

ECE 385
Fall 2019
Experiment #9

SOC With Advanced Encryption Standard in SystemVerilog

Pouya Akbarzadeh and Patrick Stach
Section: ABC
Lab TA: Vikram Anjur, Yuhong Li

Introduction

In experiment we implemented a 128-bit Advanced Encryption Standard (AES) using SystemVerilog as an Intellectual Property (IP) core. During the first week, we had to design and implement 128-bit AES encryption on the software IP core. During the second week we built the decryption in SystemVerilog + Platform Designer and designed the hardware IP core. In both the encryption and decryption we input a message and a key, and This lab helped us learn how to use C code with the beloved NIOS II processor and with SystemVerilog create a SoC. The goal of the lab was to create a circuit that is able to decrypt a message with a specific key and to recreate the original message.

AES Description Encryptor/Decryptor

Software Encryption

The user is prompted to enter a message in the console. The message is 32 characters, for example: "daec3055df058e1c39e814ea76f6747e". That message will be stored into an array of unsigned chars called msg_ascii. The user is then prompted to enter a key in the console. That key is also 32 characters, for example: "000102030405060708090a0b0c0d0e0f". That message will be stored into an array of unsigned chars called key_ascii. Note that we put these values in column major format.

```
void encrypt(unsigned char * msg_ascii, unsigned char * key_ascii,
             unsigned int * msg_enc, unsigned int * key)
```

When we call our encrypt function, the first thing we need to do is convert the arrays of ascii characters to a more usable format. Since there are 32 characters, there are arrays of 32 ascii values (which are each 8 bits, also note we are ignoring null character at end). In order to do calculations, we need to convert these ascii values into actual numbers, and this is done in charsToHex, which convert two characters to the byte values they represent. This is done in a loop for both the message and key so that we convert both 32-length unsigned char arrays to 16-length unsigned char arrays.

Next, we have call a KeyExpansion function which will take the initial round key (which is now an 16-length unsigned char array) and create a 176-length unsigned char array based on the Key Schedule algorithm in the AES encryption standard.

We create an initial 16-length unsigned char array called state which is used to store intermediate calculations, and we call AddRoundkey function (called xorThis2) for the 0th round, which returns an array with the current key in the key expansion XORed with the state. We then loop through 9 times, in each loop calling SubBytes (uses look up table), ShiftRows (returns an array with the order based

on the rearrangement of the ShiftRows algorithm), MixColumns (uses look up table), then AddRoundKey again. For the last round, we call SubBytes, ShiftRows, and AddRoundKey again.

Now we have two length-16 unsigned char arrays and we need to convert that to length-4 unsigned int arrays. We do this with bitwise shift and bitwise or to store into msg_enc and key, which the pointers to them we passed into the function.

After we do that, we write the key into the registers in aes_avalon_interface by dereferencing the AES_PTR (pointer to base address) in the 4 different spots (index 0,1,2,3).

Hardware Decryption

Our hardware decryption was done in our module called AES.sv.

```
module AES (  
    input  logic CLK,  
    input  logic RESET,  
    input  logic AES_START,  
    output logic AES_DONE,  
    input  logic [127:0] AES_KEY,  
    input  logic [127:0] AES_MSG_ENC,  
    output logic [127:0] AES_MSG_DEC  
);
```

We have instantiate 4 different base modules for our operations AddRoundKey, InvShiftRows, InvMixColumns_128, and InvSubBytes_128.

The AddRoundKey first has a mux that selects which portion of the key schedule to use as the current key (based on the round), then outputs the state that was inputted XOR'd with the selected state. This operates on 128 bits so we do not need to make any additional modifications.

InvShiftRows takes a 128bit input and using assign statements will wire all 128bit inputs to the correct output position after the InvShiftRows algorithm has taken place.

InvMixColumns_128 instantiates InvMixColumns, which performs the InvMixColumns operation on 32 bits. InvMixColumns_128 also instantiates a 4:1 Mux, which has all 128 bits of the state on the input, a 2 bit select bit controlled by the FSM to loop through all 128 bits, and a 32 bit output which goes into the 32 bit input of the InvMixColumns. InvMixColumns_128 has an

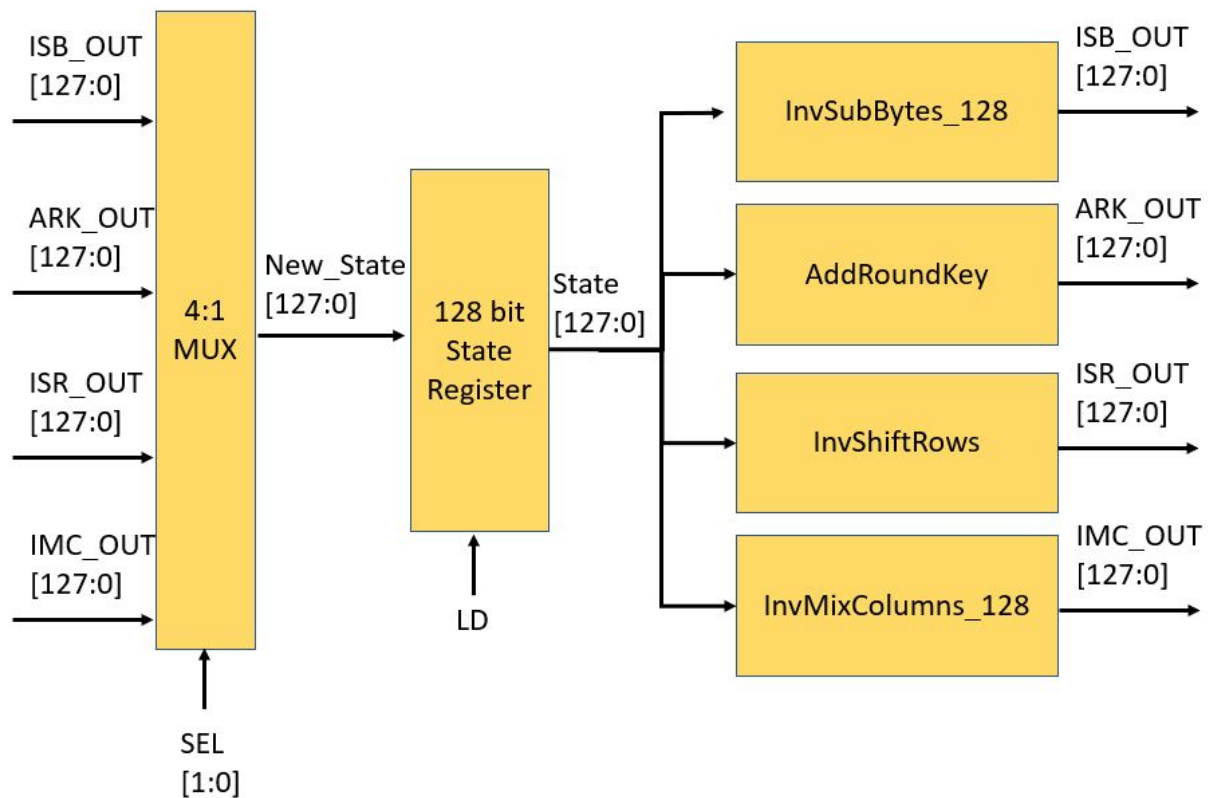
internal 128 bit register which will store the intermediate InvMixColumns 32 bit output in the specific 32 bit output corresponding to that select signal.

InvSubBytes_128 instantiates 16 InvSubBytes modules, in which each operate on a specific 8 bit section of the 128 bit input, similarly connecting the 16 8-bit outputs to the single 128-bit output. InvSubBytes on the rising edge of the clock will output the 8 bits corresponding to the S-box substitution table input.

We also instantiate a 4:1 mux that takes 4 inputs, one from each of the output of the 4 base operation modules. Based on a select bit from the FSM, it will route the specific signal to load into the state register.

We also instantiate a 128-bit register which will store the intermediate results from the operations. The output of this is connected into the input of each of the 4 operation modules. It has a LD signal controlled by the FSM

Here is a higher level view of the modules instantiated in AES.sv and how they are connected



We enumerate all the steps of the AES algorithm, and based on each state we will output specific control signals to choose between which of the operational modules we want to load into the intermediate state register. This is our finite state machine.

Hardware/Software Interface

The `avalon_aes_interface` module is the interface that connects the hardware and software.

Here we instantiate 16 32-bit registers, each responsible for storing data important to the encryption:

- 0-3 : 4x 32bit AES Key
- 4-7 : 4x 32bit AES Encrypted Message
- 8-11: 4x 32bit AES Decrypted Message
- 12: Not Used
- 13: Not Used
- 14: 32bit Start Register
- 15: 32bit Done Register

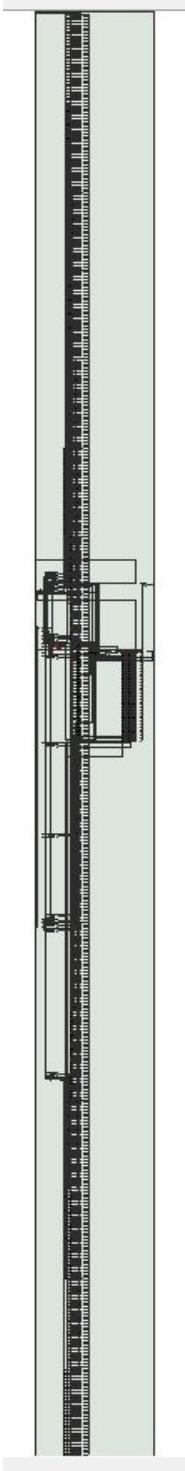
Here we instantiate AES, the module with state machine responsible for hardware decryption, and wire it to its respective inputs and outputs.

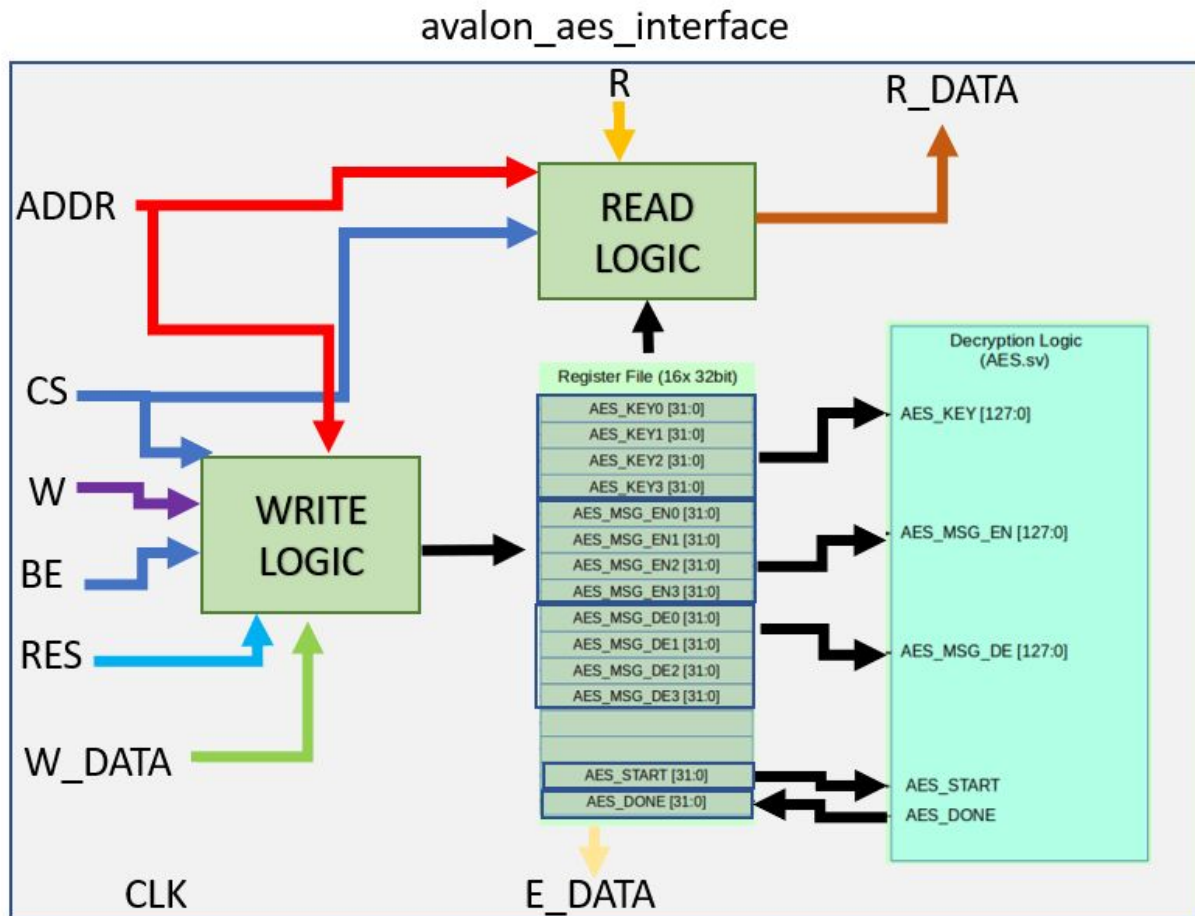
This module defines the way the Avalon-MM slave port will perform read and write operations requested from the NIOS II with specific control signals corresponding to enabling READ, WRITE, CS, BYTEENABLE, etc.

With these read and write operations defined, the NIOS II will be able to both read and write into these 16 registers which is essential for the hardware/software handshake. The software writes “1” into the start register (14) to indicate to the AES module to start decrypting, and keeps reading from the done register (15) until it sees AES module wrote “1”, indicating the finish of the decryption process.

Block Diagram

The way our design was synthesized essentially made the RTL view of the `avalon_aes_interface.sv` to be unreadable. A screenshot is attached of the RTL view of `avalon_aes_interface.sv` for purposes of the report requirements but we have another block diagram we created to describe the way we implemented it in code





In terms of the `avalon_aes_interface` file:

- CLK does not have specific arrows because it is used throughout the whole module because it is synchronous
- RESET = RES, which is used in our write logic because during a reset we should write "0" in all the registers
- AVL_READ=R, which is used in our read logic to determine if we are performing a read operation
- AVL_WRITE=W, which is used in our write logic to determine if we are performing a write operation
- AVL_CS=CS, which is essentially an enable pin for this module
- AVL_BYTE_N= BE, which picks which bytes should be written
- AVL_ADDR= ADDR, which picks which register to write and read to/from
- AVL_READ_DATA=R_DATA, which is the output for the data read
- AVL_WRITE_DATA=W_DATA, which is the input for the data to write
- AVL_EXPORT_DATA=E_DATA, which is the exported signal to the hex displays

```

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

    // Avalon Reset Input
    input logic RESET,

    // Avalon-MM Slave Signals
    input logic AVL_READ,           // Avalon-MM Read
    input logic AVL_WRITE,          // Avalon-MM Write
    input logic AVL_CS,             // Avalon-MM Chip Select
    input logic [3:0] AVL_BYTE_EN,  // Avalon-MM Byte Enable
    input logic [3:0] AVL_ADDR,     // Avalon-MM Address
    input logic [31:0] AVL_WRITEDATA, // Avalon-MM Write Data
    output logic [31:0] AVL_READDATA, // Avalon-MM Read Data

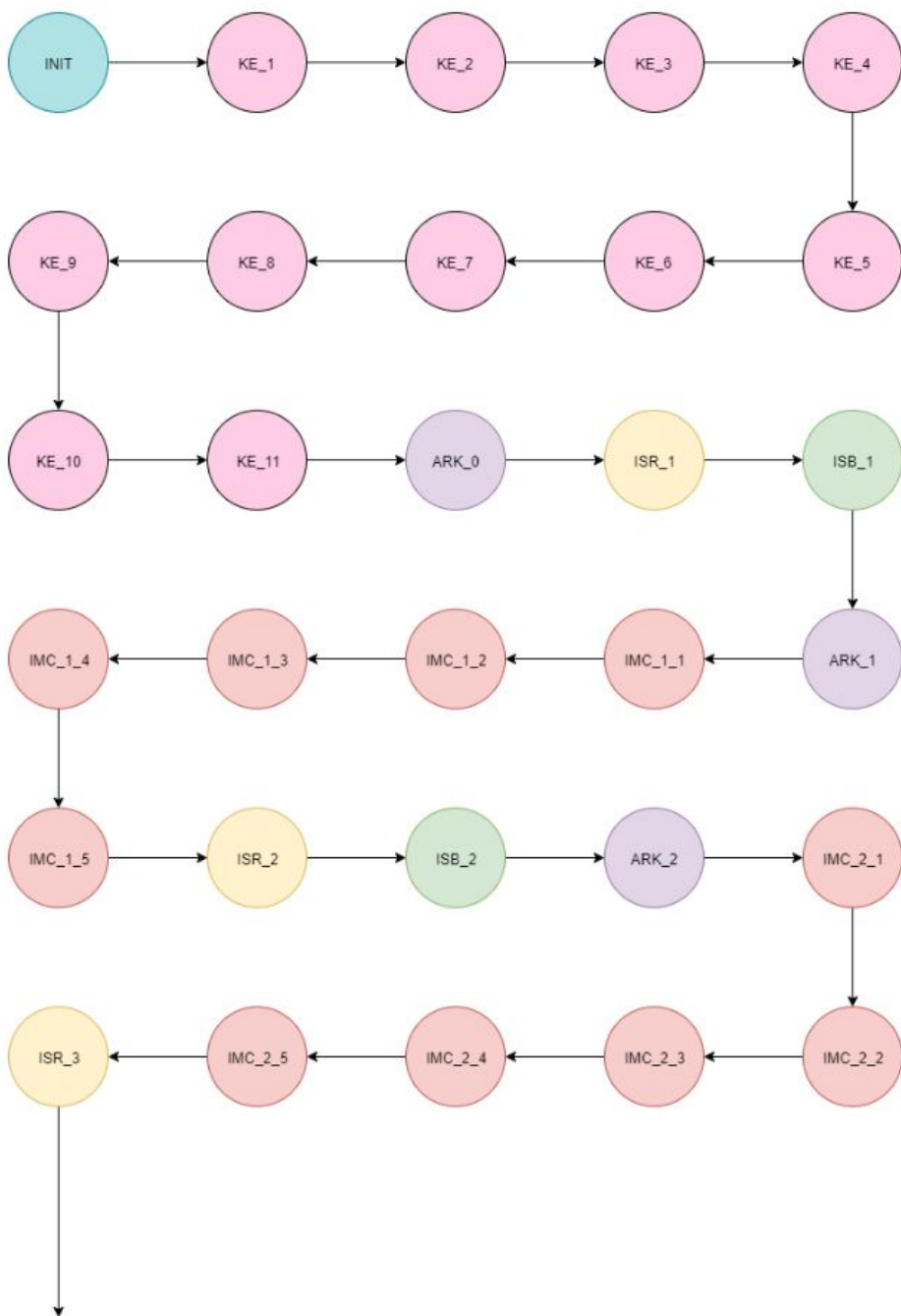
    // Exported Conduit
    output logic [31:0] EXPORT_DATA // Exported Conduit Signal to LEDs
);

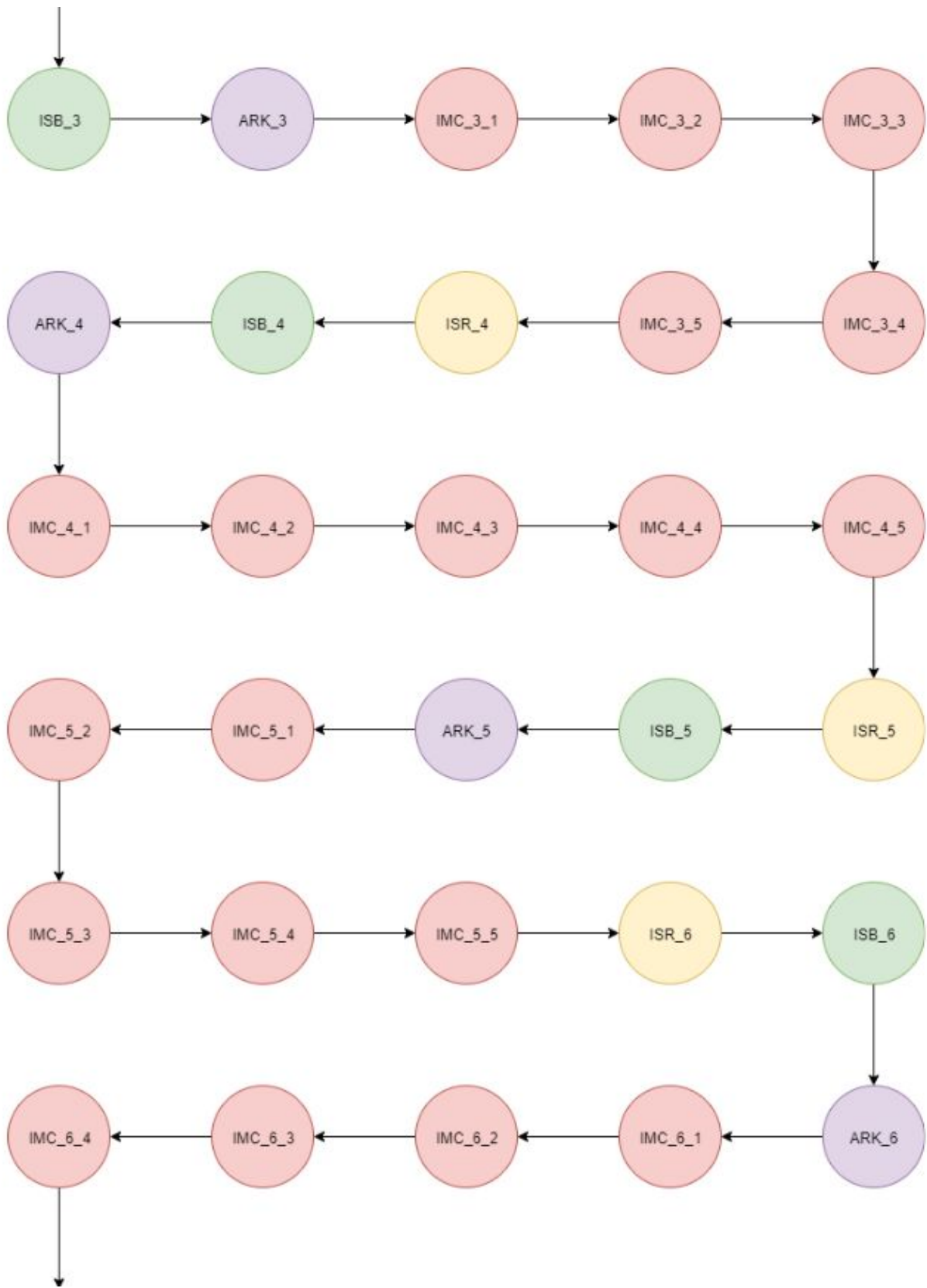
```

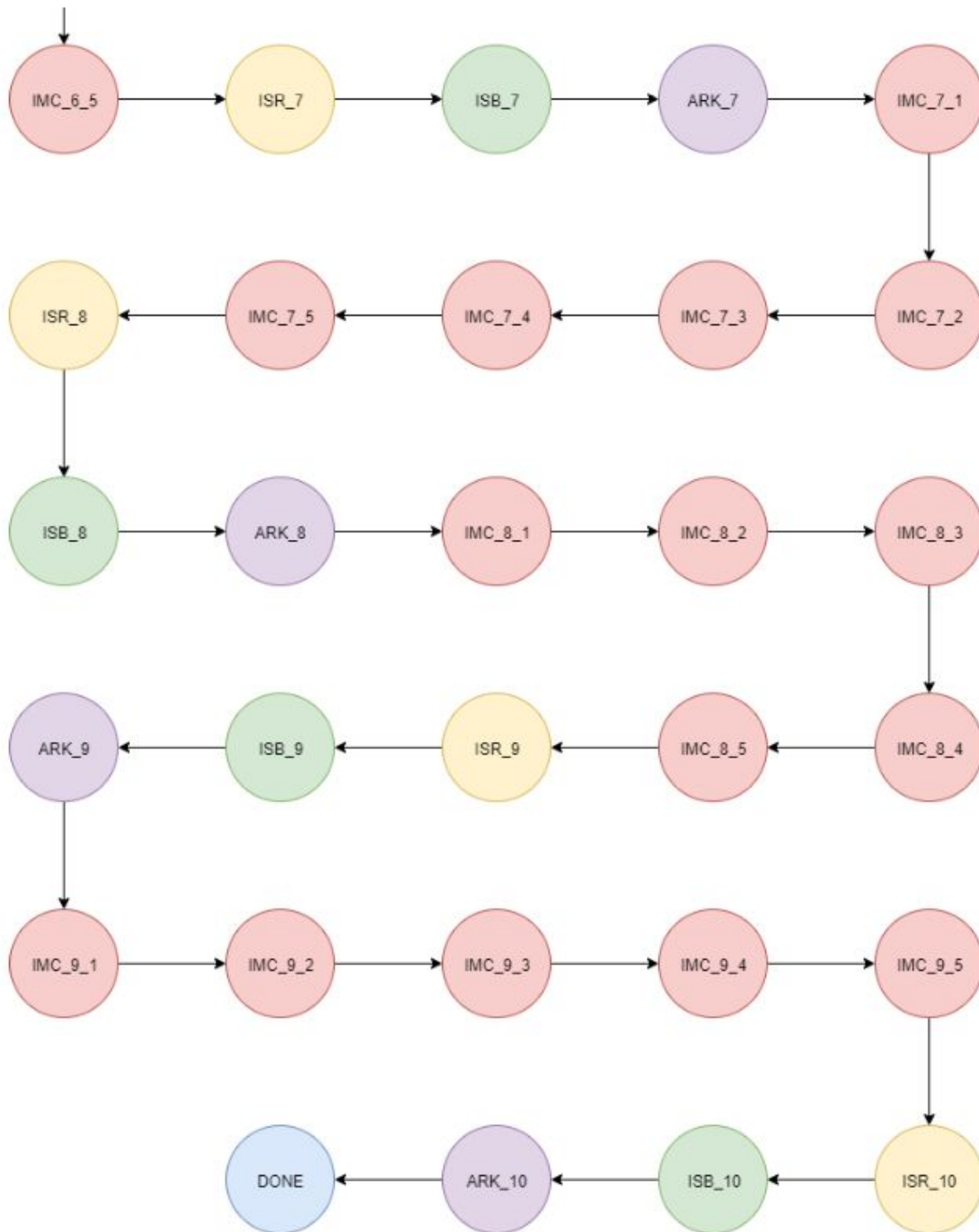
State Diagram of AES Decryptor Controller

In our FSM, we treated every single step as a separate state rather than using a counter. We did this because introducing a counter would add unnecessary complexity and constraints to our design, and there were few enough states that having each step be a state wouldn't take too much extra work.

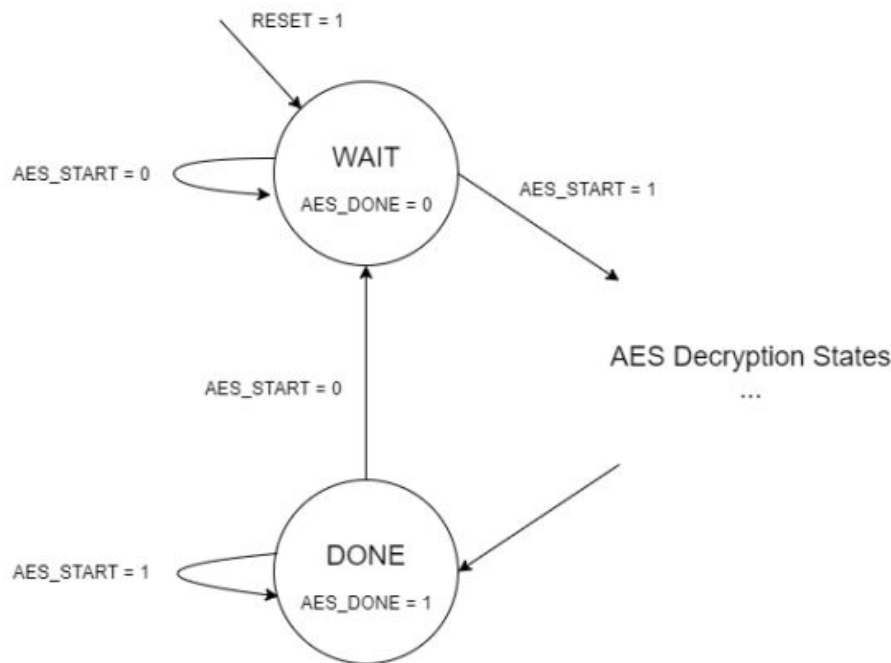
I.E: The 1st round of AddRoundKey is ARK_1, the 5th round of InverseShiftBytes is ISB_4, and the 3rd step of the 4th round of InverseMixColumns is IMC_4_3 (InverseMixColumns operation takes multiple cycles since we are restricted to only operating on 32 bits at a time).







For the sake of simplicity we also excluded the extra transitions in the start and done states in this state diagram. This is the behaviour of the start and done states:



If we are in the wait state, as long as AES_START = 0, we will stay in the wait state. If we are in the wait state and AES_START = 1, we will transition to the first state in AES decryption, in our case, "INIT". In the Decryption States, on every rising edge of the clock the state will transition to the next state regardless of the value of AES_START until DONE=1. This is because we do not want to stop in the middle of decryption, we need to finish it first. The only exception is the reset signal, which will result in the next state transition to be "Wait" and reset all signals and registers.

In the done state, as long as AES_START =1, we will stay in the done state. When AES_START=0 we will transition to the wait state and stay there until AES_START =1;

Module Descriptions (Quartus)

AES

```

module AES (
    input logic CLK,
    input logic RESET,
    input logic AES_START,
    output logic AES_DONE,
    input logic [127:0] AES_KEY,
    input logic [127:0] AES_MSG_ENC,
    output logic [127:0] AES_MSG_DEC
);
  
```

The AES.sv file is the heart of the decryption process: this includes the finite state machine used to sequentially walk through each step of the decryption process. With a given start signal (from software), it will take the give 128bit key and encrypted message and after it is finished decrypting it will output the decrypted message and output a “Done” signal of 1.

The AES.sv module is not only needed for actually decrypting the message, but also being part of the hardware/software handshake by signalling decryption is done by outputting a “Done” signal of 1.

Avalon_aes_interface

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

    // Avalon Reset Input
    input logic RESET,

    // Avalon-MM Slave Signals
    input logic AVL_READ,           // Avalon-MM Read
    input logic AVL_WRITE,         // Avalon-MM Write
    input logic AVL_CS,            // Avalon-MM Chip Select
    input logic [3:0] AVL_BYTE_EN, // Avalon-MM Byte Enable
    input logic [3:0] AVL_ADDR,    // Avalon-MM Address
    input logic [31:0] AVL_WRITEDATA, // Avalon-MM Write Data
    output logic [31:0] AVL_READDATA, // Avalon-MM Read Data

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

The avalon_aes_interface module is used as an interface that both the hardware and software share to communicate to each other. This interface includes an array of 16 32-bit registers which store the key (registers 0-3), encrypted message (registers 4-7), decrypted message (registers 8-11), and start (register 14)/ done (register 15) signals. This module defines how the Avalon-MM slave port will perform read and write operations requested from its master, the NIOS II processor.

This module is the key to the hardware/software communication because the AES.sv module will take specific registers in the avalon_aes_interface as inputs and outputs, while in our software we will access these registers by dereferencing the base pointer in Platform designer plus the offset for the specific register.

AddRoundKey

```
module AddRoundKey(input logic [127:0] State, input logic [1407:0] KeySchedule,
    input logic [3:0] Round, output logic [127:0] data_out);
```

This module uses Round as a select bit to determine which section of data from the KeySchedule for the current key (curr_key). It then outputs curr_key XOR'd with State. We use

this module for performing the AddRoundKey operation on the state in the AES decryption algorithm.

InvSubBytes

```
module InvSubBytes (  
    input  logic clk,  
    input  logic [7:0] in,  
    output logic [7:0] out  
);
```

On the rising edge of the clock the output will go to the respective S-box replacement of the input. This module is synthesized as RAM. We use this module for performing the InvSubBytes operation on a single byte.

InvSubBytes_128

```
module InvSubBytes_128 (input logic [127:0] in, input logic Clk, output logic [127:0] out);
```

This module instantiates 16 InvSubBytes modules and runs them in parallel for 128 bit operation. We use this module for performing the InvSubBytes operation on the state in the AES decryption algorithm.

InvMixColumns

```
module InvMixColumns (  
    input  logic [31:0] in,  
    output logic [31:0] out  
);
```

InvMixColumns uses XOR gates to perform the InvMixColumns operation on a 32 bit input, with the calculated 32 bit output as a result. We use this module for performing the InvMixColumns operation on a word in the AES decryption algorithm.

InvMixColumns_128

```
]module InvMixColumns_128(input logic [127:0] in,  
    input logic [1:0] sel,  
    input logic Reset, Clk, Load,  
    output logic [127:0] out);
```

InvMixColumns_128 has a single InvMixColumns module instantiated in addition to a 32 bit 4:1 mux. The mux form a select bit selects which 32 bit section of the current 128bit state to output to the InvMixColumns module since it can only operate on 32 bits at a time. This module also has an internal 128 bit register which stores the output of the internal InvMixColumns module and stores it in the respective 32 bit section based on the select bit. We use this module for performing the InvMixColumns operation on the state the AES decryption algorithm.

InvShiftRows

```
module InvShiftRows (input logic [127:0] data_in, output logic [127:0] data_out);
```

InvShiftRows uses assign statements to re-route the 128 input bits into the 128 output bits based on how the InvShiftRows algorithm would transform the ordering of the matrix. We use this module for performing the InvShiftRows operation on the state in the AES decryption algorithm.

lab9_top

```
module lab9_top (
    input logic    CLOCK_50,
    input logic [1:0] KEY,
    output logic [7:0] LEDG,
    output logic [17:0] LEDR,
    output logic [6:0] HEX0,
    output logic [6:0] HEX1,
    output logic [6:0] HEX2,
    output logic [6:0] HEX3,
    output logic [6:0] HEX4,
    output logic [6:0] HEX5,
    output logic [6:0] HEX6,
    output logic [6:0] HEX7,
    output logic [12:0] DRAM_ADDR,
    output logic [1:0] DRAM_BA,
    output logic    DRAM_CAS_N,
    output logic    DRAM_CKE,
    output logic    DRAM_CS_N,
    inout logic [31:0] DRAM_DQ,
    output logic [3:0] DRAM_DQM,
    output logic    DRAM_RAS_N,
    output logic    DRAM_WE_N,
    output logic    DRAM_CLK
);
```

This is the top level module: this is where all the modules that we will be using in our design need to be instantiated. The inputs and output names are matched up with the names in the Pin Planner to assign logic to specific pins corresponding to switches, leds, etc.

KeyExpansion

```
module KeyExpansion (
    input logic clk,
    input logic [127:0] Cipherkey,
    output logic [1407:0] KeySchedule
);
```

We use this module for performing the KeyExpansion operation on the state in the AES decryption algorithm.

HexDriver

```
module hexdriver (  
    input logic [3:0] In,  
    output logic [6:0] Out  
);
```

This module takes a 4 bit input outputs a 7 bit signal that will choose which specific segments to light up in order to display that character. We use this module for displaying part of the decrypted message onto the hex displays.

mux_128_4x1

```
module mux_128_4x1(  
    input logic [1:0] sel,  
    input logic [127:0] in0, in1, in2, in3,  
    output logic [127:0] out);
```

This module is a MUX- it takes a 4 128 bit inputs and based on the 2 bit select signal selects which to output. We use the module to select which of the four operation module outputs to load into the state register.

mux_32_4x1

```
module mux_32_4x1(  
    input logic [1:0] sel,  
    input logic [31:0] in0, in1, in2, in3,  
    output logic [31:0] out);
```

This module is a MUX- it takes a 4 32 bit inputs and based on the 2 bit select signal selects which to output. We use this module to select 32 bits out of the 128bits for the input of our InvMixColumns modules

Register_128

```
module Register_128  
(  
    input logic [127:0] in, preset_in,  
    input clk, reset, load, preset,  
    output logic [127:0] out );
```

This module is a register that will store data that is loaded in. If the load signal is high, it will take the data in the “in” pins and store it, and if the preset signal is high, it will take the data in the

“preset_in” pins and store it. It also has a synchronous reset. We use this module to store the intermediate states in the AES module.

lab8_soc


```

]module lab8_soc (
    output wire [31:0] aes_export_new_signal, // aes_export.new_signal
    input wire clk_clk, // clk.clk
    input wire reset_reset_n, // reset.reset_n
    output wire sdram_clk_clk, // sdram_clk.clk
    output wire [12:0] sdram_wire_addr, // sdram_wire.addr
    output wire [1:0] sdram_wire_ba, // .ba
    output wire sdram_wire_cas_n, // .cas_n
    output wire sdram_wire_cke, // .cke
    output wire sdram_wire_cs_n, // .cs_n
    inout wire [31:0] sdram_wire_dq, // .dq
    output wire [3:0] sdram_wire_dqm, // .dqm
    output wire sdram_wire_ras_n, // .ras_n
    output wire sdram_wire_we_n // .we_n
);


```

This module is the module that Platform Designer automatically generates when we press “Generate HDL”. It takes our visual design and writes it into this .sv code so it is in a format that we can actually compile and upload to the board. We use this module so we can synthesize our SoC from Platform designer and connect it to the rest of our hardware.

Module Descriptions (Platform Designer)

Name	Description	Export
 clk_0	Clock Source	
clk_in	Clock Input	clk
clk_in_reset	Reset Input	reset
clk	Clock Output	<i>Double</i>
clk_reset	Reset Output	<i>Double</i>

This module contains the clock from the 50 MHz oscillator on the FPGA. This clock is used to connect to other synchronous parts in both platform designer and both in quartus.

 nios2_gen2_0	Nios II Processor
clk	Clock Input
reset	Reset Input
data_master	Avalon Memory Mapped Master
instruction_master	Avalon Memory Mapped Master
irq	Interrupt Receiver
debug_reset_request	Reset Output
debug_mem_slave	Avalon Memory Mapped Slave
custom_instruction_m...	Custom Instruction Master

NIOS II is the processor that we are going to be using for both the encryption algorithm and for the keyboard I/O.

onchip_memory2_0	On-Chip Memory (RAM or ROM) Intel ...
clk1	Clock Input
s1	Avalon Memory Mapped Slave
reset1	Reset Input

This block is the on-chip memory, we have it as a placeholder but we never use it since we store our program on the SDRAM. We have it for future usage in the final project where we will need very fast storage to store a small amount of things (since the on chip memory is very small)

sdr	SDRAM Controller Intel FPGA IP	
clk	Clock Input	Double-click to
reset	Reset Input	Double-click to
s1	Avalon Memory Mapped Slave	Double-click to
wire	Conduit	sdr_wire

This module is the SDRAM controller. We use the off-chip SDRAM to store our program, however we need a specific controller the SDRAM cannot be interfaced directly to the bus. This is because it has a complicated row/column addressing scheme and needs to be constantly refreshed because of the nature of DRAM

sdr_pll	ALTPLL Intel FPGA IP	
indk_interface	Clock Input	Double-click to
indk_interface_reset	Reset Input	Double-click to
pll_slave	Avalon Memory Mapped Slave	Double-click to
c0	Clock Output	Double-click to
c1	Clock Output	sdr_clk

We need the PLL block for the clock in order to compensate for the clock being skewed from how the board is laid out, and propagation of signals throughout the traces. The SDRAM requires precise timing so we feed the system clock into the PLL and output a clock that is 3ns behind the controller clock to go into the SDRAM clock.

sysid_qsys_0	System ID Peripheral Intel FPGA IP
clk	Clock Input
reset	Reset Input
control_slave	Avalon Memory Mapped Slave

This module is a system ID checker- it ensures that the hardware is compatible with the software we are loading on and prevents it if there is any mismatch. For example, if we make any modification to the hardware without reprogramming the FPGA, this block will prevent us from uploading software onto possibly incompatible hardware

jtag_uart_0	JTAG UART Intel FPGA IP
clk	Clock Input
reset	Reset Input
avalon_jtag_slave	Avalon Memory Mapped Slave
irq	Interrupt Sender

This block allows us to use console commands such as printf and scanf. We need this so that we can both input the key/message, but also print out the encrypted/decrypted message for debugging purposes.

[-] AES_Decryption_Cor...	AES_Decryption_Core	
CLK	Clock Input	<i>Double-click</i>
RESET	Reset Input	<i>Double-click</i>
AES_Slave	Avalon Memory Mapped Slave	<i>Double-click</i>
Export_Data	Conduit	aes_export

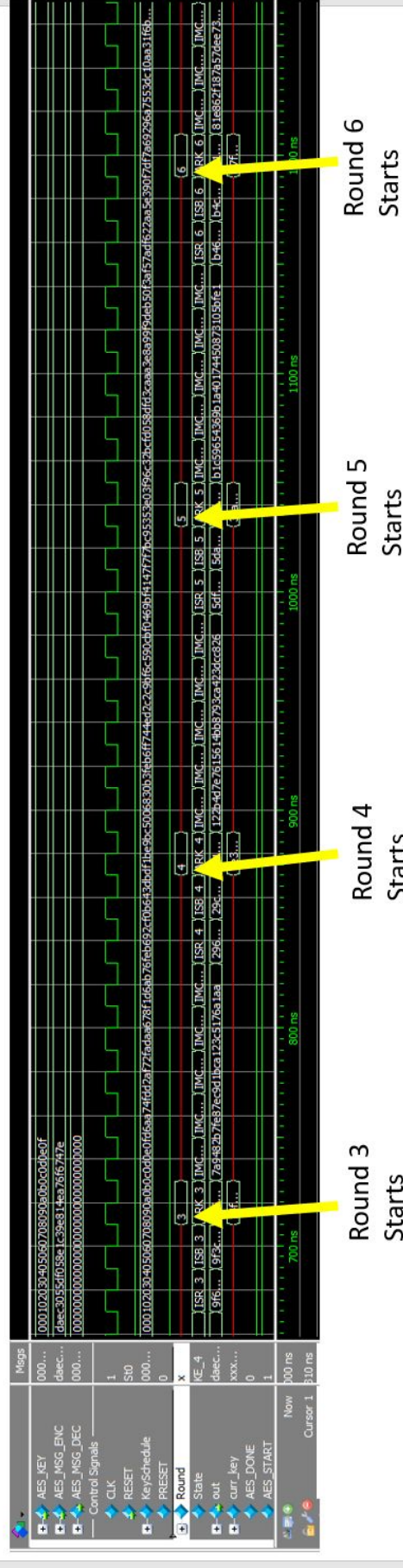
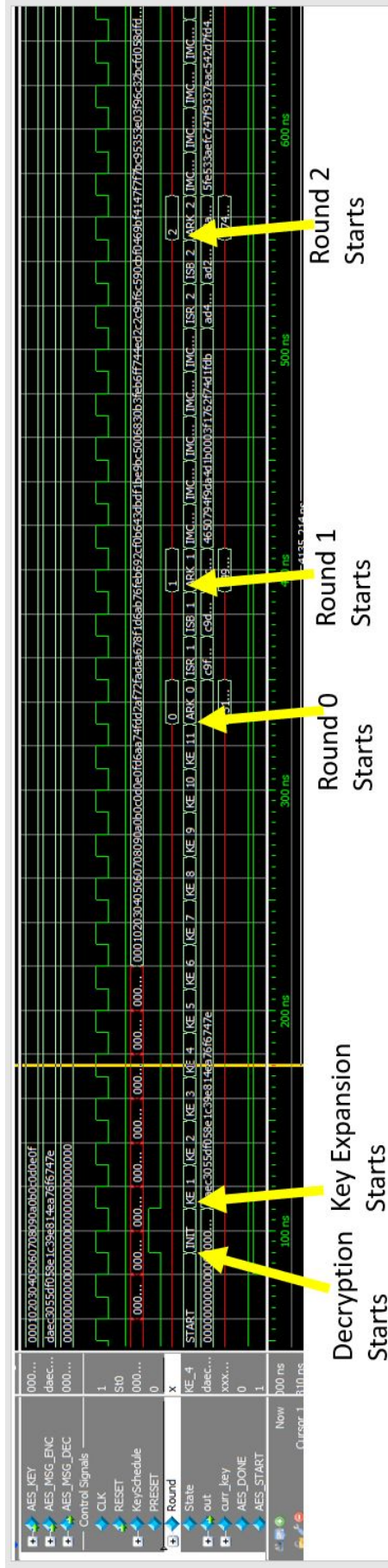
This block is the IP core that we designed in Platform Designer. This decryption core is the accelerator that we use to speed up decryption by processing it in hardware. It will take the Key and Message and return a decrypted message.

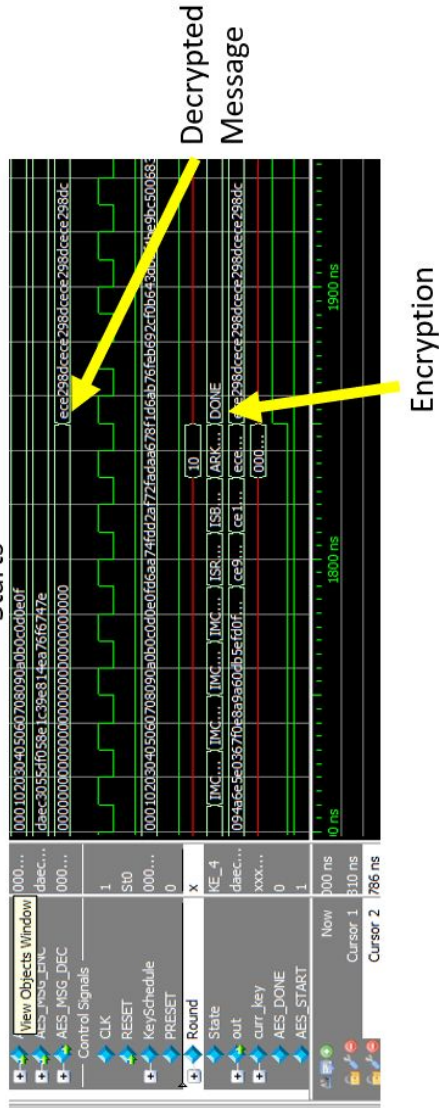
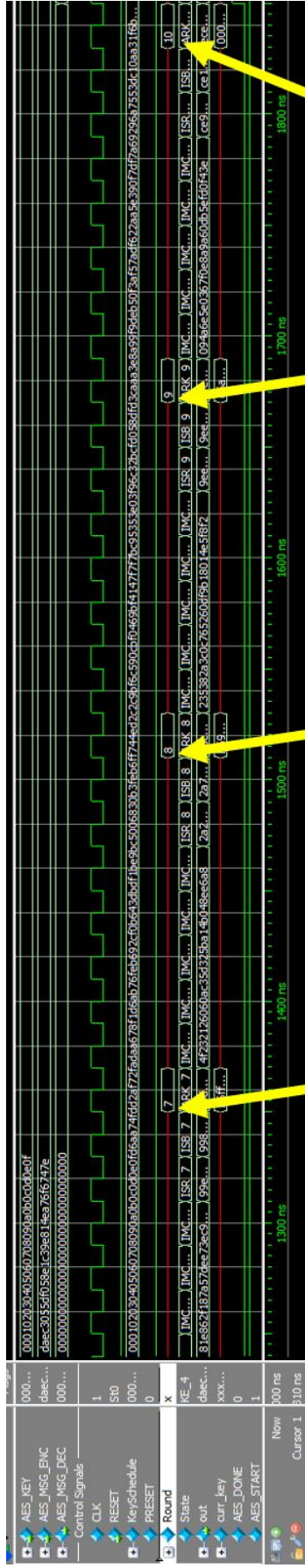
[-] TIMER	Interval Timer Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
irq	Interrupt Sender

This is an interval timer- we need this timer to send interrupts periodically because the NIOS II doesn't have a way to measure time and we want to be able to time how long each encryption/decryption took in our C program.

Annotated Simulations of AES Decryptor

```
5 //Signals
6 logic [127:0] AES_MSG_ENC;
7 logic [127:0] AES_KEY;
8 logic [127:0] AES_MSG_DEC;
9 logic AES_START, AES_DONE;
10 logic CLK, RESET;
11
12
13 AES aes_inst(.*);
14
15
16
17 // Clock initialization
18
19 always begin : CLOCK_GENERATION
20 #1 CLK = ~CLK;
21 end
22
23 initial begin: CLOCK_INITIALIZATION
24 CLK = 0;
25 end
26
27 initial begin: TEST_VECTORS
28
29 //Defaults
30 AES_START=0;
31
32 AES_MSG_ENC= 128'h439d619920ce415661019634f59fcf63;
33 AES_KEY = 128'h3b280014beaac269d613a16bfdc2be03;
34
35 #4 AES_START =1;
36 #4 AES_START =0;
37 //#4 AES_START =0;
38 #200;
39 AES_MSG_ENC= 128'hdaec3055df058e1c39e814ea76f6747e;
40 AES_KEY = 128'h000102030405060708090a0b0c0d0e0f;
41 #200;
42
43 end
44 endmodule
45
```



Post Lab Questions

Design Resources and Statistics Table

LUT	6082
DSP	0
BRAM	126,336
Flip-Flop	3402
Frequency	70.61 Mhz
Static Power	102.28 mW
Dynamic Power	1.39 mW
Total Power	174.71 mW

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)

We would expect the encryption/decryption to be faster on hardware than software. This is what our results showed

```
Select execution mode: 0 for testing, 1 for benchmarking
1
Software Encryption Speed: 0.442087 KB/s
Hardware Encryption Speed: 222.222222 KB/s
```

First of all, decryption is basically the same thing as encryption because it performs all the same steps with small modifications, so we wouldn't expect one to be at an advantage or disadvantage because of whether it was decryption or encrypting.

So we know that this is a fair comparison which is essentially deciding which is faster: hardware and software.

Consider that the NIOS II is actually synthesized on the FPGA- it is not a dedicated processor chip off the FPGA. Since the FPGA needs to be able to store all of the hardware we need, we are limited as to how big the processor can be. Because of space limitations, the NIOS II has to be relatively small which means that performance will take a hit.

Also, consider what the NIOS actually has to do: the NIOS has to fetch the instruction from the SDRAM which is off the chip. Then after that, it has to decode the instruction. Only until after that, then it can actually execute the instruction. Now consider hardware: we don't need to do any fetching, and decoding, etc.. All we have to do is wait for the combinational logic to finish propagating which is negligible compared to what the NIOS has to do. A simple XOR operation like AddRoundKeys in hardware is just an XOR gate while the NIOS has to fetch the instruction, decode the instruction to know that it is an XOR operation, then actually execute it.

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

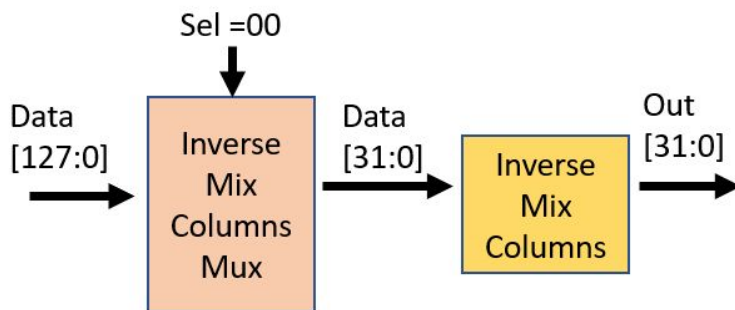
In this lab we are restricted to only have one InvMixColumns module, and this module operates on 32 bits at a time. This means that we have to operate serially on the first 32 bit of the 128bit current state, then on the next clock cycle operate on the next 32 bits, etc.

Instead of extending this operation to multiple states for each 32 bit portion, we would instantiate 4 of the InvMixColumns modules with each module processing a specific 32 bit section.

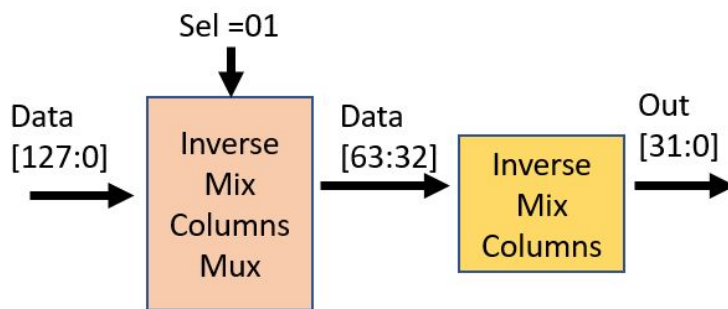
By operating in parallel we can reduce this operation to a single clock cycle, and since we do InvMixColumns 9 times, there will be very significant time savings.

Serial Operation (As in Lab)

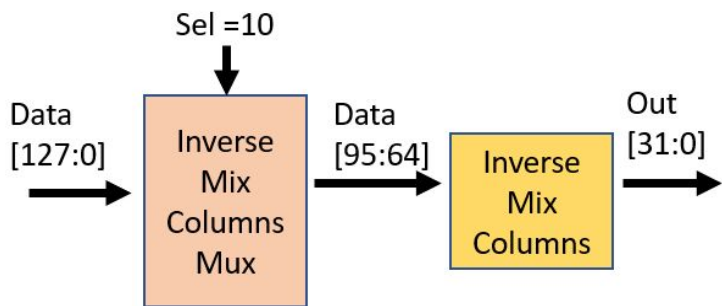
Cycle 1:



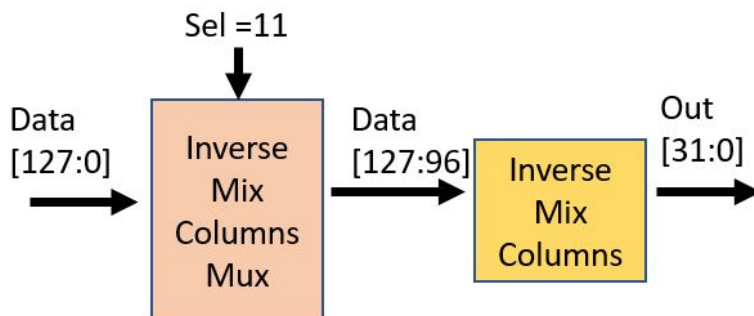
Cycle 2:



Cycle 3:

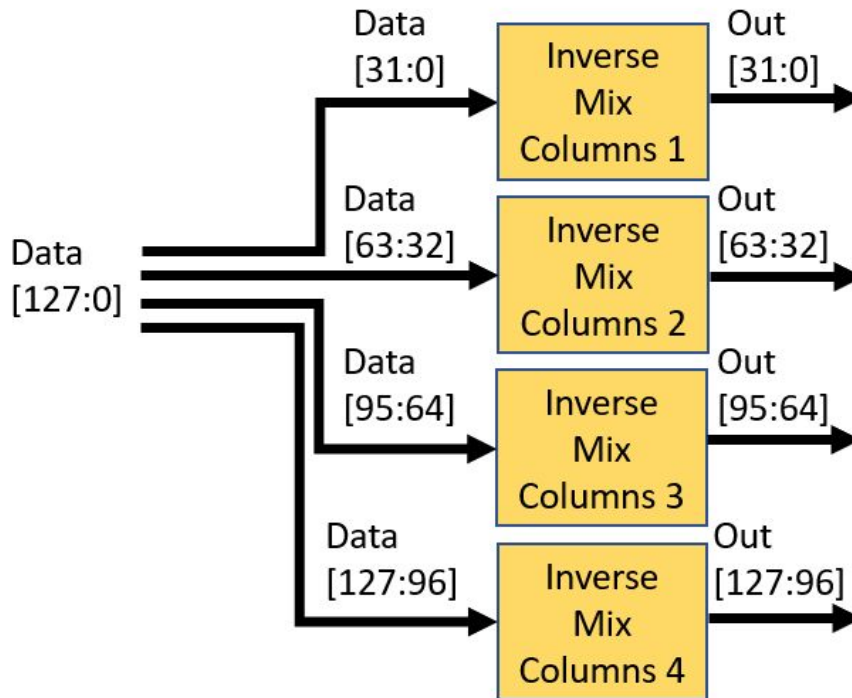


Cycle 4:



Parallel Operation (Proposed method)

Cycle 1:



Conclusion

Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it

Our design was fully functional. When the user entered a message and key, the software would successfully encrypt the message and the hardware would successfully decrypt the message. The hex displays would display the first 4 characters and the last 4 characters of the decrypted message. We also successfully added the timer module so that we can run benchmarking, and our results clearly showed what we should expect, that hardware was much faster than decryption.

Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right so it doesn't get changed.

There seems to be something wrong with the proposed fix to fix the "new_component" from "avalon_aes_interface" bug. I copied and pasted the code in the AES_Decryption_Core_hw.tcl file and when I opened platform designer it gave me a bunch of errors. Every TA we asked didn't know how to do this fix and told us to manually change the name every time after regenerating HDL. Also, there was barely any coverage of SignalTap in lecture and very few resources even though it seems to be very useful, and again, every TA we asked didn't know how to use this.