ECE 385

Spring 2019

Experiment #9

SOC with Advanced Encryption Standard in SystemVerilog

Rob Audino and Soham Karanjikar
Section ABJ, Friday 2:00-4:50
TA: David Zhang

Introduction:

These last two weeks, our task was to use System Verilog and C in order to implement 128 bit AES encryption and decryption as an Intellectual Property core. In the first week, we had to implement AES encryption on the software IP core through mainly C, and in the second week we had to implement AES decryption through System Verilog and design our own IP core. The IP core functioned very much like a software library in C would, allowing for repeated reuse across a range of applications. Incorporating the IP core with NIOS and C allowed us to create a System on Chip. The end goal of this lab was to input a key with an encrypted message, and produce a decrypted output accordingly. In summation, this lab taught us how to implement a goal (encryption and decryption) through hardware (SV) as well as software (C).

Written Description and Diagrams of the AES encryptor/decryptor

Written description of the software encryptor:

The NIOS processor does a little more than half of the work. The main reason for it is to encrypt the a message using a key with the AES algorithm. Once this key is encrypted, it is sent to a register that is accessible in hardware where the decryption is done. Another utility of the NIOS processor is to interface with the keyboard through the EZ-OTG USB Chip. Through using the JTAG interface we can use the console in eclipse to debug as well as fully run our AES Encryption/Decryption lab.

Written description of the hardware decryptor:

The hardware is decryption is done through passing the value through many different states, and in each state the value is changed. First the key expansion is done, which gives us the key schedule containing all the keys for every AddRoundKey we need to do. The first state after key expansion is “first”; in this round, the first AddRoundKey is done. Following thi,s there are 9 rounds, in which each of the operation is done once. To accomplish this, a counter is set up. The counter is the select signal to a 4:1 MUX which picks from the 4 inputs: INVSUBBYTES, INVSHIFTROWS, ADDROUNDKEY, MIXCOLUMNS, and saves it into a register each clock cycle. After this set, the final 3 operations are done and the decrypted value is saved in the register. This marks the end of the decryption in hardware, but not the end of the whole process. The final value is sent to the aes_interface and stored in 4 registers each of 32-bits which can be accessed through the C code (software). Once the final value is read is C, a DONE signal is sent to hardware which puts it back into the halted state. A design constraint we had was that we could only initiate the MIXCOLUMNS module once; to effectively get past this, we had to use another 4:1 MUX and another counter. We passed in 32-bits of the state each counter iteration and put the output of the mux into the correct 32-bit location using a 1:4 decoder with the same counter signal.

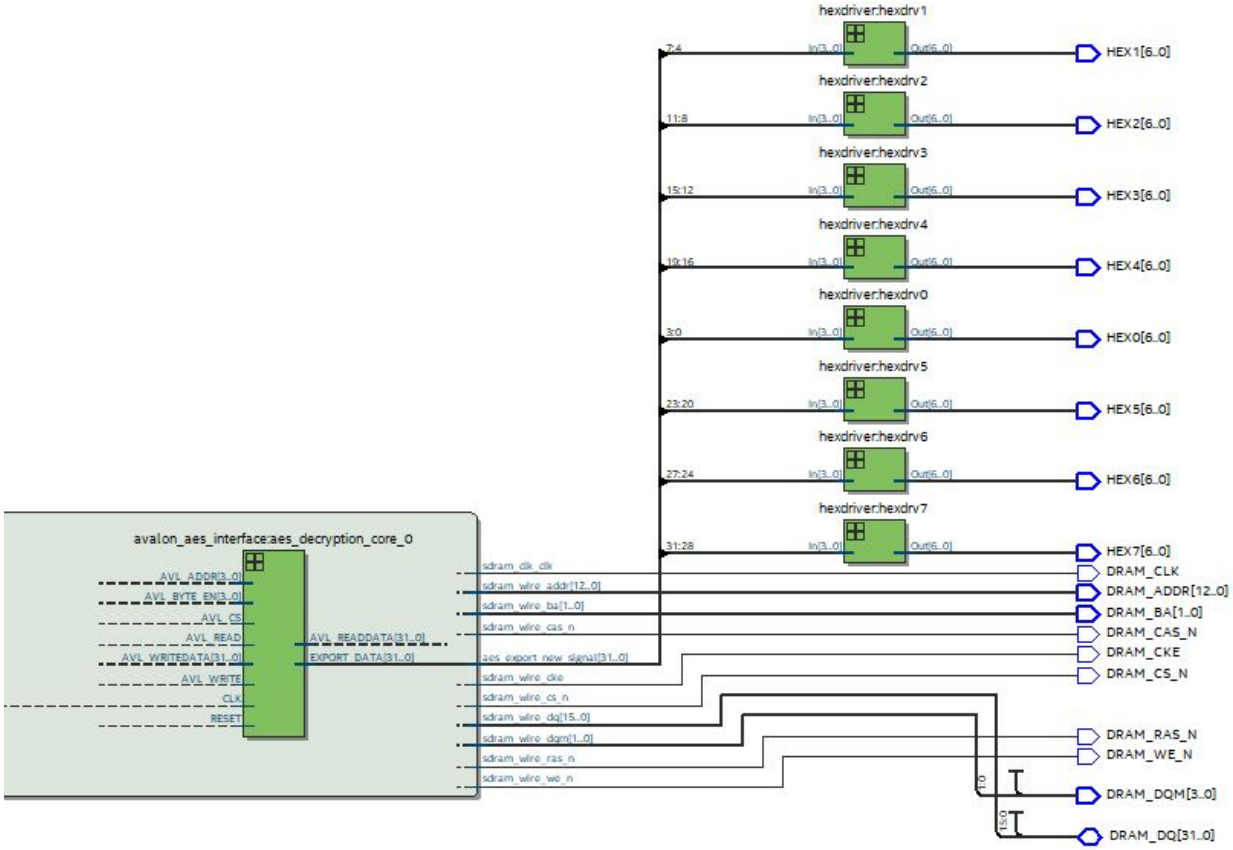
All the of the software/hardware communication is done through the Avalon Bus we made in platform designer. There is a pointer initiated in the C code which starts at the base address of the REGFILE created in hardware. The registers are accessible through modifying the pointer in C in software and just as an unpacked array in hardware.

Written description of the hardware/software interface (avalon_aes_interface.sv):

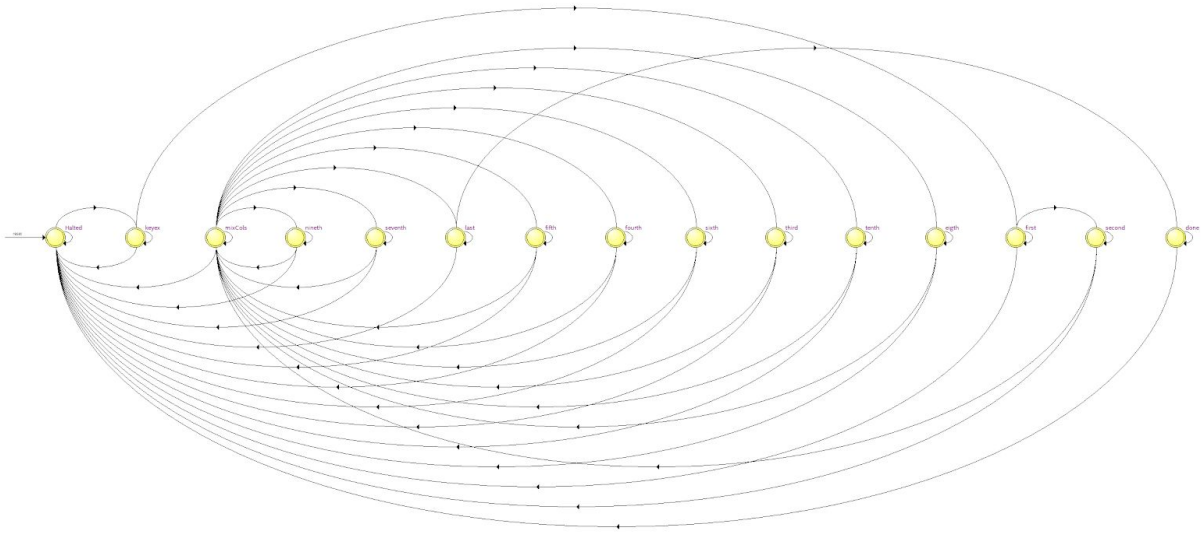
Our register file was designed as an array of size 16 comprised of 32 bit registers. 14 of the 16 arrays were mapped to a specific purpose within the lab. The first four registers were mapped to the AES_Key, since 4 32 bit registers are required to hold a key of size 128 bits. The second group of four registers was mapped to hold the AES Encrypted Message, which was also 128 bits. These first eight registers were used as an input to the decryption logic, but also had an in-out relationship with the avalon slave MM port. Registers 9-12 were used to hold the decrypted message, and had an input from the decryption logic and an in-out relationship with the avalon slave MM port. Registers 13 and 14 were unused, but register 15 was used to tell the Decryption logic when to start, and register 16 indicated when the Decryption logic was finished. We created the reset function to clear all of the contents of any particular register, and the read function to read the contents of any given register. However, the write function only allowed access to certain combinations of bits within the register, depending on byte enable. We were able to access the full 32 bits, the two upper bytes, the two lower bytes, and byte 0, 1, 2, or 3 depending on the corresponding signal from byte enable.

The design of the avalon_aes_interface module plays hand in hand with how the register file is designed, in terms of how it manages the connections between software and hardware. If the interface receives a “done” signal from the hardware decryptor, it will load the corresponding decrypted message into the four corresponding registers and mark the “done” register as done. In addition, the byte enable signal serves in a similar fashion as described for the register files above: each bit of the 4 bit signal enables a corresponding byte of the selected register. Reading from and writing to these registers were both controlled by the chip select signal, as well as the read and write enable signals respectively. In summation, the software doesn’t input new messages to the hardware unless it is informed that the hardware has received the previous part of the message; this is to remedy the fact that the message is 128 bits, but the port between hardware and software is only 8 bits wide.

Block diagram:



State Diagram of AES decryption controller:



Transitions:

	Source State	Destination State	Condition
1	done	done	(start).(!Reset)
2	done	Halted	(!start) + (start).(Reset)
3	eigth	Halted	(Reset)
4	eigth	eigth	(!c1[2]).(!c1[3]).(!Reset)
5	eigth	mixCols	(!c1[2]).(c1[3]).(!Reset) + (c1[2]).(!Reset)
6	fifth	fifth	(!c1[2]).(!c1[3]).(!Reset)
7	fifth	Halted	(Reset)
8	fifth	mixCols	(!c1[2]).(c1[3]).(!Reset) + (c1[2]).(!Reset)
9	first	Halted	(Reset)
10	first	first	(!c1[0]).(!c1[1]).(!Reset) + (!c1[0]).(c1[1]).(!c1[2]).(c1[3]).(!Reset) + (!c1[0]).(c1[1]).(c1[2]).(!Reset) + (c1[0]).(!Reset)
11	first	second	(!c1[0]).(c1[1]).(!c1[2]).(!c1[3]).(!Reset)
12	fourth	fourth	(!c1[2]).(!c1[3]).(!Reset)
13	fourth	Halted	(Reset)
14	fourth	mixCols	(!c1[2]).(c1[3]).(!Reset) + (c1[2]).(!Reset)
15	Halted	Halted	(!start) + (start).(Reset)
16	Halted	keyex	(start).(!Reset)

:

	Source State	Destination State	Condition
16	Halted	keyex	(start).(!Reset)
17	keyex	Halted	(Reset)
18	keyex	keyex	(!key_c[0]).(!key_c[1]).(!key_c[2]).(!key_c[3]).(!Reset) + (!key_c[0]).(!key_c[1]).(!key_c[2]).(key_c[3]).(!key_c[4]).(!Reset) + (!key_c[0]).(!key_c[1]).(key_c[2]).(!Reset) + (!key_c[0]).(key_c[1]).(!Reset) + (key_c[0]).(!Reset)
19	keyex	first	(!key_c[0]).(!key_c[1]).(!key_c[2]).(key_c[3]).(key_c[4]).(!Reset)
20	last	done	(!c1[2]).(c1[3]).(!Reset) + (c1[2]).(!Reset)
21	last	Halted	(Reset)
22	last	last	(!c1[2]).(!c1[3]).(!Reset)
23	mixCols	fifth	(!round[0]).(!round[1]).(round[2]).(!round[3]).(!c2[0]).(!c2[1]).(c2[2]).(!c2[3]).(!Reset)
24	mixCols	fourth	(round[0]).(round[1]).(!round[2]).(!round[3]).(!c2[0]).(!c2[1]).(c2[2]).(!c2[3]).(!Reset)
25	mixCols	Halted	(Reset)

	Source State	Destination State	Condition
26	mixCols	last	(!round[0]).(round[1]).(!round[2]).(round[3]).(!c2[0]).(!c2[1]).(c2[2]).(!c2[3]).(!Reset)
27	mixCols	seventh	(!round[0]).(round[1]).(round[2]).(!round[3]).(!c2[0]).(!c2[1]).(c2[2]).(!c2[3]).(!Reset)
28	mixCols	sixth	(round[0]).(!round[1]).(round[2]).(!round[3]).(!c2[0]).(!c2[1]).(c2[2]).(!c2[3]).(!Reset)
29	mixCols	tenth	(round[0]).(!round[1]).(!round[2]).(round[3]).(!c2[0]).(!c2[1]).(c2[2]).(!c2[3]).(!Reset)
30	mixCols	third	(!round[0]).(round[1]).(!round[2]).(!round[3]).(!c2[0]).(!c2[1]).(c2[2]).(!c2[3]).(!Reset)
31	mixCols	eigth	(round[0]).(round[1]).(round[2]).(!round[3]).(!c2[0]).(!c2[1]).(c2[2]).(!c2[3]).(!Reset)

	Source State	Destination State	Condition
32	mixCols	mixCols	(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!c2[0]).(!c2[1]).(!c2[2]).(!Reset) + (!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!c2[0]).(!c2[1]).(c2[2]).(c2[3]).(!Reset) + (!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!c2[0]).(c2[1]).(!Reset) + (!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(c2[0]).(!Reset) + (!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(Decoder0).(!Reset) + (!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(Decoder0).(!Reset) + (!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(Decoder0).(!Reset) + (!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(Decoder0).(!Reset) + (!Decoder0).(!Decoder0).(!Decoder0).(Decoder0).(!Reset) + (Decoder0).(!Reset)

	Source State	Destination State	Condition
33	mixCols	nineth	(!round[0]).(!round[1]).(!round[2]).(round[3]).(!c2[0]).(!c2[1]).(c2[2]).(!c2[3]).(!Reset)
34	nineth	Halted	(Reset)
35	nineth	mixCols	(!c1[2]).(c1[3]).(!Reset) + (c1[2]).(!Reset)
36	nineth	nineth	(!c1[2]).(!c1[3]).(!Reset)
37	second	Halted	(Reset)
38	second	mixCols	(!c1[2]).(c1[3]).(!Reset) + (c1[2]).(!Reset)
39	second	second	(!c1[2]).(!c1[3]).(!Reset)
40	seventh	Halted	(Reset)
41	seventh	seventh	(!c1[2]).(!c1[3]).(!Reset)
42	seventh	mixCols	(!c1[2]).(c1[3]).(!Reset) + (c1[2]).(!Reset)
43	sixth	Halted	(Reset)
44	sixth	sixth	(!c1[2]).(!c1[3]).(!Reset)
45	sixth	mixCols	(!c1[2]).(c1[3]).(!Reset) + (c1[2]).(!Reset)
46	tenth	Halted	(Reset)
47	tenth	tenth	(!c1[2]).(!c1[3]).(!Reset)
48	tenth	mixCols	(!c1[2]).(c1[3]).(!Reset) + (c1[2]).(!Reset)
49	third	Halted	(Reset)
50	third	third	(!c1[2]).(!c1[3]).(!Reset)
51	third	mixCols	(!c1[2]).(c1[3]).(!Reset) + (c1[2]).(!Reset)

Module Descriptions:

Module: four_one
Inputs: [127:0] A,B,C,D, [1:0] S
Outputs: [127:0] Out
Description: This is a four to one multiplexer with a data width of 128.
Purpose: Used for the 4 different inputs to state register from 4 different operations that can be performed.

Module: four_one32
Inputs: [31:0] A,B,C,D, [1:0] S
Outputs: [31:0] Out

Description: This is a four to one multiplexer with a data width of 32.
Purpose: This is a mux for the mixcolumns operation because we were only allowed to instantiate one of these modules. It takes 32 bit wide segments of the state and passes it into the mix columns module once at a time.

Module: one_four32
Inputs: [31:0] in, clk, [1:0] S
Outputs: [127:0] A
Description: This is a one to four decoder with width 32.
Purpose: This is a decoder for the mixcolumns operation to put the 32-bit wide segments back into the state register according to the correct counter.

Module: counter
Inputs: [3:0] in, inc, clk, Reset
Outputs: [3:0] out
Description: This is a counter that counts from 0 up to 16.
Purpose: This counter goes from 0 to various values depending on how high we need to count.

Module: rcounter
Inputs: [3:0] in, inc, clk, Reset
Outputs: [3:0] out
Description: This is a counter that counts to 10.
Purpose: This is the rounds counter that we use, just like the counter module, to count the number of rounds completed.

Module: keycounter
Inputs: inc, clk, Reset
Outputs: [4:0] out
Description: This is a counter that counts to 24.
Purpose: This is a keycounter that makes sure the correct roundkey from the key schedule is used while performing the addroundkey operation.

Module: AES_CONTROL
Inputs: CLK, Reset, start, [3:0] c1, [3:0] c2, [4:0] key_c, [3:0] round
Outputs: i_c1, r_c1, i_c2, r_c2, i_k, r_k, LD_S, done_s, i_round, r_round, firstARK
Description: This module controls the state machine for decryption.
Purpose: This is the whole state machine which controls all the signals to the registers, muxes, and counters. This makes sure that all the steps and operations are done, and completed in the correct order. This is the brains of the decryption.

Module: avalon_aes_interface
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: This module controls the interaction between hardware and software.
Purpose: This is a bus interface that connects the hardware and software. The main point of this is to store the key and message that are inputted into NIOS and stored into the REGFILE which is then accessible in hardware. This works the same way for sending data back from software to hardware because the registers are still same.

Module: SubBytes
Inputs: clk, [7:0] in
Outputs: [7:0] out
Description: This module does the invSubBytes Operation.
Purpose: Since we had no constraint on this module we just instantiated 16 of them to complete the invSubBytes operation.

Module: InvSubBytes
Inputs: clk, [7:0] in
Outputs: [7:0] out
Description: This module uses the subBytes module to do the whole 128 bits.
Purpose: This module instantiates 16 subbytes modules to do the whole 128 bits and sends it as an input to the 4:1 Mux.

Module: InvSubMsg
Inputs: clk, [127:0] in
Outputs: [127:0] out
Description: Completes the InvSubMsg module.
Purpose: This takes the 128bits from state register as an input and outputs 128 bits to the 4:1 MUX after completing the operation.

Module: lab9_top
Inputs: [31:0] DRAM_DQ, 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, [31:0] DRAM_DQ, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_WE_N, DRAM_CLK
Description:
Purpose:

Module: KeyExpansion
Inputs: clk, [127:0] Cipherkey
Outputs: [1407:0] KeySchedule
Description: This module performs the inverse KeyExpansion that was implemented in software.
Purpose: This module is used for the decryption of the message, just as the original key expansion is used for the encryption of the message.

Module: KeyExpansionOne
Inputs: clk, [127:0] oldkey, [7:0] Rcon
Outputs: [127:0] newkey
Description: This module completes the first addroundkey operation
Purpose: This takes the input key and outputs the first addroundkey that is used in the rounds. Since XOR is a reversible operation, it is the same as the encryption Addroundkey = Invaddroundkey.

Module: AddRoundKey
Inputs: [127:0] msg, [1407:0] keyS, [3:0] round
Outputs: [127:0] out
Description: This does the AddRoundKey Operation
Purpose: Takes the correct key from keyschedule depending on the round which is a counter and completes the addroundkey operation which is an XOR. The output from this module is sent to the 4:1 MUX.

Module: InvShiftRows
Inputs: [127:0] data_in
Outputs: [127:0] data_out
Description: This module does the InvShiftRows operation.
Purpose: Takes the input as current state and send the output, which is the new state after the operation has been compeltes, to the 4:1 MUX.

Module: InvMixColumns
Inputs: [31:0] in
Outputs: [31:0] out
Description: This module does the InvMixColumns operation on 32 bits at a time.
Purpose: The purpose of this module is to do the InvMixColumns operation, however we were only allowed to instantiate this once. To account for this we had to make a counter that stayed in the MixColumns state for 4 cycles to do each of the 32 bits from state. The input of this module is taken from a 4:1 MUX which divided the 128 bits into 4 32-bit segments. The output of this module is sent to a 1:4 decoder which puts the 32-bits into the correct place. The signals to the 1:4 decoder and 4:1 MUX is the same.

Module: hexdriver
Inputs: [3:0] In
Outputs: [6:0] Out
Description: This module drives the HEX LEDS on the board.
Purpose: We used this to show some parts of the encrypted message in week 1 and just testing in week 2 as we did not have to display anything on the LEDS.

Module: AES
Inputs: CLK, RESET, AES_START, [127:0] AES_KEY, [127:0] AES_MSG_ENC
Outputs: AES_DONE, [127:0] AES_MSG_DEC
Description: This is the top level for the encryption part and what we tested in testbench.
Purpose: This module instantiates all the registers, muxes, decoders and counters we need. It also instantiates the AES_CONTROL which is the state machine and sends the control signals to all the other modules instantiated in AES.

Module: REGS
Inputs: CLK, RESET, AVL_WRITE, [3:0] AVL_BYTE_EN, [3:0] AVL_ADDR, AVL_READ, AVL_CS, [31:0] DATA_IN
Outputs: [31:0] DATA_OUT
Description: This module does the interfacing between getting data from software and hardware.
Purpose: This takes the data from software and stores it into a register. The hardware does not need to use this module as it can directly access the REGFILE. However, when the software wants to read data from the hardware this module is used. As the in/out data is taken/put from/into the REGFILE.

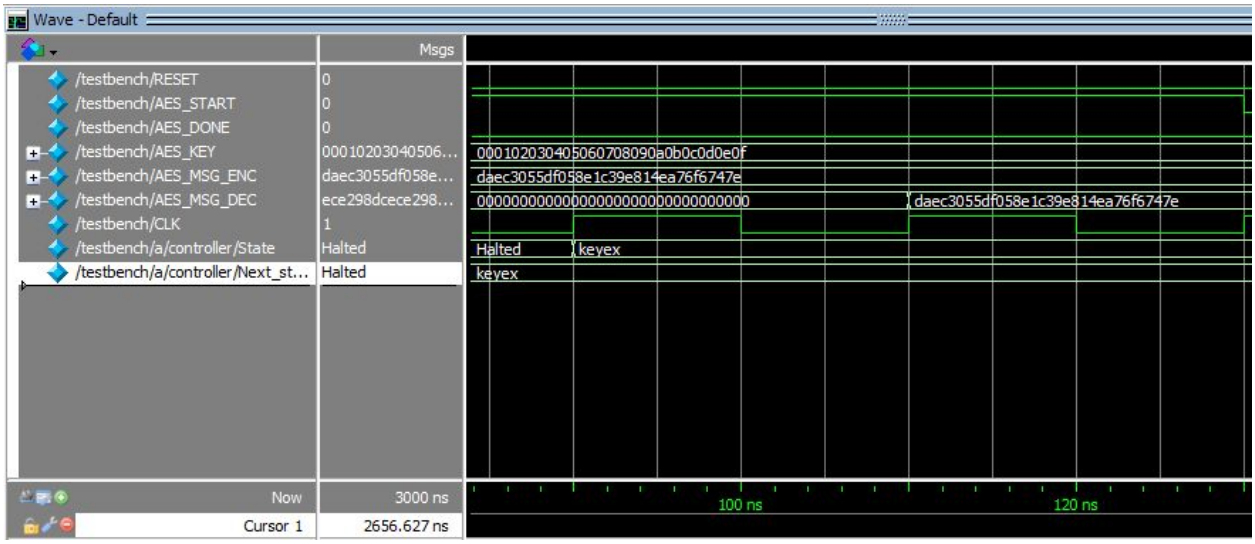
Module: reg_128
Inputs: clk, Reset, Load, [127:0] D, [127:0] d2
Outputs: [127:0] Data_Out
Description: 128 bit register
Purpose: This register is used to hold the state in our decryption. The output of this is sent to the modules and the input comes from the 4:1 mux that has all the operation modules' outputs connected to it.

Module: eightreg_16
Inputs: clk, Reset, Load, [15:0] D, [2:0] SR1IN, [2:0] SR2, [2:0] DRIN
Outputs: [15:0] SR2O, [15:0] SR1O
Description: eight registers stored as an unpacked array.
Purpose: This is used in the software hardware interface as all the registers used are instantiated here at once.

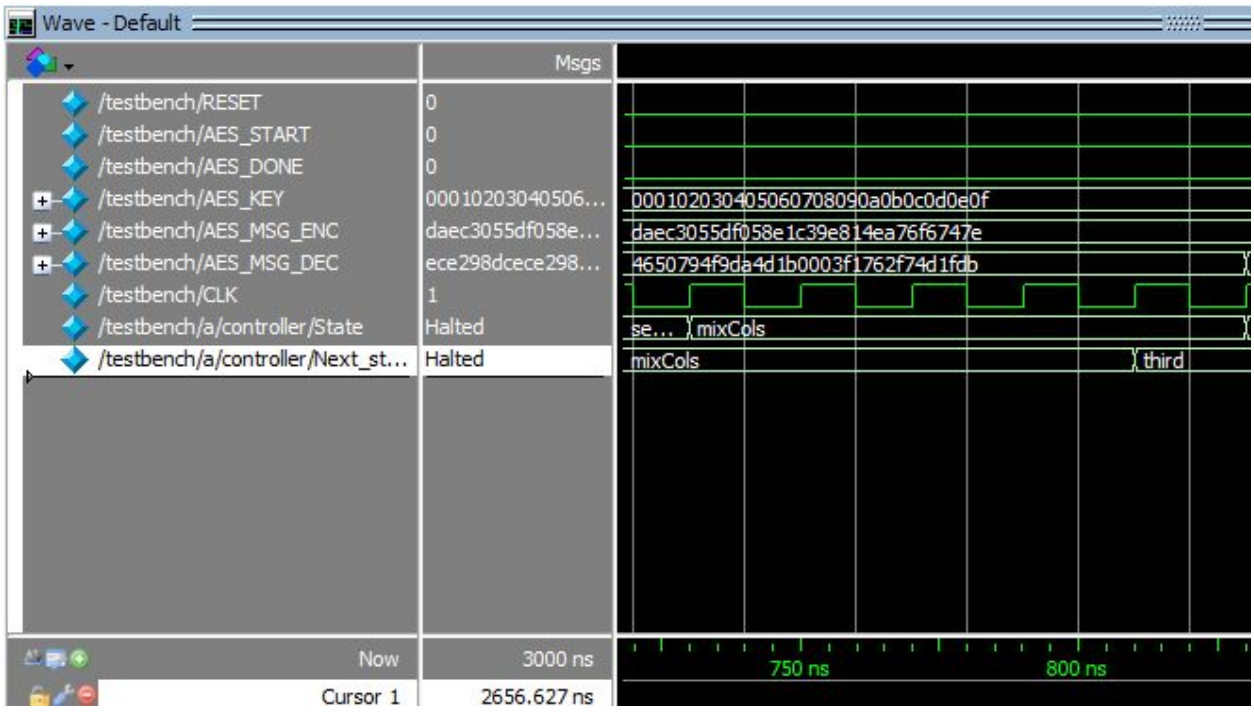
Module: lab8_soc
Inputs: clk_clk, reset_reset_n, reset_wire_export, [15:0] sdram_wire_dq
Outputs: [31:0] aes_export_new_signal, sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [1:0] sdram_wire_dqn, sdram_wire_ras_n, sdram_wire_we_n
Description: This module connects NIOS to the FPGA, and is generated by our Platform designer.
Purpose: This module allows us to run C on our FPGA board.

Annotated Simulation of the AES decryption:

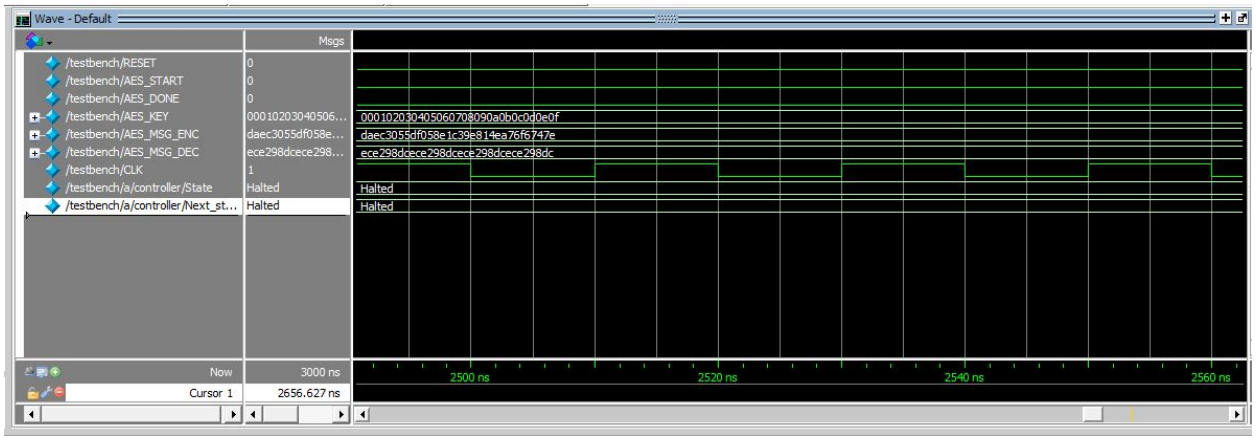
Screenshot 1: Key expansion begins, and the encrypted message is loaded into AES_MSG_DEC.



Screenshot 2: The first round of decryption is finished, and the intermediate message appears in AES_MSG_DEC.



Screenshot 3: The state machine is at a Halt state and the final results are shown.



Post-Lab Questions:

1. Q: Fill out the design resources and statistics table (duplicated here for convenience).
A:

LUT	5922
DSP	0
Memory (BRAM)	126080
Flip-Flop	2823
Frequency	103.11 MHz
Static Power	102.28 mW
Dynamic Power	0.87 mW
Total Power	175.68 mW

2. Q: Which would you expect to be faster to complete encryption/decryption, the software or hardware? Is this what your results show? (List your encryption and decryption benchmark here)
A: We can assume that the hardware will complete decryption faster than the software can complete encryption because in general, hardware will always be faster than software. This is because software runs on hardware, and any instructions in software have to wait on the requisite hardware to run. Our benchmark shows the same, and is listed below.

Benchmark:

```
fd949b3
64068bc
d5d1778

d6dea965
46b6b758
bf4e2e8b
21c8c02e
Software Encryption Speed: 1.934236 KB/s
Hardware Encryption Speed: 333.333333 KB/s
```

3. Q: If you wanted to speed up the hardware, what would you do? (Note: restrictions of this lab do not apply to answer this question)
A: The simplest way we could speed up the hardware would be to instantiate multiple instances of the InvMixColumns module. This would allow us to perform our calculations faster, since the state machine would be simplified and InvMixColumns isn't dependent on the output of a previous InvMixColumns instantiation.

Conclusion:

Our project was able to work perfectly for both parts of the project. Upon demoing the encryption in the first week, we found that our encryption was an order of magnitude faster than the other groups. At first, we thought our benchmark code had an issue; however, upon further inspection, we had simply devised a more efficient algorithm to complete the encryption. When we were testing the decryptor for the second week, we had two issues. Firstly, it took us several days to notice that we were in fact testing the encryptor with the testbench, and not the decryptor. After fixing this, we were still presented with another error, which produced the incorrect decrypted message in NIOS but the correct decrypted message in the testbench. We corrected the way we constructed the key from the key schedule and this ended up fixing the issue entirely.

In summary, this lab was interesting because it showed us how to use both hardware and software to do and undo a process: in this case, AES encryption. It also helped us learn that although implementing something through software might be a simpler solution given our previous experience with C, implementation through hardware is more efficient and much faster. The hardest part of the lab was understanding and implementing the state diagram, but after watching several videos on the AES process, we were able to glean how it should be put together.