c39e814ea76f6747e’ along with AES_Key being set to ‘000102030405060708090a0b0c0d0e0f’. Then, AES_START signal goes high to begin passing the key and encoded message. We then enter the decryption states and loop the appropriate number of times through AddRoundKey and the inverse modules. At around time 1600 ns, we see the AES_DONE signal go high, after AES_START is already low. Decryption is complete and we have gone through all the states. Decrypted output is seen.

Post-Lab Question(s)

Design Resources and Statistics

LUT	5721
DSP	0
Memory (BRAM)	562,176 / 3,981,312 (14 %)
Flip-Flop	2989
Frequency (MHz)	117.1
Static Power (mW)	102.25
Dynamic Power (mW)	0.95
Total Power (mW)	174.23

- *Which would you expect to be faster to complete encryption/decryption, the software or hardware? Is this what your results show?*

The hardware, and specifically the decryption process we implemented here, is much faster than the software encryption that we implemented. The hardware implementation benefits from optimization and logic blocks that can process words in parallel fashion, with less delay between signals and less delay between “function” handoffs and simplified memory storage fashion. Software takes much more clock cycles to perform the operation.

- *If you wanted to speed up the hardware, what would you do? (Note: restrictions of this lab do not apply to answer this question)*

The hardware portion of the lab could be sped up by simply performing less state transitions and thus operational time by combining states and minimizing the state machine and our other glue logic that increases consumption. We could theoretically increase the size of the words or blocks that are handled by each module as well to decrease I/O times. We could also try forcing the clock frequency a bit higher until the output and functioning is no longer stable.

Conclusions

This lab was the last in our latest series of using the FPGA with the NIOS II processor to implement processes involving complex I/O and memory management along with the Avalon-MM (Memory Mapped) Bus. We ran into a bit of a struggle in our debugging in tracking down the problem with our hardware decryptor because of the complex interface of NIOS II and multiple modules. We ended up not progressing through the description steps to generate the output because on a case in the state on AES_START, our load enable bit was being set to high instead of low ‘ 1’b0 ’. We found the test bench to actually be more helpful in testing during our debugging for this lab than the last lab. We nocticed that our calculation for the column major order access for the matrices was reversed by looking at the state during InvMixColumns and AddRoundKey.