CUSTOM CRYPTOGRAPHIC MODULE

Final Report

Submission Date: 5/3/2018
Thursday 2:30-5:20 PM
GTA: Reena Elangovan

Prepared by:
Samuale Yigrem
Ryan Devlin
Dhairya Agrawal
Samanth Mottera

1. Executive Summary

Trusted Platform Module is an international standard created by the Trusted Computing Group for a secure cryptoprocessor, which is a dedicated microcontroller designed to secure hardware through integrated cryptographic keys. The U.S Department of Defense (DoD) specifies that "new computer assets (e.g., server, desktop, laptop, thin client, tablet, smartphone, personal digital assistant, mobile phone) procured to support DoD will include a TPM version 1.2 or higher where required by DISA STIGs and where such technology is available". One of the main pieces of the TPM module is the cryptographic core. This, in conjunction with a random number generator, compute all of the encryption and decryption done by the TPM.

In this paper, a solution for a Trusted Cryptographic Module [TCM] is presented. A TCM would be housed within a TPM to accelerate the cryptographic capabilities of the top level module. The TPM is becoming more and more relevant as time goes on due to the dynamic nature of the information industry. As more and more data is encrypted, the computational time spent calculating encryption algorithms increases. A dedicated piece of hardware (ASIC) can greatly increase the speed of the encryption, while allowing the CPU to perform other tasks. This allows for more data to be encrypted at a faster rate. On top of this, one of the major problems faced by encryption schemes is where and how the keys are stored. With a TPM/TCM combination, keys would be stored internally. This prevents keys from being intercepted in the computer when they are in use and allows for data to be much more secure when it is stored.

The proposed TCM would be capable of receiving data on a 128-bit AHB lite Bus, and encrypting the data with AES-128. The output of this module would be the ciphertext generated by the hardware, as well as an encrypted version of the key needed to decrypt the data. If the encrypt/decrypt line is switched to decrypt mode, the module would instead take in an encrypted key and ciphertext and output decrypted data. Normally, a TPM stores keys in internal memory to prevent them from leaving the module. Therefore, we have opted to store a master key in ROM inside the TCM. This masterkey will be a hardcoded value that will be randomly generated upon manufacturing of the device and stored internally. When data is to be encrypted, the RNG on SOC will gather entropy and then output a 128 bit random number to be used as a key for AES. This key will be sent to the TCM to encrypt data. After encryption and outputting the cipher text to external memory, this key will be fed into the AES engine and encrypted with the master key, and then output to be stored in a hashed password/decryption key table. This is done so that the hash table can be used to retrieve the encrypted key after authenticating the user during a decryption process. This way, the only way to decrypt the data would be to authenticate the user, and retrieve & feed the key back into this specific chip, as the master key inside the chip's ROM is the only thing that can be used to perform the decryption. Normally a TPM uses passwords in conjunction with a master key to authenticate the user, encrypt/decrypt files, and store hashed key tables. This involves more non-volatile memory, hash engines, HMAC engines, and other techniques that are well beyond the scope of what we are capable of completing in a semester. That is why this TCM solution is meant to be hooked into a larger TPM package that would perform the other tasks.

2 Design Specifications

2.1 System Usage

2.1.1 System Usage Diagram

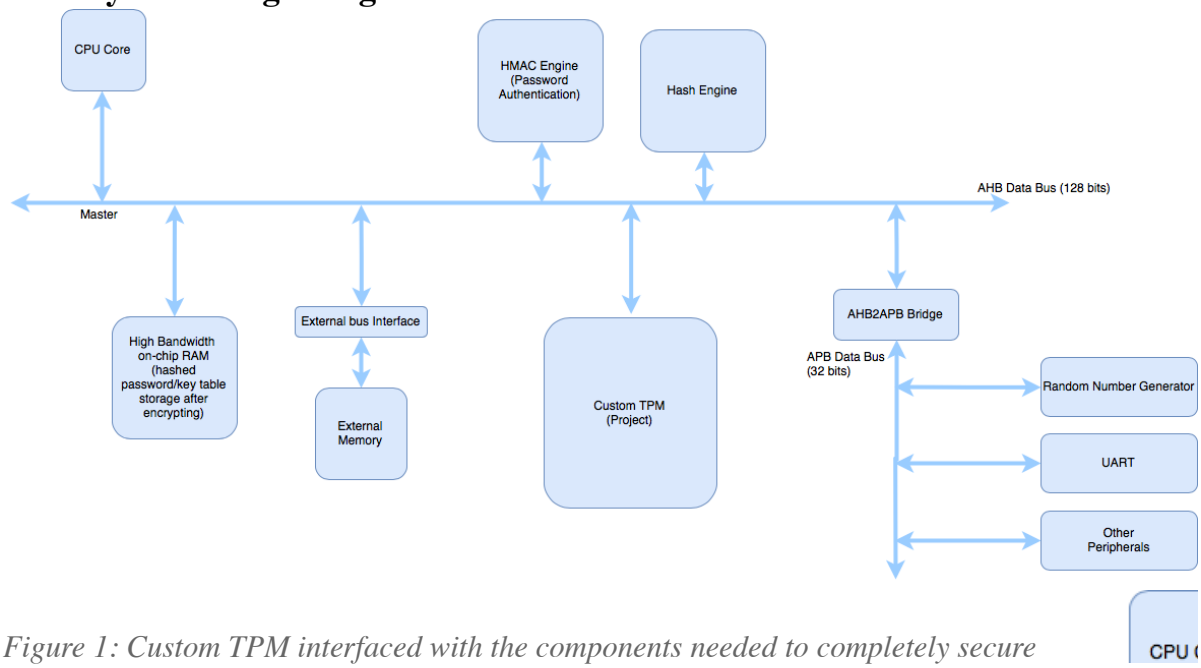


Figure 1: Custom TPM interfaced with the components needed to completely secure a system.

Figure 1 shows the TCM hooked up to several external devices that allow it to completely secure a system. The HMAC engine would be used in conjunction with the Hash engine to take in a password and authenticate a user. High Performance RAM would be used as a storage location for the hashed password/decryption key tables generated by the data output from the TCM. External memory may be used to receive files to encrypt or to store encrypted files. The CPU Core would help determine when to save to RAM, when to encrypt/decrypt, when to authenticate the user, etc.

2.1.2 Implemented Standards and Algorithms

- AES Encryption Algorithm
  - 128 bit encryption key for maximum possible security
  - 128 bit decryption key
  - 128 bits encode/decode data block size
  - 10 rounds of processing to encrypt/decrypt data
  - Rounds consists of steps such as Key Addition, Byte Substitution, Mix Columns and Shift Rows for encryption
  - Rounds consists of steps such as Key Addition, Inverse Byte Substitution, Inverse Mix Columns and Inverse Shift Rows for decryption
- AHB-Lite Data Bus
  - 128 bits data bandwidth
  - Can read the entire key or data in a single cycle
  - 67 MHz transfer rate

2.2 Design Pinout

Table 1: Miscellaneous pin-out

SIGNAL NAME	TYPE	NUM OF BITS	DESCRIPTION
clk	IN	1	Universal system clock
n_rst	IN	1	Active low asynchronous reset

Table 2: Custom TCM Interface Pins

SIGNAL NAME	TYPE	NUM OF BITS	DESCRIPTION
enable	IN	1	Active high module enable signal
data_done	IN	1	Signal asserted by CPU at the same time that last data packet to be encrypted/decrypted is sent
e_or_d	IN	1	Select line. 1 - encrypt. 0 - decrypt
irq_resp	IN	1	Active high read signal that tells TCM that our irq signal has been accepted and that data will start to be sent through the slave
HTRANS_SLAVE	IN	2	Signal that tells TCM the transfer mode of data from the master according to the AHB protocol
HWRITE_SLAVE	IN	1	Active high signal that tells TCM that the AHB master is going to write to the TCM's AHB slave
HSELx	IN	1	Active high signal that tells TCM that the device has been selected
HREADY_SLAVE	IN	1	Active high signal that tells the TCM that the data bus is ready to take in data
HWDATA_SLAVE	IN	128	Plain/Cipher text sent in 128 bit packets to the AHB slave
HWRITE_MASTER	OUT	1	Active high signal indicating that data is being sent from the master to the SRAM
HBURST_MASTER	OUT	3	The burst type indicates if the transfer is a single transfer or forms part of a burst
HTRANS_MASTER	OUT	2	The type of transfer in the AHB protocol

			implementation
HADDR	OUT	32	Contains address of where in SRAM to write plain/cipher text to
HWDATA_MASTER	OUT	128	Plain/Cipher text sent in 128 bit packets to the SRAM
irq	OUT	1	Active high interrupt to signal master to initiate a read request to the TCM
ready	OUT	1	Active signal that lets the master know the TCM is ready to start a new operation

2.3 Operational Characteristics

2.3.1 AES Module Usage & Flow of Operations

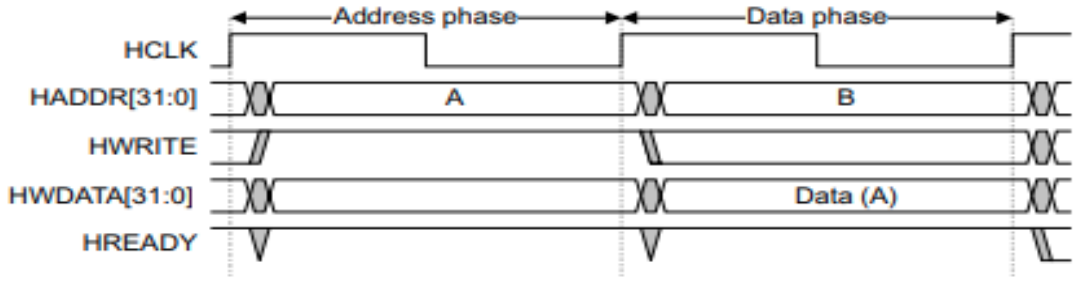
Normal TPM Usage:

1. Data received through AMBA AHB Protocol 128-bit bus.
2. 128-bit bus transfers data to 128-bit register in the RCU using AHB Slave unit.
3. Cryptographic Control Unit (CCU) will signal that fifo register is full and will tell AES core which part of the algorithm to run based on whether the encrypt/decrypt line is selected.
4. All 128 bits for the key are passed to the AES core.
5. The machine waits to expand that key and when done, sends an interrupt signal output to the cpu to start sending the data.
6. The cpu responds and starts sending data with an address data packet to indicate where to write in the sram and then sends data every clock cycle until out of data.
7. If Encrypting:
    a. Once the RCU determines the data is fully loaded, it tells the AES Core to start processing the data.
    b. The core encrypts as instructed and passes each encrypted block out.
    c. Each encrypted 128-bit block is passed out of the TCM to an SRAM and an AMBA AHB Protocol Master transmits the blocks out in one clock cycle (burst mode).
    d. Once the CCU determines all of the data has been fully encrypted, the key used is fed through the AES core and encrypted with a TCM-unique Master Private Key (MPK) stored in ROM. (This makes it so that this ASIC is the only entity in the world capable of decrypting the data. The MPK is assumed to have been cryptographically randomly generated at the time of ASIC creation. In reality this would most likely be handled with an RNG and EEPROM, but that is outside the scope of this project).
    e. The key is then similarly output on the data bus with the AMBA AHB Protocol to be saved on the SoC.
8. If Decrypting:
    a. When we first receive the key, we use our expanded master key to decrypt this auxiliary key so we can use it to decrypt data. Once its decrypted, we expand this decrypted auxiliary key and then send the interrupt.
    b. Once the RCU determines the data is fully loaded, it tells the AES Core to start processing the data.
    c. The core encrypts as instructed and passes each decrypted block out.
    d. Each decrypted 128-bit block is passed out of the TCM to an SRAM and an AMBA AHB Protocol Master transmits the blocks out in one clock cycle (burst mode).
    e. After the CCU receives a signal that no more data will be sent, it decrypts the last block, and sends it out via the AHB Slave.
    f. The key is discarded as it is no longer needed.

2.3.2 AHB Lite Protocol

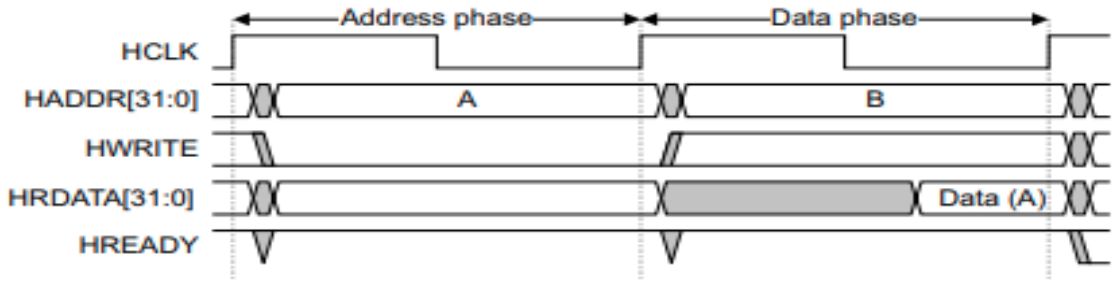
Width of data bus: 128 bits

Write Transfers



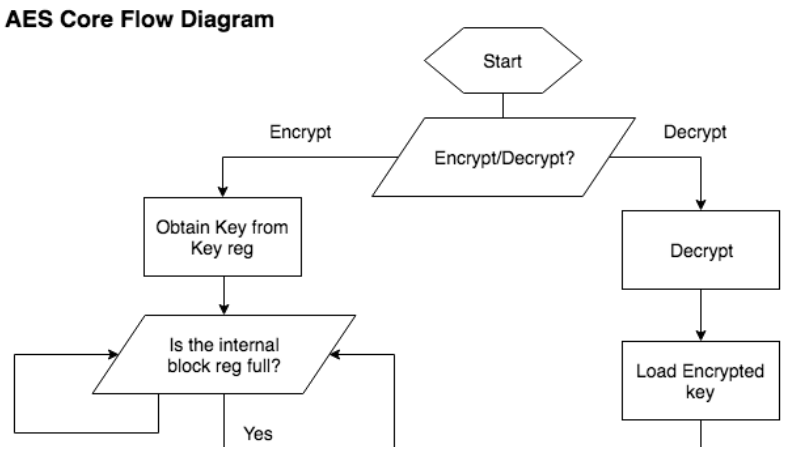
The basic transfer of type NONSEQUENTIAL will be used during write transfers. There will be one clock cycle of address phase during which the HADDR(valid), HWDATA(valid), HWRITE(high), HREADY(high), and HSELx(high) will indicate a valid write transfer. At the next clock edge, our AHB Slave will read all 128 bits present on the HWDATA line.

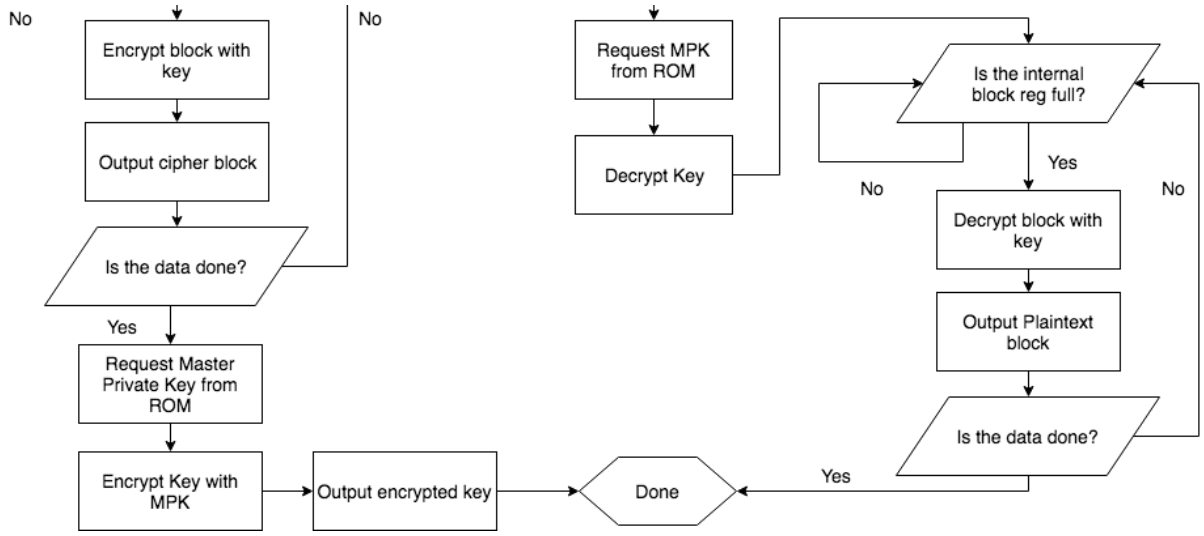
Read Transfers



The basic transfer type of NONSEQUENTIAL will be used during read transfers. Once again, there will be one clock cycle of address phase during which HADDR(valid), HWRITE(high), HREADY(high), and HSELx(high) will indicate a valid read request to our AHB Slave module. Then, our AHB Slave will put the encrypted/decrypted data on the HRDATA line and all 128 bits will be read by the AHB Master.

2.3.3 AES Encryption/Decryption Algorithm





The above flow diagram is included to aid in visualizing the sequence of operations of our custom AES encryption module in detail. This is meant to give an in-depth look at an algorithm that was introduced in section 3.1 AES Module Usage.

2.4 Requirements for Design

The intention of this proposal is to provide plans to build a TCM that is able to quickly and efficiently encrypt data using AES-128. The primary optimization focus for this design will be speed. Encryption involves a fairly large amount of processing power and takes an exceptionally long time for large files if not implemented properly. One of the main goals will be to improve the speed at which the ASIC can encrypt raw data. In doing so, the incentive to use the ASIC dramatically goes up as it becomes lightening fast as compared to doing the encryption via software. Based off of this current design, clock speed and internal logic will be the most critical parameters. Finding the right clock speed for the most optimal version of the design is something that we need to improve upon as we build the device because this value will depend on the critical path within the AES core. Because AES can only encrypt 128 bits at a time, using the 128-bit AHB protocol as well as using 128-bit parallel internal transfer lines allow for maximum data transfer through the device. As for the algorithm itself, there will be many optimizations used to improve its processing speed. The main way this will be done is by processing the S-boxes of the algorithm in parallel. Also, the implementation of the algorithm we have designed is composed of almost entirely combinational logic. This allows for output to be calculated as fast as physically allowed by the speed of the transistors.

Another factor to consider is the fact that the data throughput is variable as many things can affect it. Besides clock speed, which is the most obvious parameter that could improve throughput, there are other factors such as the maximum clock speed at which ROM is accessed, as well as the maximum speed the registers can switch. Also, how fast the Cryptographic Controller Unit is capable of recognizing a change in state and responding to that change could place a limit on speed. Sending data too fast could result in data corruption, garbage generation, or possible entrance into an unknown “locked in” state. This will be combated by optimizing the AES algorithm and internal state machines as much as possible and sending data every clock cycle and outputting encrypted data every clock cycle using a pipelined approach.

3 Design Implementation

3.1 Design Architecture

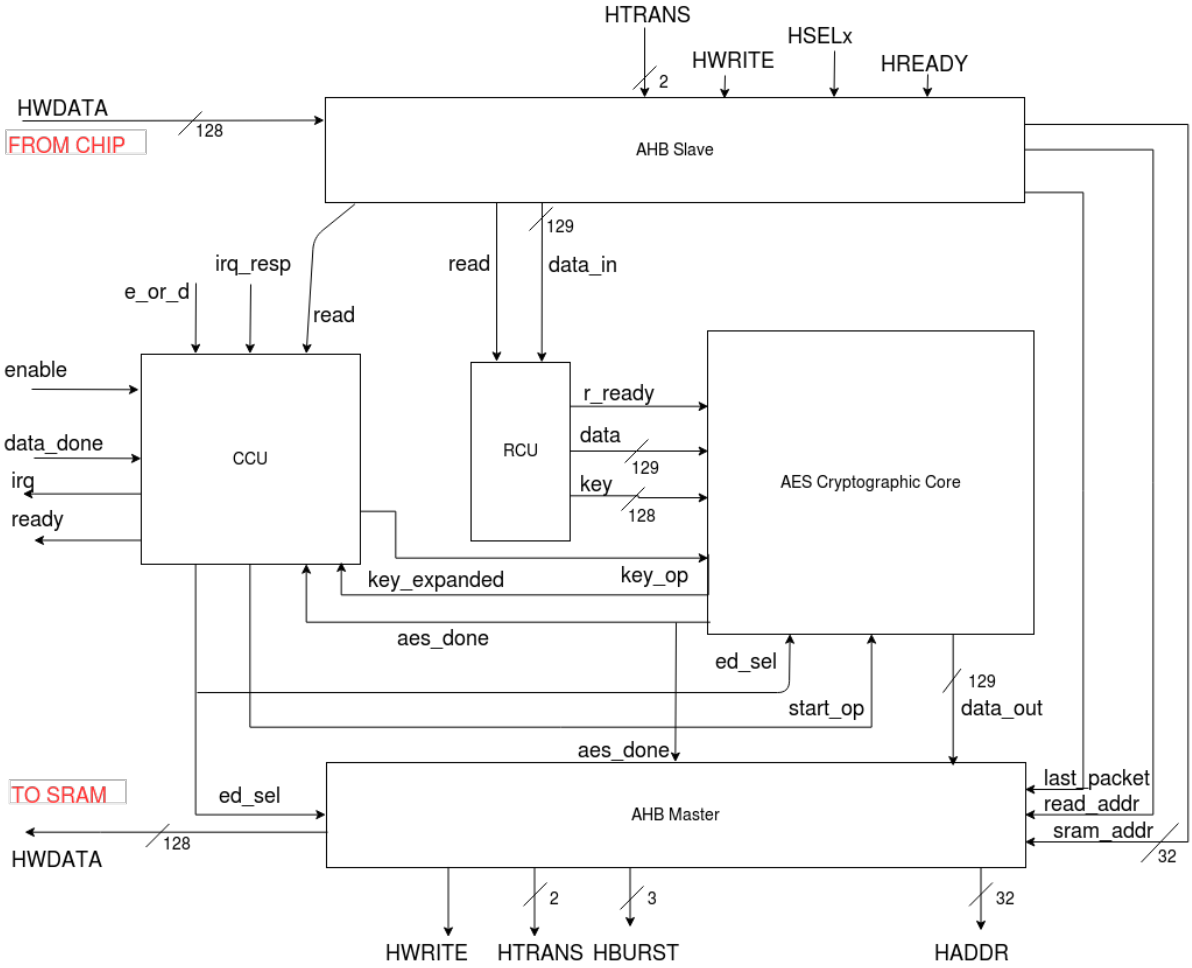
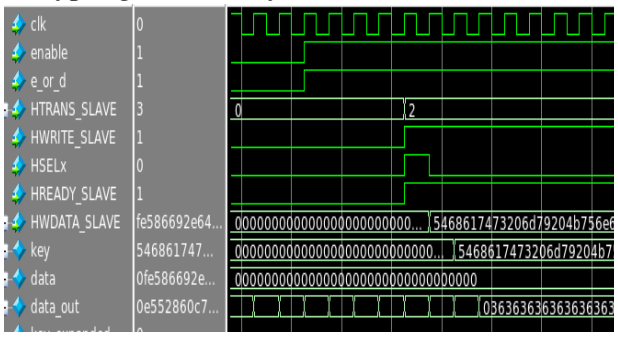


Figure 2a. Architectural Design for the Customized TCM.

Figure 2 shows a more in depth look at just how the Customized TCM would work. The AMBA AHB-Lite Data Bus would be used to send data to and from the TCM. When encrypting, a key is first sent over the bus to be stored internally. Next, data is sent in packets of 128 bit blocks to be encrypted one block at a time. The data is encrypted with the key, and then the key is encrypted with a master key. The last data block and the encrypted key are the lasts things passed out of the module to an SRAM through the master unit. When decrypting, the encrypted key is passed in first, followed by blocks of data. Once the key is decrypted with the master key, each block is decrypted with the previously decrypted key. The data is passed out the same way as when encrypting and the key is discarded.



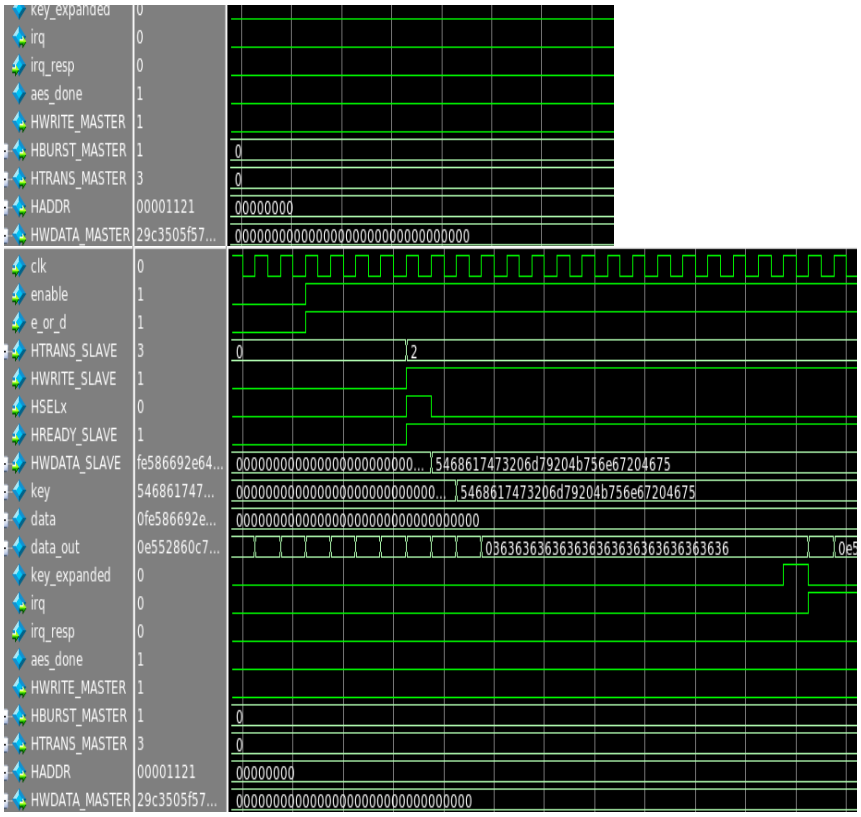


Figure 2b. Timing Diagram for the Customized TCM Top Level.

The timing diagram above shows how the TCM normally operates (in this case it is encrypting, but decrypting behaves roughly the same way). One thing to notice is that these are not all of the signals as seen in the pinout/architectural diagram, but these are the most relevant signals for a normal block operation using the TCM. First, the enable signal is raised high to initialize the TCM into a state that will perform the operation requested. Next, the e_or_d line is set to tell the TCM if the SOC is attempting to encrypt or decrypt the data. After some time (it does not matter how long), the master will initiate an AHB bus write request to the AHB slave unit. If the HREADY, HSELx and HWRITE signals go high at the same time, an AHB write operation will be performed. The first step is to initiate a write via AHB with data presented on the 128 HWDATA lines. The HSELx, HREADY, and HWRITE_SLAVE lines are all set during what is called the “address phase” one clock cycle before the “data phase” in which the values on the HWDATA_SLAVE lines are processed. The HWDATA_SLAVE lines are valid for one clock cycle after the HSELx line goes high and it is on this clock edge the TCM intakes all 128 bits of key. Now that the 128-bit data block has been loaded into the TCM, it expands the key and issues an interrupt request to receive data onece done. It receives the data in the same way as before. There is roughly 12 clock cycles where the TCM encrypts that data block. Once the encryption is finished and the TCM is ready to offload the data. It sends the data to the AHB Master unit which writes it out to an SRAM by deasserting HWRITE_MASTER and changing HTARNS from idle transfer to signle transfer to start writing data. The data is put on the 128 HWDATA_MASTER lines to be read out of the TCM in the next cycle. The two red lines show the packet coming in and going out.

Name of Block	Category	Gate/ FF Count	Area (um2)	Comments
AES Cryptographic Core			8,028,450	From AES Core functional block
CCU			49,500	From CCU RTL Area
RCU			633,600	From RCU RTL Area

AHB Slave Unit			36,600	From AHB Slave RTL Area
AHB Master Unit			36,600	From AHB Master RTL Area
Total Core Area			8,778,150	
Chip Area Calculations (units in um or um2)				
Number of I/O Pads:	16			
I/O Pad Dimensions:	90	by	300	
I/O Based Padframe Dimensions:	960	by	960	
Core Dimensions	2,958	by	2,958	
Core Based Padframe Dimensions:	3,858	by	3,858	
Final Padframe Dimensions:	3,858	by	3,858	
Final Chip Area:	14,882,059			

Description: The rationale behind the total area of the architectural block diagram is to sum all the areas from the sub functional blocks and RTL diagrams. Further explanations have been given in the sub functional blocks' sections and RTL diagrams' descriptions.

3.2 Functional Block Diagrams

3.2.1 Cryptographic Control Unit

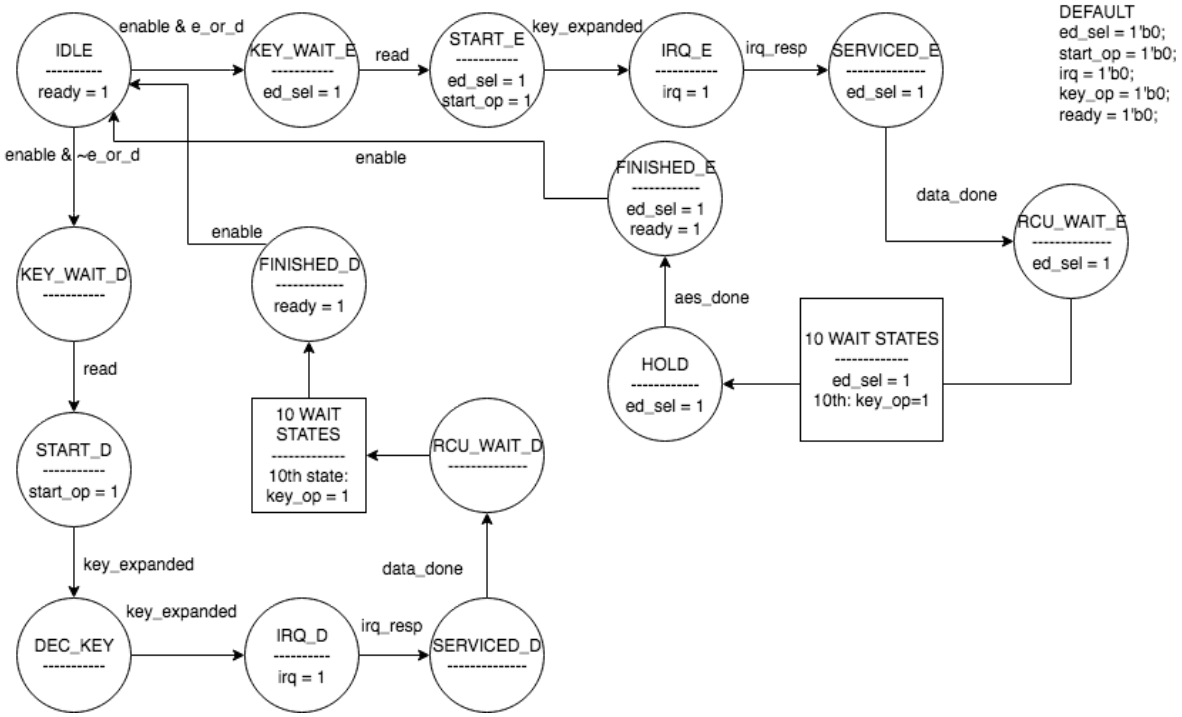


Figure 2c. Cryptographic Control Unit State Transition Diagram

Figure 2c shows the state transition diagram of the CCU. The CCU is used to monitor the internal states of the TCM and tell each block what it is supposed to do. It is ready in IDLE state to start a new operation. Then uses signal lines connected to an external controller to determine what it is being commanded to do. When the TCM is enabled, it checks the state of the D/E, the

CCU will then either decrypt the key or start encrypting the block. The CCU will wait for the key to be read in. If encrypting, the CCU will wait for key to expand and then wait to receive the first data packet. Initially it will wait 10 clock cycles before the first packet is encrypted. After that, every clock cycle an encrypted packet comes out. After it is done with data, it waits for the auxiliary key to be encrypted with the master key and then goes to IDLE. If decrypting, the process is similar but instead of doing key at the end, we decrypt the supplied key using the master key first. Then it decrypts all the data packets until finished with packets and then goes back to IDLE. aes_done tells the AHB Master controller that data is ready to be transmitted out of the TCM via the AHB bus.

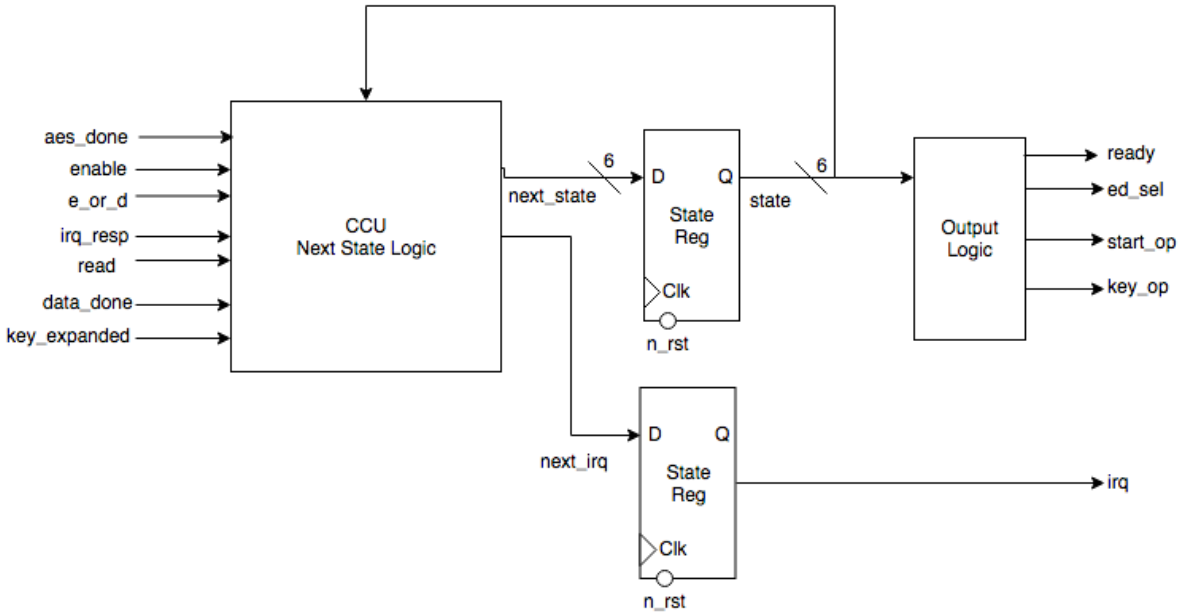


Figure 2d. Cryptographic Control Unit RTL Diagram

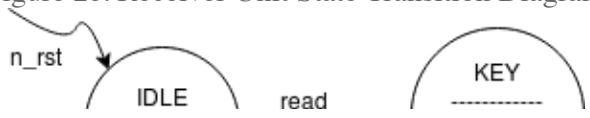
Figure 2d shows the RTL diagram for the CCU. Since the CCU is only a finite state machine, the diagram is very simple and timing diagram won't really explain anything. There is a state register, a next state logic block and output logic block. Irq is registered so that the interrupt request is glitch free and the cpu doesn't get interrupted randomly.

Core Area Calculations				
Name of Block	**Category**	**Gate/ FF Count**	**Area (um2)**	**Comments**
CCU State Register	Reg. w/ Reset	5	12,000	18 states needed. 2^5 = 32 states
CCU Next State Logic	Combinational	32	24,000	32*500*1.5
CCU Output Logic	Combinational	18	13,500	18*500*1.5
CCU Total		55	49,500	Total Area

Description: The way that the area of the State Register is computed using the number of states we have (5 flip flops): 2^5 = 32 states available of which 18 we use. Then the area of the Next State Logic is determined by counting the number of states plus the number of times a state changes based on an input to the Next State Logic block. The Output Logic was then determined by counting the states because the output only depends on the current state (Moore model).

3.2.2 Receiver Control Unit

Figure 2e. Receiver Unit State Transition Diagram



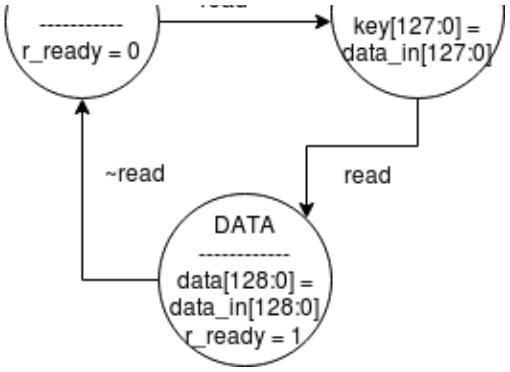


Figure 2e shows the design flow for our Receiver Unit working with the AMBA AHB-Lite protocol. The Receiver Unit is responsible for taking in a key as well as data/cipher data and store them in their respective places in the data and key registers. Internally, this is done by waiting for a read signal from AHB-Slave to start reading of the data. It first reads the key. Then it reads the data. It continues to just read the data until we reset it when starting to process a new file. Key and data outputs are registered so in the state the values are shown to be assigned, they are clocked in.

Figure 2f. Receiver Controller Unit RTL

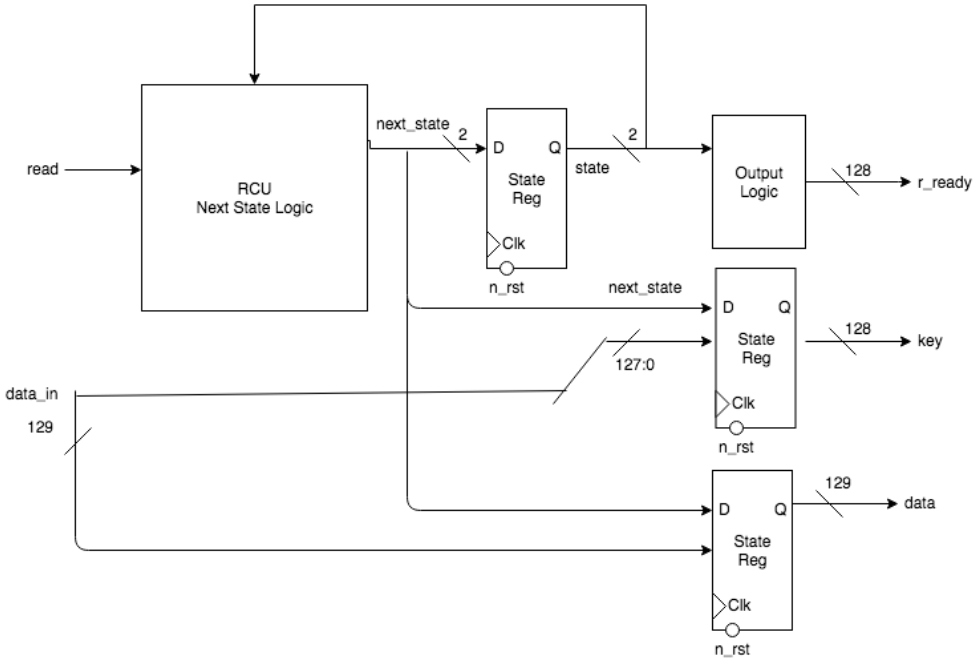


Figure 2f shows the RTL diagram for the Receiver control unit. The diagram is just a state machine so it has a state register, next state logic, and an output logic block. It also has two output regs for the key and data so that we can hold the values for the encryption/decryption logic to process them. Since this is a highly simple state machine, a timing diagram won't give any new details that the diagram already doesn't give.

Core Area Calculations				
Name of Block	**Category**	**Gate/ FF Count**	**Area (um2)**	**Comments**
RCU State Register	Reg. w/ Reset	3	4,800	3 states needed. 2^2 = 4 states
RCU Next State Logic	Combinational	10	7,500	10*500*1.5
RCU Output Logic	Combinational	6	4,500	6*500*1.5
	Reg. w/			

RCU Key Register	Reset	128	307,200	128 bits for key
RCU Data Register	Reg. w/ Reset	128	307,200	128 bits for data
RCU Total		275	633,600	Total Area

Description: The way that the area of the State Register is computed using the number of states we have (2 flip flops): 2^2 = 4 states available of which 3 we use. Then the area of the Next State Logic is determined by counting the number of states plus the number of times a state changes based on an input to the Next State Logic block. The Output Logic was then determined by counting the states because the output only depends on the current state (Moore model). Then we have two 128 bit registers with a reset for the key and data.

3.2.3 AES Core

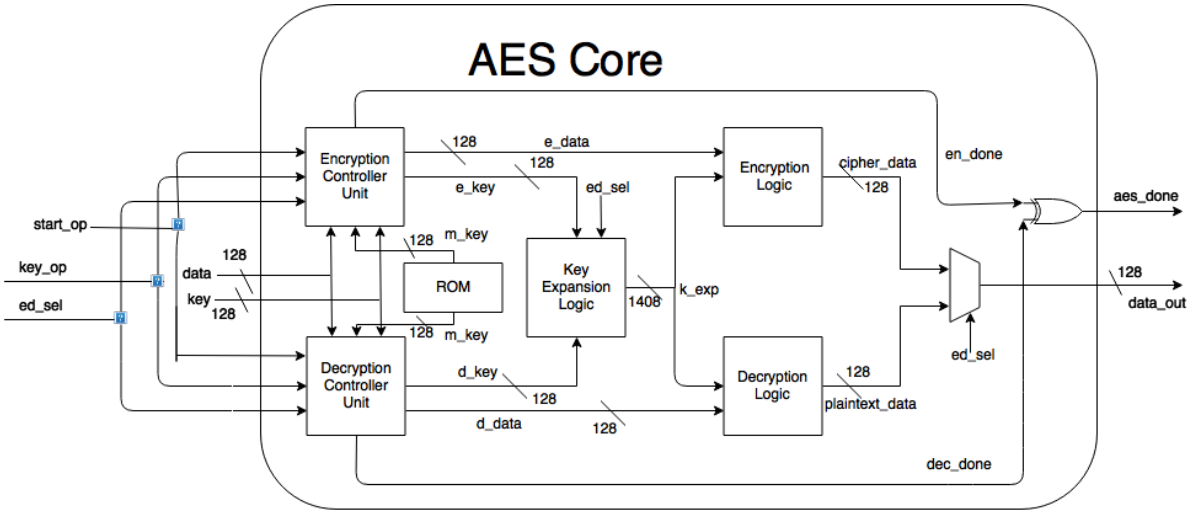
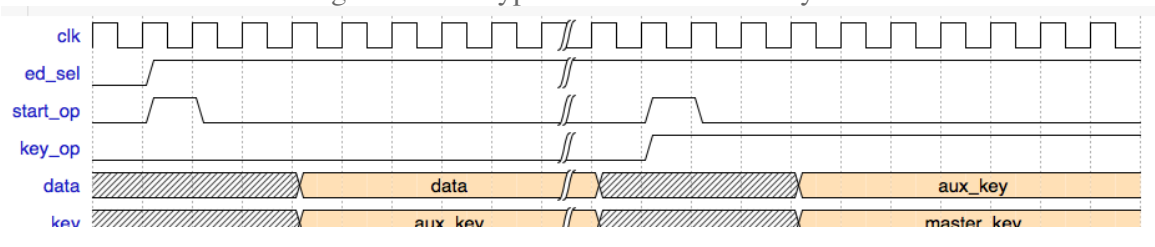


Figure 2g. AES Core

Figure 2g shows a block diagram for the AES Core. The core is responsible for processing all the encryption and decryption for the TCM. It consists of 6 internal blocks which handle each step of the AES process. The most simple block is the ROM which hold the AES Master Private Key even while power is turned off. The Encryption and Decryption Controller Units are two state machines that govern the behavior of the algorithm as the data is processed. For instance, if the core is encrypting, the ECU will wait for a "start_op" pulse before it starts encrypting. If it has just finished encrypting the last block, it is responsible for routing the key back into the encryption logic to be encrypted with the MPK. The Key expansion logic is combinational logic that calculates an expansion of the key used in each of the ten steps of the AES algorithm. The Encryption and Decryption Logic blocks are combinational logic that perform the ten steps needed to encrypt/decrypt the data. By using almost entirely combinational logic for the encryption/decryption process, the TCM is able to rapidly compute the AES algorithm. Finally, the aes_done signal is used to signal the other blocks in the TCM that the data presented on the 128 data_out lines is valid AES data that now needs to be sent out via AHB.
The key details of AES encryption works is very general and can be looked up. What's special about our implementation is that top optimize encryption speed we used a lot of LUTs for key expansion and encryption and decryption. Only simpler mathematical calculations were implemented using combinational gates and that too mostly using XOR and AND gates.

Figure 2h. Encryption Controller Unit Cycle



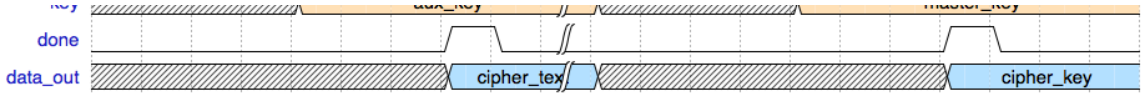


Figure 2h shows a timing diagram for the AES encryption. First we see a start_op and low key_op signal along with a high ed_sel line. This tells the ECU to start encrypting the data we have with the auxiliary key (key for the current round of AES). The done flag will be pulsed every time the 128 bit data in the data register has been encrypted *(for us it's every cycle now so done stays high until we are done with data packets)*. This will happen in a loop until we finish all data packets. Then, the ECU routes the auxiliary key onto our data register and the master key from ROM onto the key register to encrypt the auxiliary key with the master key: the TCM knows to do this when key_op goes high. After this, the ECU block is finished and goes back to IDLE. ECU controls the encryption logic block to perform encryption operations.

Figure 2i. Decryption Controller Unit Cycle

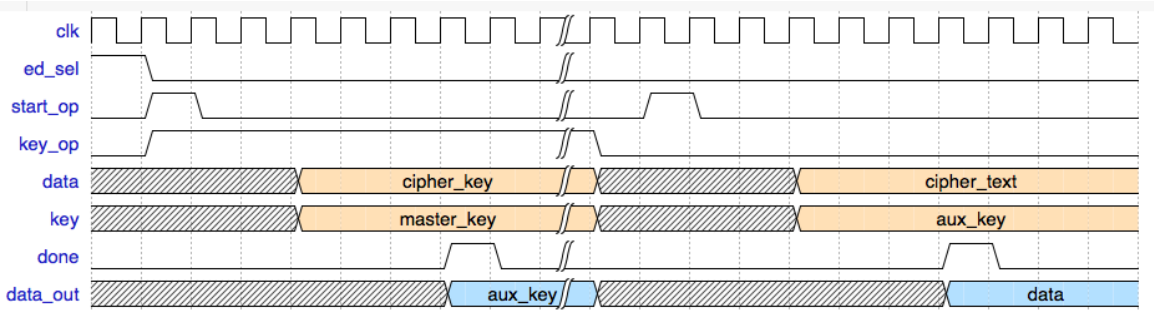


Figure 2i shows a timing diagram for the AES decryption. First we see a start_op and key_op along with a low ed_sel line. This tells the DCU to start decrypting the key with the master key from our ROM. This will get us the auxiliary key (which is what was used to encrypt the data to begin with) and then every cycle after that, we get a a new 128 bit data packet to decrypt using the auxiliary key and get the plain text. Also, the done flag is pulsed every time a decryption cycle is finished *(for us it's every cycle now so done stays high until we are done with data packets)*. After this, the DCU block is finished and goes back to IDLE. DCU controls the decryption logic block to perform decryption operations.

Core Area Calculations

Name of Block	Category	Gate/ FF Count	Area (um2)	Comments
AES ECU functional block			635,100	From ECU RTL Area estimation
AES DCU functional block			635,850	From DCU RTL Area estimation
AES Key Expansion functional block	Combinational	50	37,500	50*500*1.5
AES Encryption Logic	Combinational	4480	3,360,000	128 bits for key
AES Decryption Logic	Combinational	4480	3,360,000	128 bits for data
AES Core Total		9010	8,028,450	Total Area

Description: The area of the AES Cryptographic Unit is the sum of all the functional blocks within it. The ECU and DCU functional blocks have an area design explanation below. The AES Key Expansion, Encryption Logic, and the Decryption Logic blocks (all combinational) are all estimated based on the description of the AES algorithm and how many gates it would take to achieve the required function of this algorithm. (This preliminary estimation is going to be very different from our actual area because we were originally going to perform all the mathematical operations and in our actual design we ended up using a lot of lookup tables which blew up our actual area. We even ended up adding registers in between to fix violations)

actual area. We even ended up adding registers in between to fix violations).

3.2.3.1 Encryption Controller Unit

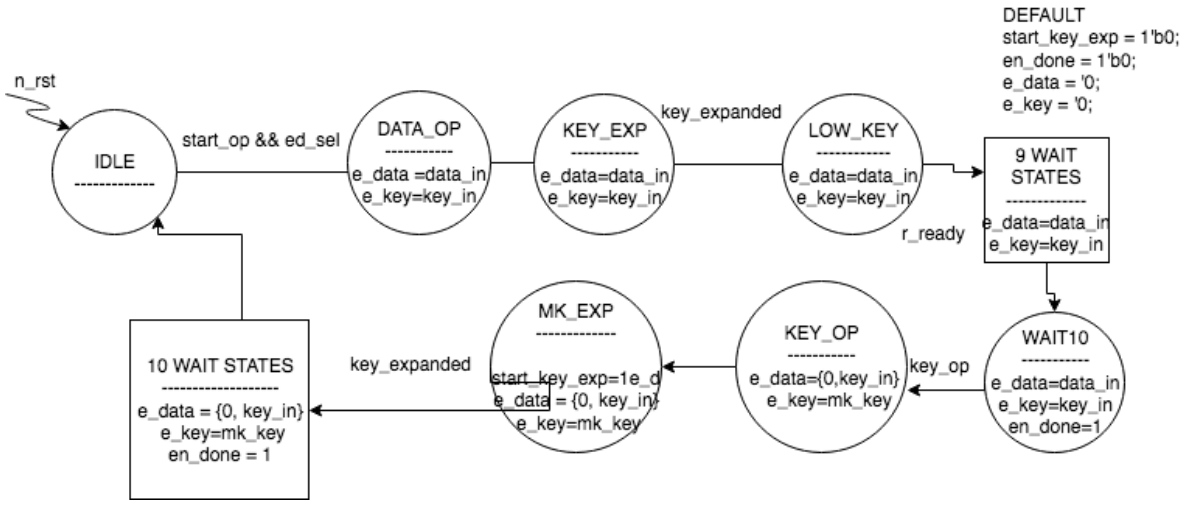


Figure 2j. Encryption Control Unit State Transition Diagram

Figure 2j shows the design flow for the Encryption Control Unit of our TCM. This handles the case when we want to encrypt data with a provided random key. It uses the key and the data input from the receiver control unit. First it waits to be enabled. Then it gets the key and waits for it to be expanded. After expanding the key, it waits to receive data to encrypt. After 10 clock cycles (shown as wait states) the data has finished processing and en_done goes high to signal data is read to be read out. Then, it checks if the key_op line is asserted to check if its the end of data. If not, the core keeps encrypting the data packets (through combinational logic blocks) using the key provided. Then, it goes back to idle in the next cycle. If it is the end of data, the core encrypts the encryption key using the master key stored on the TCM and asserts an encryption done flag (en_done) when finished. Then, it goes back to idle in the next cycle. The waveform diagram for an encryption operation is shown in the parent section 3.2.3.

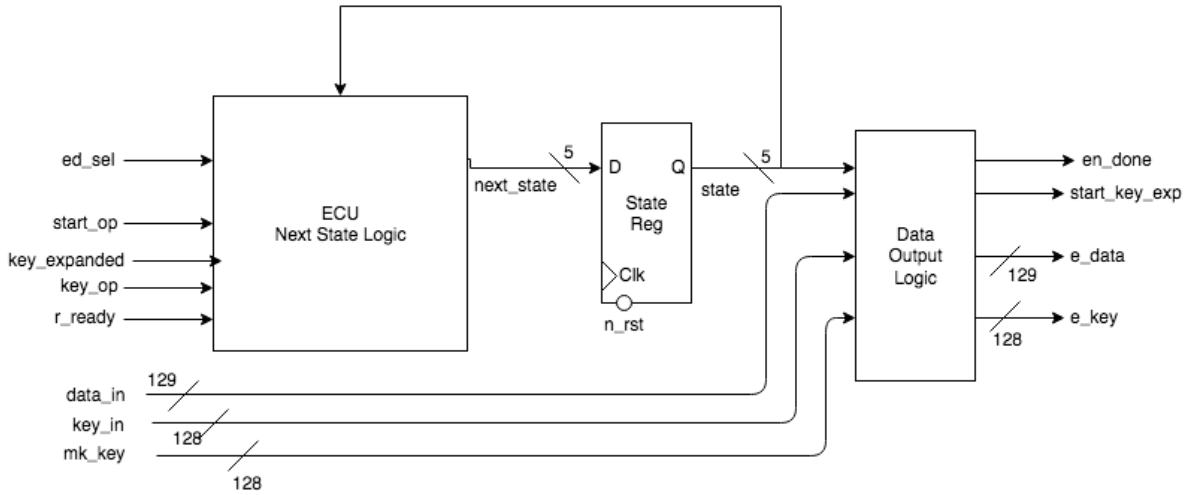


Figure 2k. Encryption Control Unit RTL

Figure 2k shows the RTL diagram for the ECU. This again is just a simple state machine so it has a state register, a next state logic block and an output logic block. We have a couple key and data lines going to output logic block which are just directed to separate output lines e_key and e_data to be used by combinational logic.

Core Area Calculations

Name of Block	Category	Gate/ FF Count	Area (um2)	Comments
ECU State Register	Reg. w/ Reset	3	12.000	26 states needed, 2^5 = 32 states

ECU State Register	Reset	5	12,000	26 states needed: 2^5 = 32 states
ECU Next State Logic	Combinational	10	7,500	10*500*1.5
ECU Output Logic	Combinational	8	6,000	8*500*1.5
ECU Key Register	Reg. w/ Reset	128	307,200	128 bits for key
ECU Data Register	Reg. w/ Reset	128	307,200	128 bits for data
ECU Total		277	635,100	Total Area

Description: The way that the area of the State Register is computed using the number of states we have (5 flip flops): 2^5 = 32 states available of which 26 we use. Then the area of the Next State Logic is determined by counting the number of states plus the number of times a state changes based on an input to the Next State Logic block. The Output Logic was then determined by counting the states because the output only depends on the current state (Moore model). Then we originally had two 128 bit registers with a reset for the key and data (which we got rid of).

3.2.3.2 Decryption Controller Unit

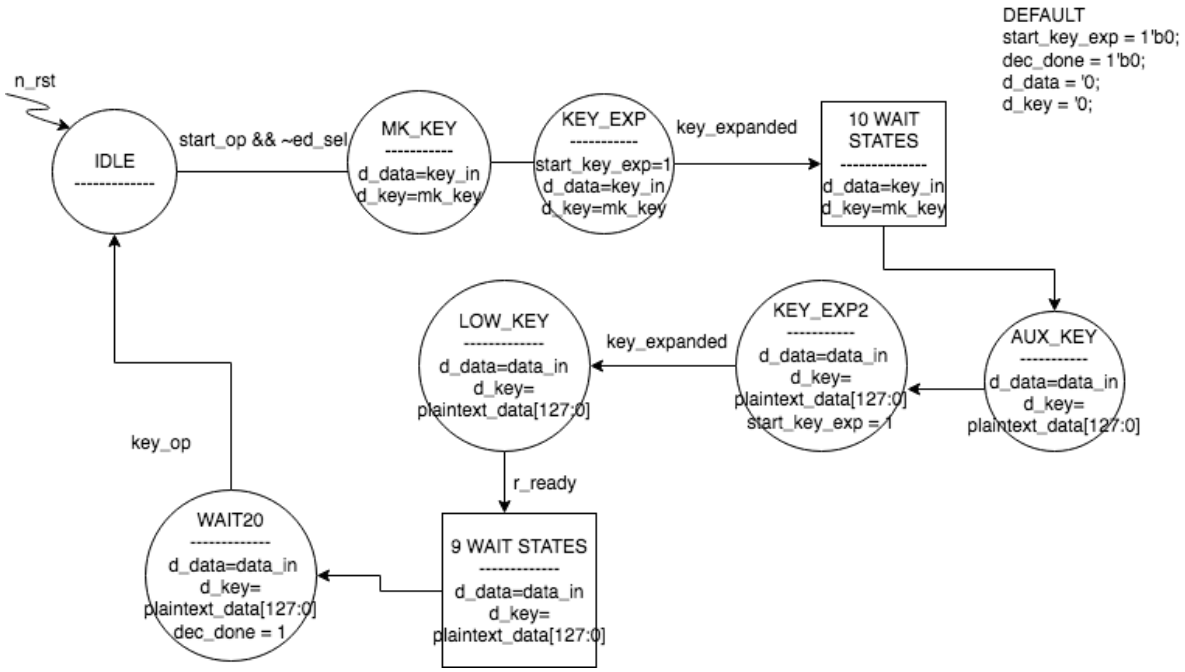


Figure 2l. Decryption Control Unit State Transition Diagram

Figure 2l shows the design flow for the Decryption Control Unit of our TCM. This handles the case when we want to decrypt data with a key. It uses the key and the data input from the receiver control unit. First it waits to be enabled, then it decrypts the key from our Receiver Control Unit using the master key on the TCM. The first key expanded is for master key expanding, that then decrypts out auxiliary key which is then expanded. At this point, our state machine will wait until it is called upon again with an r_ready signal. When its called the next time, the encrypted ciphertext is decrypted using the decrypted key. It asserts a decryption done

flag (dec_done) to signal that it has finished decrypting and then goes to idle when does with packets. Data is decrypted in 128-bit blocks until all the data for a file is finished. The waveform diagram for a decryption operation is shown in the parent section 3.2.3.

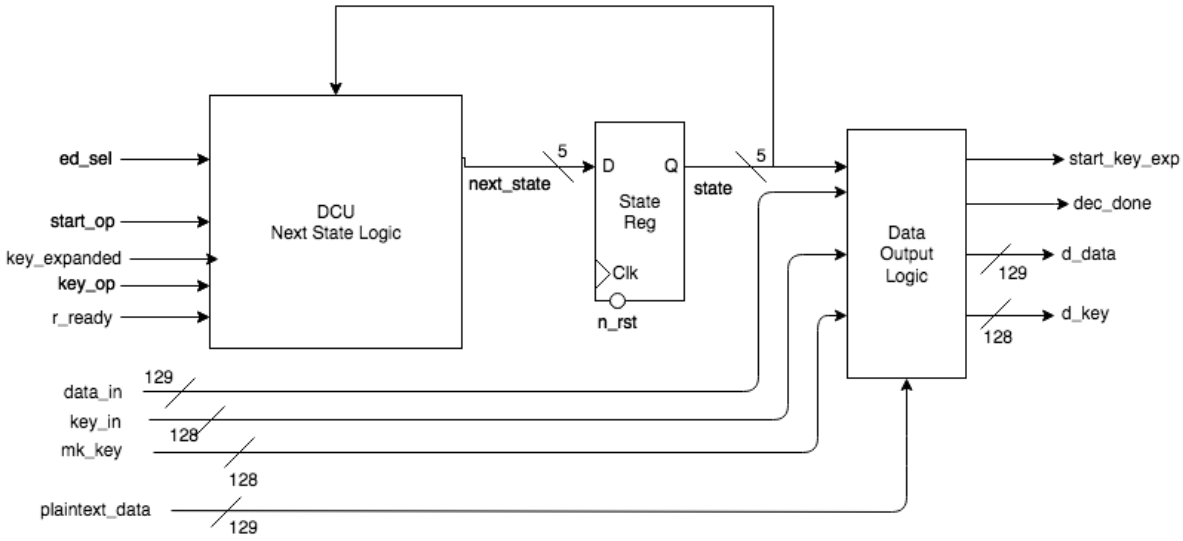


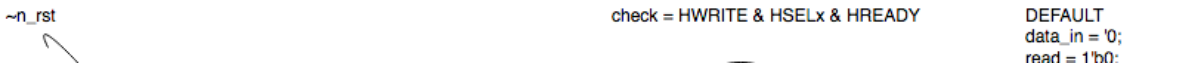
Figure 2m. Decryption Control Unit RTL

Figure 2m shows the RTL diagram for the ECU. This again is just a simple state machine so it has a state register, a next state logic block and an output logic block. We have a couple key and data lines going to output logic block which are just directed to separate output lines d_key and d_data to be used by combinational logic.

Core Area Calculations				
Name of Block	**Category**	**Gate/ FF Count**	**Area (um2)**	**Comments**
DCU State Register	Reg. w/ Reset	3	12,000	26 states needed. 2^5 = 32 states
DCU Next State Logic	Combinational	11	8,250	11*500*1.5
DCU Output Logic	Combinational	8	6,000	8*500*1.5
DCU Key Register	Reg. w/ Reset	128	307,200	128 bits for key
DCU Data Register	Reg. w/ Reset	128	307,200	128 bits for data
DCU Total		278	635,850	Total Area

Description: The way that the area of the State Register is computed using the number of states we have (5 flip flops): 2^5 = 32 states available of which 26 we use. Then the area of the Next State Logic is determined by counting the number of states plus the number of times a state changes based on an input to the Next State Logic block. The Output Logic was then determined by counting the states because the output only depends on the current state (Moore model). Then we originally had two 128 bit registers with a reset for the key and data (which we got rid of).

3.2.4 AHB Slave Unit



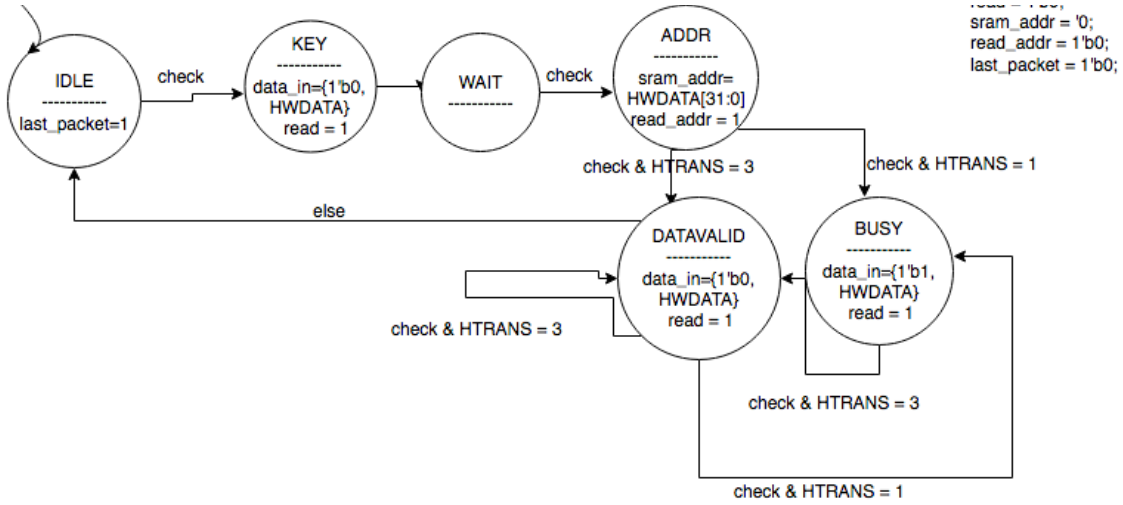


Figure 2n. AHB Slave Unit State Transition Diagram

Figure 2n shows the design flow for our AHB Slave Unit. The AHB module first reads in a key and then address. It then sends that address to the AHB master. It reads the ket and data into a register. This is the preliminary storage of the key and data. From there, it will go on a continuous loop of reading in the next 128 bits of data and sending them to the Receiver Controller Unit. If the CPU sending data sets the transfer type to busy by setting HTRANS to 1, all packets are concatenated with a MSB of 1 to indicate to the AHB Master that this data is not valid, otherwise the master will write it out to SRAM.

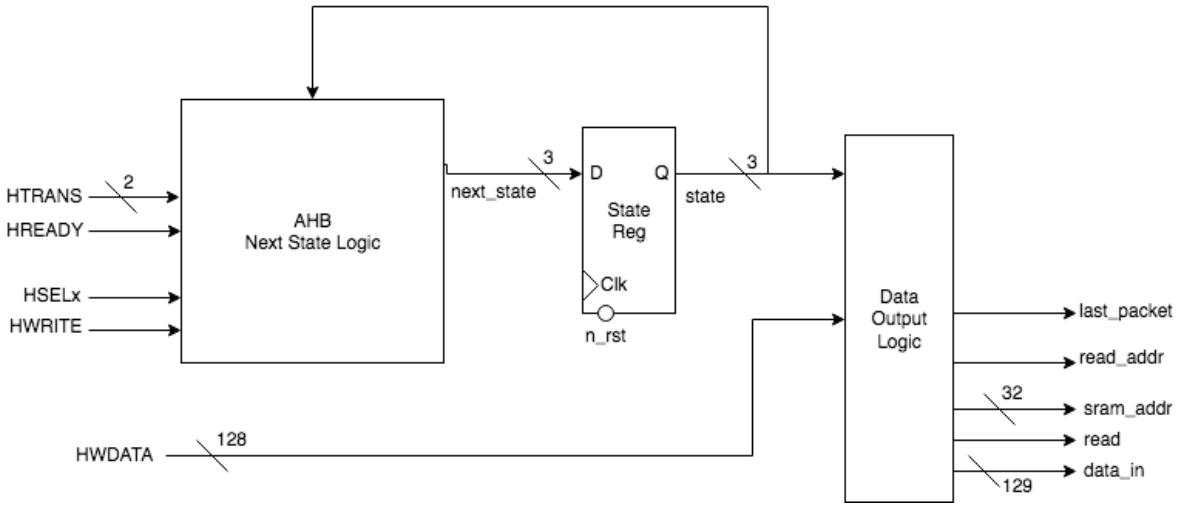


Figure 2o. AHB Slave Unit RTL

Figure 2o shows the RTL diagram for the AHB Slave Unit. This again is just a simple state machine so it has a state register, a next state logic block and an output logic block that determines when to write data out to RCU and send signals needed by AHB Master to write to SRAM. The preliminary waveform for the AHB slave write request is shown in the detailing of the AHB protocol in Section 2.3.2. The proof of this waveform is shown in a read from bus timing diagram screenshot in Section 5.

Core Area Calculations				
Name of Block	**Category**	**Gate/ FF Count**	**Area (um2)**	**Comments**
AHB State Register	Reg. w/ Reset	4	7,200	6 states needed. 2^3 = 8 states
	Combinati			

AHB Next State Logic	Combinational	25	18,750	25*500*1.5
AHB Output Logic	Combinational	11	8,250	11*500*1.5
AHB Total		40	34,200	Total Area

Description: The way that the area of the State Register is computed using the number of states we have (3 flip flops): 2^3 = 8 states available of which 6 we use. Then the area of the Next State Logic is determined by counting the number of states plus the number of times a state changes based on an input to the Next State Logic block. The Output Logic was then determined by counting the states because the output only depends on the current state (Moore model).

3.2.5 AHB Master Unit

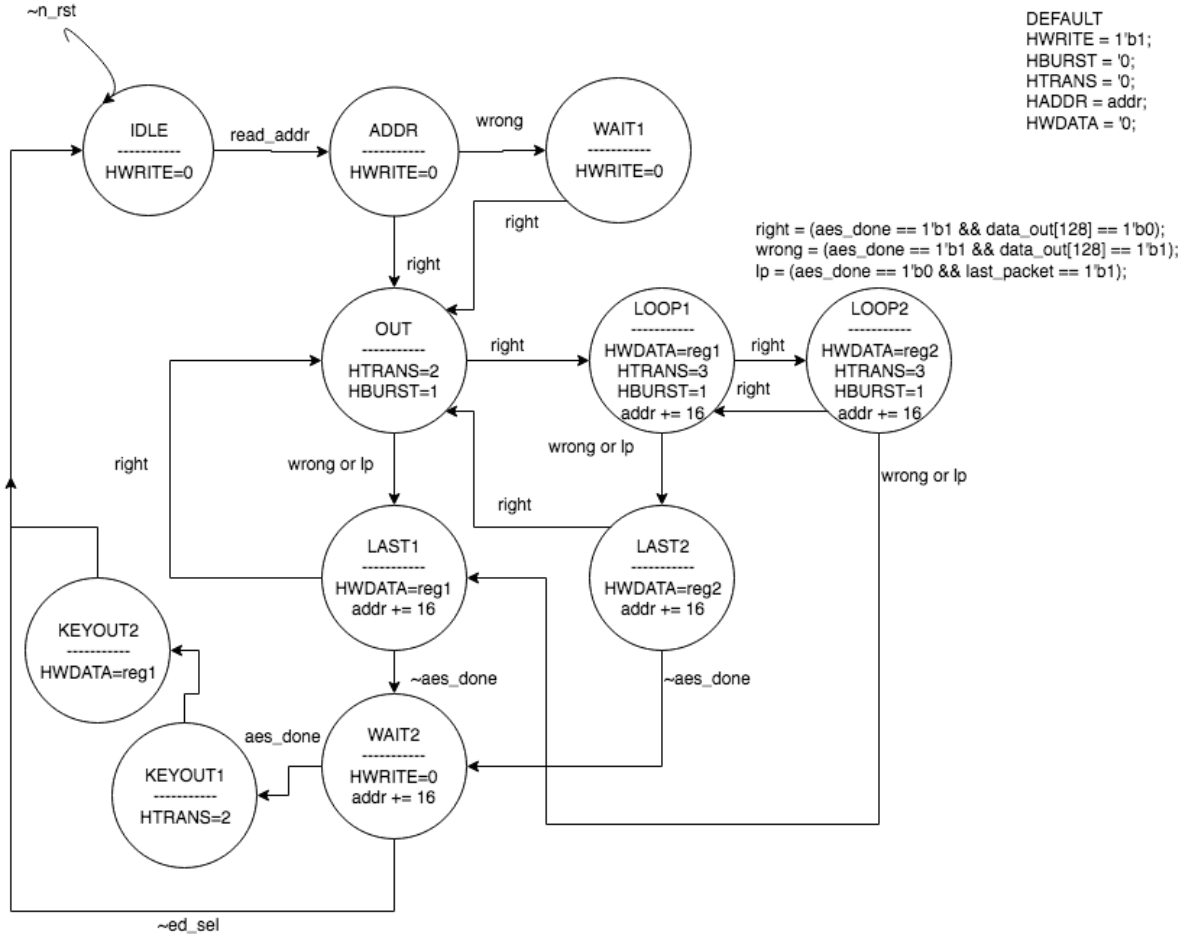
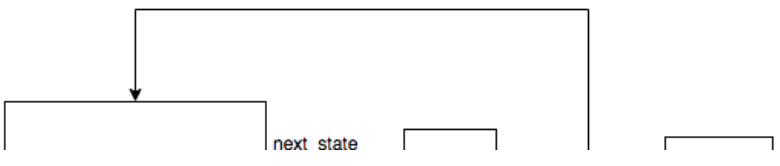


Figure 2p. AHB Master Unit State Transition Diagram

Figure 2p shows the design flow for our AHB Master Unit. The AHB module first reads in an address to decide where to store in sram. It stores the address in a register and adds 16 to it every time after writing data to the bus. From there, it will go on a continuous loop of reading in the next 128 bits of data and sending them to the SRAM every clock cycle. The Master unit also has logic to determine if the packet was read in during an idle transmit of the bus and ignores that packet if the MSB is 1. The state machine looks complicated because of the pipelined nature of the bus but all it actually is doing is looking at the MSB of the data packet and if its 1 it starts the address phase for that packet and writes it out. While its writing the first packet it start the address phase for the next packet based on the MSB of the second packet.



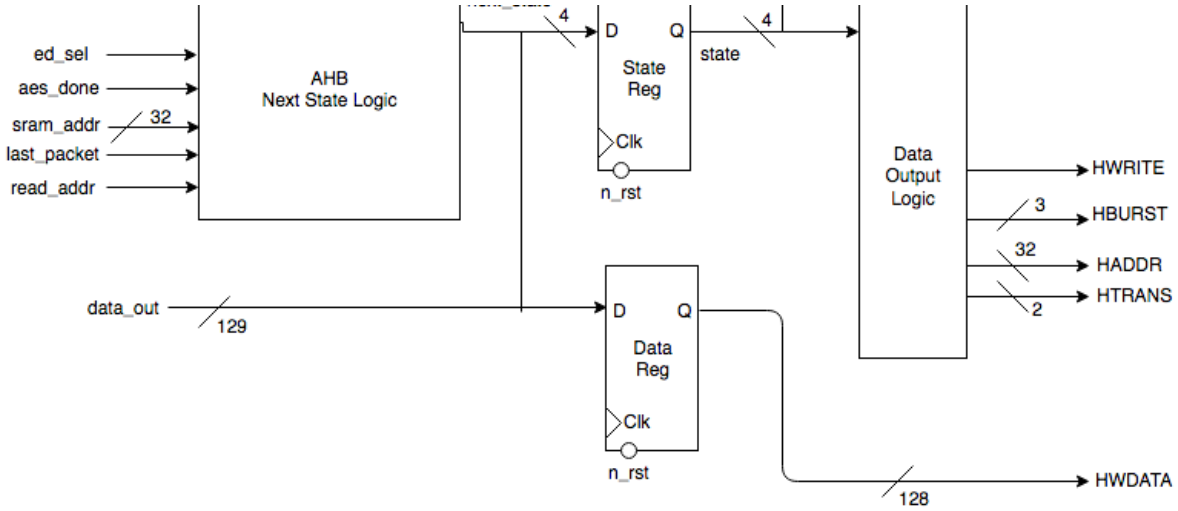


Figure 2q. AHB Master Unit RTL

Figure 2q shows the RTL diagram for the AHB Master Unit. This again is just a simple state machine so it has a state register, a next state logic block and an output logic block that determines when to write data out to SRAM. The preliminary waveform for the AHB write request is shown in the detailing of the AHB protocol in Section 2.3.2. The proof of this waveform is shown in a write to SRAM timing diagram screenshot in Section 5.

Core Area Calculations				
Name of Block	**Category**	**Gate/ FF Count**	**Area (um2)**	**Comments**
AHB State Register	Reg. w/ Reset	4	9,600	11 states needed. 2^4 = 16 states
AHB Next State Logic	Combinational	25	18,750	25*500*1.5
AHB Output Logic	Combinational	11	8,250	11*500*1.5
AHB Total		40	36,600	Total Area

Description: The way that the area of the State Register is computed using the number of states we have (4 flip flops): 2^4 = 16 states available of which 11 we use. Then the area of the Next State Logic is determined by counting the number of states plus the number of times a state changes based on an input to the Next State Logic block. The Output Logic was then determined by counting the states because the output only depends on the current state (Moore model).

3.2.5 Area Analysis

The original estimated area determined by team calculations was around 93 mm^2. This number was originally lower, but as more and more of the device was synthesized, there were better estimations of total core area developed. To calculate this number a single s-box lookup table was first synthesized to determine how large the 360 s-boxes in the core core would be. Then the team estimated that with the combined number of registers and combinational logic needed in the core and other blocks, there would be a total area of 82 mm^2 for the device. This was later revised to 93 mm^2 after the entire core was synthesized. This calculation included a multiplication by 1.5 to determine the rough estimation of wiring needed.

Once synthesized in a layout, the device turned out to actually take up 162.6246 mm^2. It is unfortunate that this area is so large as the original design attempted to take into account all of the wiring needed to build the device. If there was more time before the deadline for the project,

it is theorized that the team would have been able to shrink the area down to close to 100 mm^2 through code and compiler optimizations. Increase the density of the chip also may have yielded a smaller layout. Due to the fact that the design took over 3 hours to layout when the school servers were at minimal load, the team was not able to exhaustively teak and layout the design.

3.3 Design Timing Analysis

3.3.1 Five most Critical Paths for the Current TCM Design

Starting Component	Propagation Delay (ns)	Combinational Logic	Propagation Delay (ns)	Ending Component	Setup Time or Propagation Delay (ns)	Total Path Delay (ns)	Target Clock Period (ns)
Reg-out (DCU)	0.4	Decrypt logic	240.4	Reg-in (RCU)	0.2	241.0	125
Reg-out (ECU) Key	0.4	Encrypt logic	240	Reg-in (AHBSLAVE)	0.2	240.6	125
Reg-out (DCU) Key	0.4	Decrypt logic	240	Reg-in (TCU)	0.2	240.6	125
Reg-out (ECU) Data	0.4	Encrypt logic	225	Reg-in (AHBSLAVE)	0.2	225.6	120
Reg-out (DCU) Data	0.4	Decrypt logic	225	Reg-in (TCU)	0.2	225.6	120

*note: the clock periods shown in this table are purposefully shorter than the critical paths. This phenomenon is compensated for with extra states in the state machines that rely on these paths. The reason for this, is that the TCM is able to run at a faster clock speed, therefore running all of its state machines faster. After a careful analysis of different ways of AES implementation (including inserting intermediate flip flops to shorten the critical path), it was determined that this was the fastest way to compute compute the algorithm.

Path 1:
This path begins in the AES Core. It leads from the register in the DCU that holds the current key block to the register in the RCU that hold the key. This path is used during decryption when the key is brought into the system. When the key is decrypted, data is sent from the DCU key reg, through the Key expansion block, through the Decryption Logic and through a mux into the key reg in the RCU. This all occurs without interruption from a flip flop as most of the AES algorithm is combinational logic. It is estimated that there will be roughly 1,000 gates in series in the AES algorithm based off of an earlier assumption that there will need to be 4,480 gates in both the encrypt or decrypt block. This includes the contribution of roughly 50 gates total used in the key expansion block. Multiplying this 1000 by 0.2 propagation delay of a single gate yields 200ns. Next, multiplying by the 1.2 factor of wiring overhead yields a total of 240 ns critical path delay. The number 240.4 comes from the fact that in this specific path, there will need to be a small amount of combinational logic to route the key into the key reg in the RCU.

Path 2:
This path begins in the ECU within the AES Core. It goes from the register that holds the current key data to the register in the AHB Slave that transmits it onto the data bus. This path is used when the signals originating from the key register in the ECU travel through the key expansion block, the Encryption Logic block, and end up in the AHB Slave out register. The number 240 comes from the estimation that there will be roughly 1,000 gates in series in the AES algorithm based off of an earlier assumption that there will need to be 4,480 gates in both the encrypt or

decrypt block. This includes the contribution of roughly 50 gates total used in the key expansion block. Multiplying this 1000 by 0.2 propagation delay of a single gate yields 200ns. Next, multiplying by the 1.2 factor of wiring overhead yields a total of 240 ns critical path delay.

Path 3:
This path begins in the AES Core. It leads from the register in the DCU that holds the current key block to the register in the AHB Slave. This path is used during decryption of any 128 bit data that is in the DCU register. This path is used when the signals originating from the key register in the DCU travel through the key expansion block, the Decryption Logic block, and end up in the AHB Slave out register. The number 240 comes from the estimation that there will be roughly 1,000 gates in series in the AES algorithm based off of an earlier assumption that there will need to be 4,480 gates in both the encrypt or decrypt block. This includes the contribution of roughly 50 gates total used in the key expansion block. Multiplying this 1000 by 0.2 propagation delay of a single gate yields 200ns. Next, multiplying by the 1.2 factor of wiring overhead yields a total of 240 ns critical path delay.

Path 4:
This path begins in the AES Core. It leads from the register in the ECU that holds the current data block to the register in the AHB Slave that transmits it onto the data bus. This path is used during decryption of any 128 bit data that is in the ECU register. This path is used when the signals originating from the key register in the ECU travel through the Encryption Logic block, and end up in the AHB Slave out register. The number 225 comes from the estimation that there will be roughly 935 gates in series in the AES algorithm based off of an earlier assumption that there will need to be 4,480 gates in both the encrypt or decrypt block. Multiplying this 935 by 0.2 propagation delay of a single gate yields 200ns. Next, multiplying by the 1.2 factor of wiring overhead yields a total of 225 ns critical path delay.

Path 5:
This path begins in the AES Core. It leads from the register in the DCU that holds the current data block to the register in the AHB Slave. This path is used during decryption of any 128 bit data that is in the DCU register. This path is used when the signals originating from the key register in the DCU travel through the the Decryption Logic block, and end up in the AHB Slave out register. The number 225 comes from the estimation that there will be roughly 935 gates in series in the AES algorithm based off of an earlier assumption that there will need to be 4,480 gates in both the encrypt or decrypt block. Multiplying this 935 by 0.2 propagation delay of a single gate yields 200ns. Next, multiplying by the 1.2 factor of wiring overhead yields a total of 225 ns critical path delay.

3.3.2 Estimated Critical Path vs. Actual Critical Path

```
Path 1: VIOLATED Setup Check with Pin AESCORE/ENCRYPTIONFULL/round1_reg[45]/CLK
Endpoint:   AESCORE/ENCRYPTIONFULL/round1_reg[45]/D (^) checked with  leading
edge of 'clk'
Beginpoint: AESCORE/ACTUAL_ECU/state_reg[4]/Q       (v) triggered by  leading
edge of 'clk'
Path Groups: {reg2reg}
Analysis View: osu05
Other End Arrival Time          1.002
- Setup                         0.415
+ Phase Shift                  15.000
- Uncertainty                   0.050
= Required Time                15.537
- Arrival Time                 17.678
= Slack Time                   -2.141
```

Figure 3a. Timing Report.

Figure 3a shows the output of the innovus timing report. As can be seen here the team ran

into the unfortunate problem of not having enough time to adjust layout constraints to avoid setup time errors. This can easily be fixed by adjusting the clock speed to run slower to avoid the errors, or through further optimizing the design. Since Innovus took on average 3-4 hours to generate a layout, there was not enough time to fully optimize and fix these errors. All figure 3a really shows is that the current design must be run with a slightly slower clock speed to function and be stable.

The estimated clock speed for this design was originally set at around 150MHz. This estimation was determined using the timing reports generated by each synthesized block estimating the critical path to be around 5-10ns. The issue the team encountered here was that the actual location of the critical path in the device was incorrectly estimated. The original path thought to be the critical path was that from a register in the RCU holding the current data packet to the register at the output of round 1 (Round 0 length is insignificant so there is no register after it). In reality though, this was not the case as there was actually a path caused by the ed_sel line leading from the CCU and cascading through the AES Core and into a reg in the AHB Master. This was fixed by registering the ed_sel value inside the AES Core, but after another innovus layout (4 more hours) and timing report were generated it was found that the critical path originally thought to be 10ns was actually 17ns. This path would not be possible to fix without greatly reworking our design and further increasing our area. Unfortunately there was not time for this, so the critical path currently remains at ~17ns.

4 Success Criteria

4.1 Fixed Criteria

1) (2 points) Test benches exist for all top-level components and the entire design. The test benches for the entire design can be demonstrated or documented to cover all of the functional requirements given in the design specific success criteria.

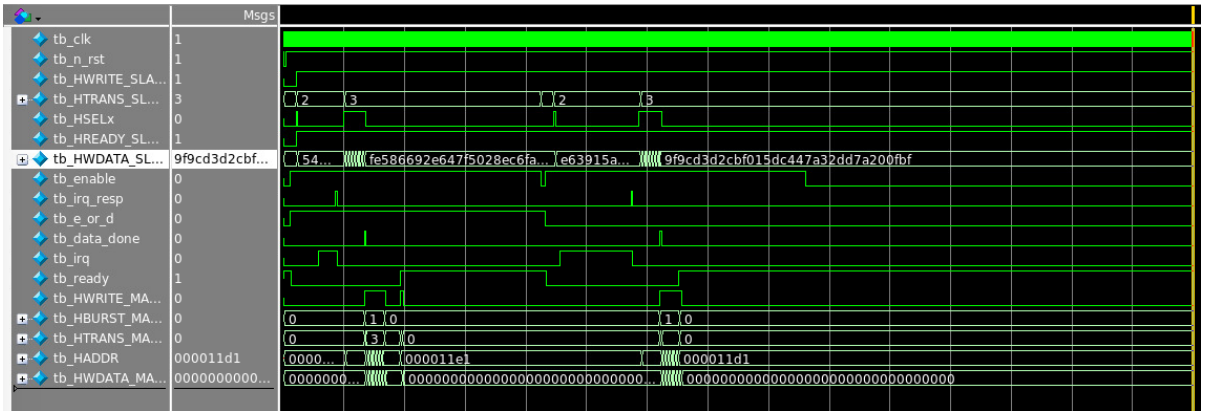
See: Submission of all verilog test bench files and the Appendix
Criteria met

2) (4 points) Entire design synthesizes completely, without any inferred latches, timing arcs, and, sensitivity list warnings.

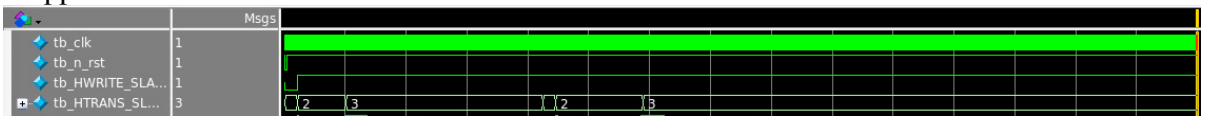
See: Top level report file as well as log file in the Appendix and submission of files
Criteria met

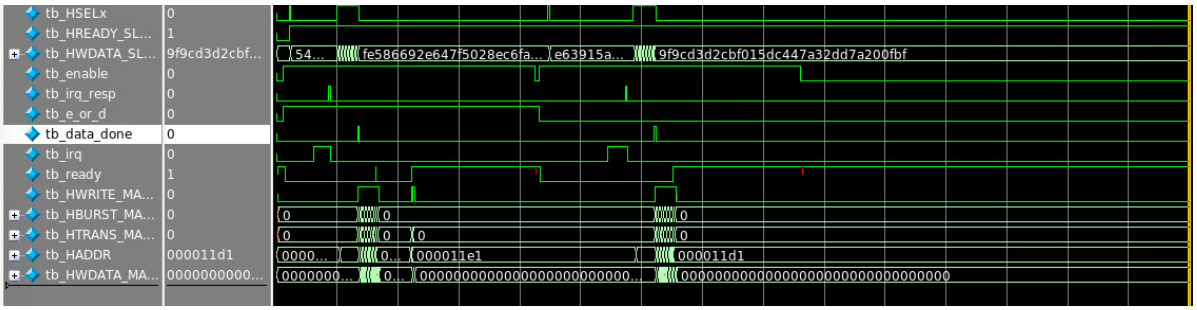
3) (2 points) Source and mapped version of the complete design behave the same for all test cases. The mapped version simulates without timing errors except at time zero.

Source version



Mapped version

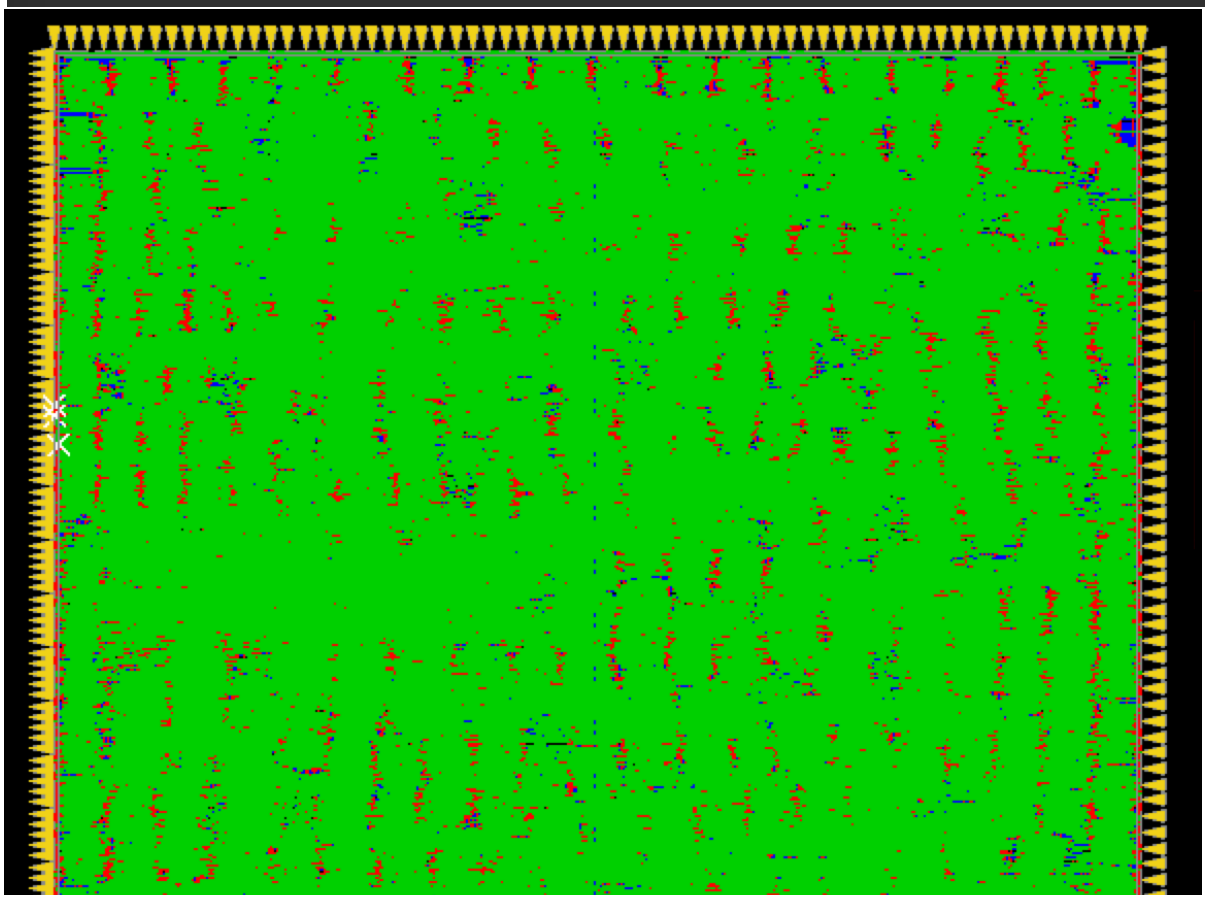


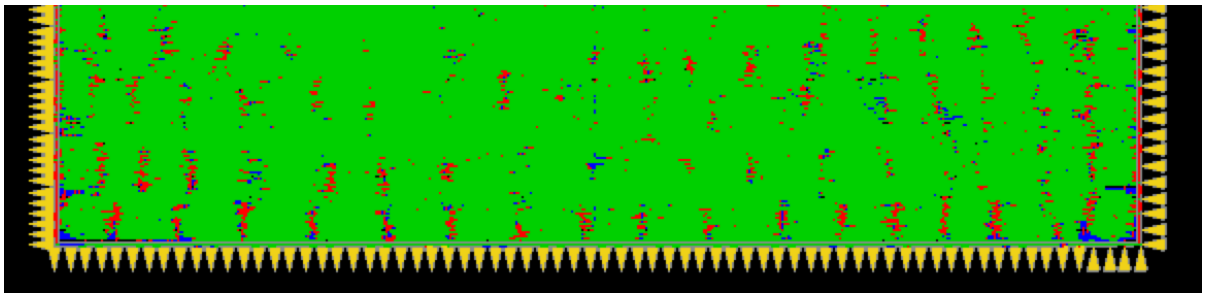


Criteria met

4) (2 points) A complete IC layout is produced that passes all geometry and connectivity checks.

```
############################################################
#  Generated by:      Cadence Innovus 16.12-s051_1
#  OS:                Linux x86_64(Host ID ecegrid-thin1.ecn.purdue.edu)
#  Generated on:      Wed May  2 15:16:01 2018
#  Design:            top_level
#  Command:           verifyConnectivity -type all -noWeakConnect -noUnroutedNet -connLoop -noSoftPGConnect -error 1000 -warning 50
############################################################
Verify Connectivity Report is created on Wed May  2 15:16:01 2018



Begin Summary
    Found no problems or warnings.
End Summary
############################################################################
#  Generated by:      Cadence Innovus 16.12-s051_1
#  OS:                Linux x86_64(Host ID ecegrid-thin1.ecn.purdue.edu)
#  Generated on:      Wed May  2 15:18:47 2018
#  Design:            top_level
#  Command:           verifyConnectivity -type all -geomLoop -noSoftPGConnect -error 1000 -warning 50
############################################################################
Verify Connectivity Report is created on Wed May  2 15:18:47 2018



Begin Summary
    Found no problems or warnings.
End Summary
```





See: top_level.conn.rpt & top_level.conn.rpt.old
Criteria met

5) (2 points) The entire design complies with targets for area, pin count, throughput (if applicable), and clock rate. The final targets for these parameters will be determined by course staff based on your design review. Failure to reach any of the targets will result a score of 1 out of 2 provided that you are within 50% on area, 10% on pin count, and 25% on throughput. Doing worse in any category will result in a score of 0 out of 2.

Criteria not met for area and throughput
Pin Count Criteria has been met

4.2 Design Specific Criteria

1) Demonstrate by simulation of Verilog test benches that the entire design is able to respond to encryption and decryption requests and successfully encrypts and decrypts data read and wrote using AHB bus protocols. (2 point)

- Criteria met

2) Demonstrate by simulation of Verilog test benches that the encryption logic block is able to successfully encrypt data and decryption logic block is able to successfully decrypt data based on the AES-128 standard. (2 point)

- Criteria met

3) Demonstrate by simulation of Verilog test benches that at the end of file AES Core block successfully encrypts and decrypts the supplied auxiliary key with the master key in the ROM according to AES-128 standards. (1 point)

- Criteria met

4) Demonstrate by simulation of Verilog test benches that the Receiver block is able to successfully receive key and data and store it in the right registers to be encrypted or decrypted. (1 points)

- Criteria met

5) Demonstrate by simulation of Verilog test benches that the AHB-Slave block of the AHB-Lite Bus properly responds to read and write transfers on the bus. (1 points)

- Criteria met

6) Demonstrate by simulation of Verilog test benches that the Inverse Mix Columns operation of the Decryption Unit in the AES Core has the correct functionality.

- Criteria met

- Criteria met

Note: For more detail on the Design Specific Success Criteria, look into
Section 5: Design Verification

5 Design Verification

5.1 Fixed Criteria

Correct Chip Response to Encryption & Decryption Requests (Top Level)

- Shown in Demo: Yes
- DSSC(s) Proved: 1
- Highest Level of Design Module(s) involved:
  - Total Design/Chip
- Test bench Expectations/Requirements:
  - Have temporary data registers to capture encryption results for reuse in decryption request
  - Use pseudo random number generation to construct encryption key
  - Use pseudo random number generation to file data for encryption/decryption
  - Emulate encryption request AHB-Lite bus transaction
  - Emulate decryption request AHB-Lite bus transaction
- No external or premade references are needed
- No pre/post processing is needed
- Main Verification Test Steps:
  1. Generate encryption request bus transaction for design
  2. Capture encrypted data result
  3. Generate decryption request bus transaction for design using prior encrypted data result
  4. Check that decrypted data result matches original raw data
  5. Repeat Steps 1-4 for multiple randomized data values

Proof (The following screenshot also will act as proof for some of the criteria below):

Encryption:

Figure 5a

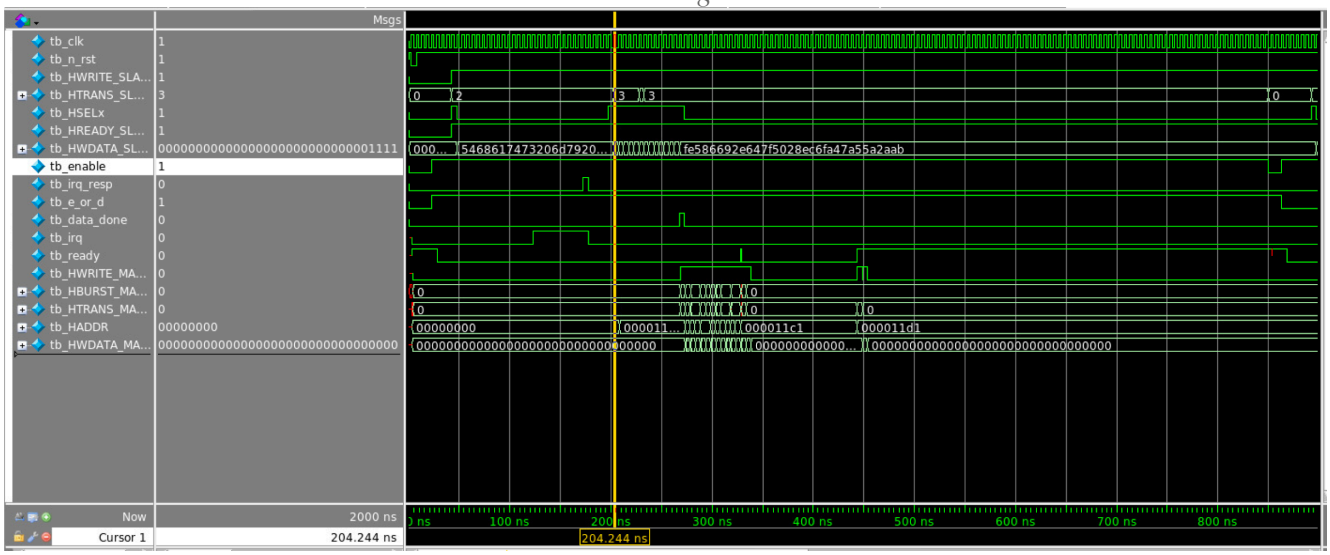


Figure 5a shows the start of the encryption cycle. At the top left side of the image the AHB Slave bus signals can be seen, commanding the device to begin listening for the first packet (Note: this is after an enable signal has already been set to initialize the device). The selected cursor time allows for the viewing of the data on the "tb_HWDATA_SLAVE" shown on the left. This data is 28 zeros and four ones. It is the first packet of the sequence and will be used as an address to iterate from when writing to SRAM.

Figure 5b

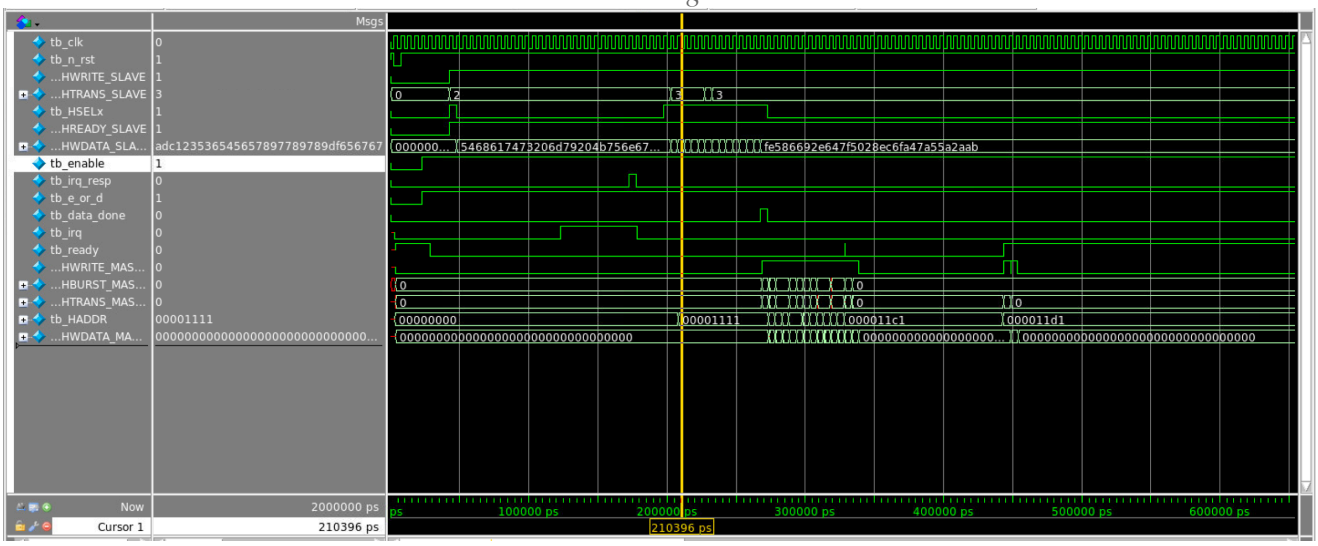


Figure 5b shows the cursor at the time of the arrival of the first data packet to be encrypted. The value of this packet is shown on the "tb_HWDATA_SLAVE" line on the left (It starts with "adc"). Note how at this point the address being output to the SRAM from the tb_HADDR lines of the AHB Master now show the address value passed to the device earlier.

Figure 5c

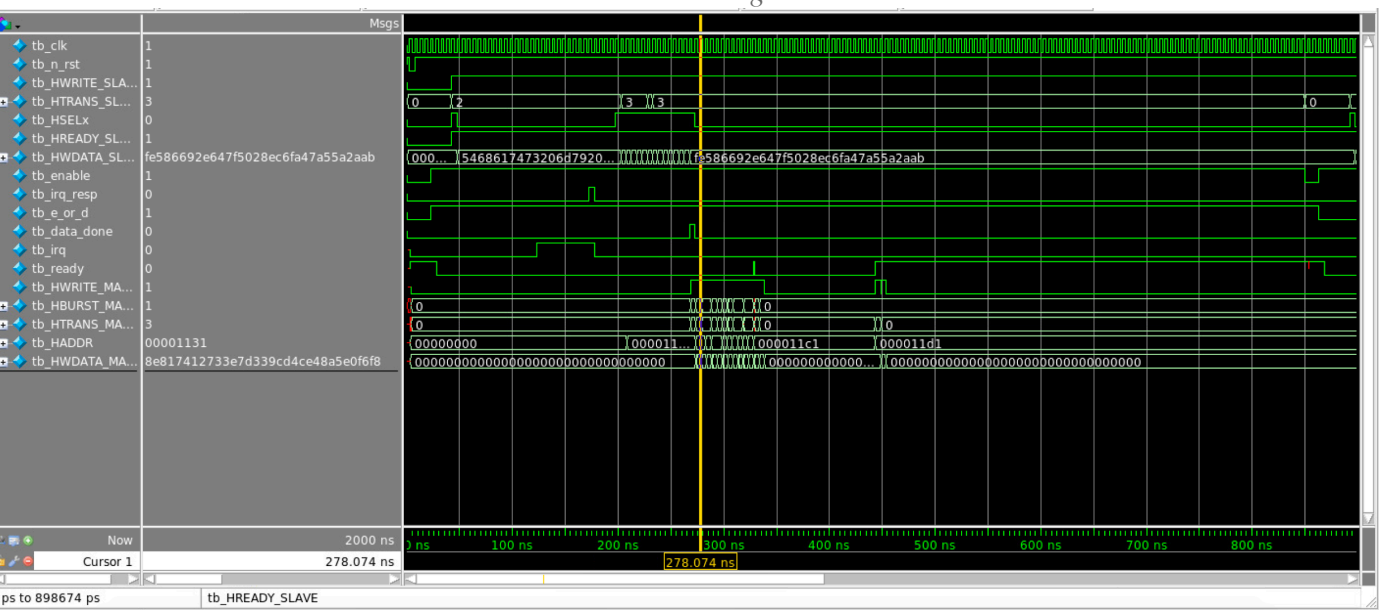
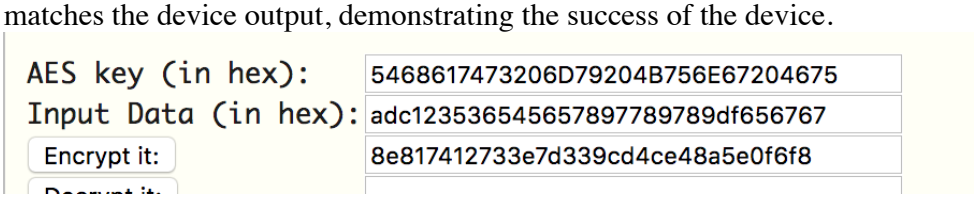


Figure 5c shows the first encrypted packet being output on the "tb_HWDATA_MASTER" lines shown on the left. Notice how this value starts with "8e817". As seen in figure 5d below, the data packet observed in figure 5b was put into an online AES 128 calculator along with the key value shown on the "tb_HWDATA_SLAVE" the first time HSELx goes high (546861747….). The output of the calculator is "8e817412733e7d339cd4ce48a5e0f6f8". This output exactly matches the device output, demonstrating the success of the device.



Decrypt it:

Figure 5e

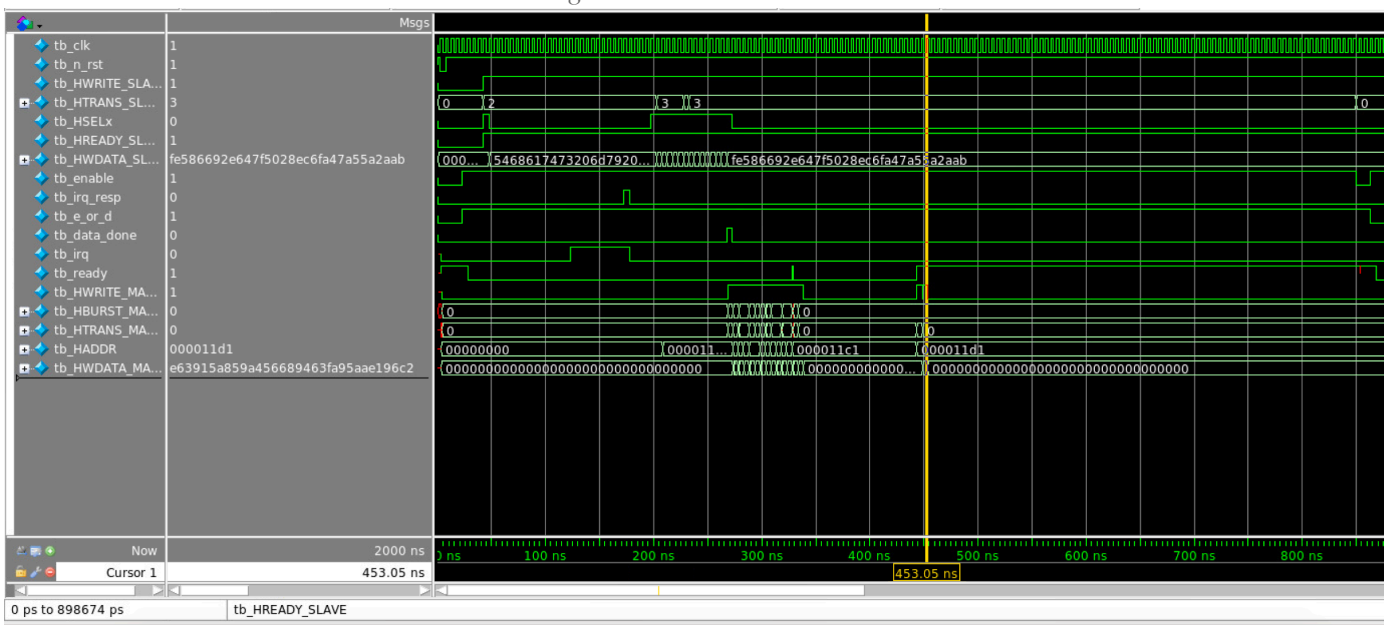
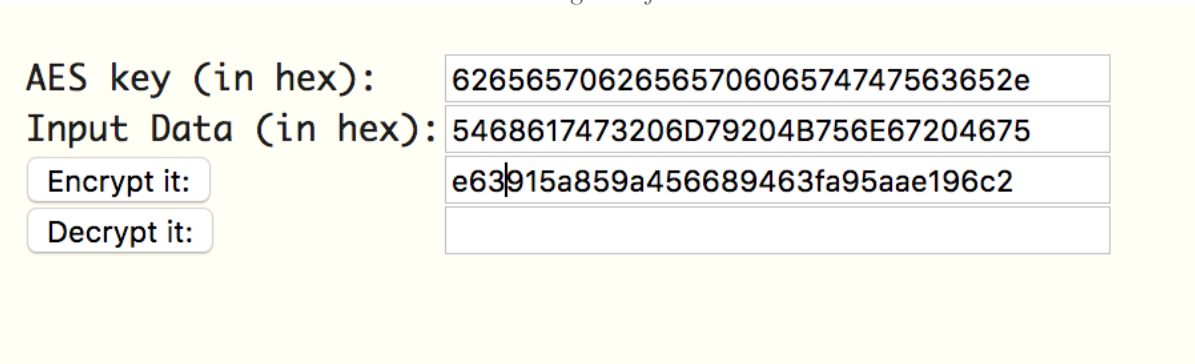


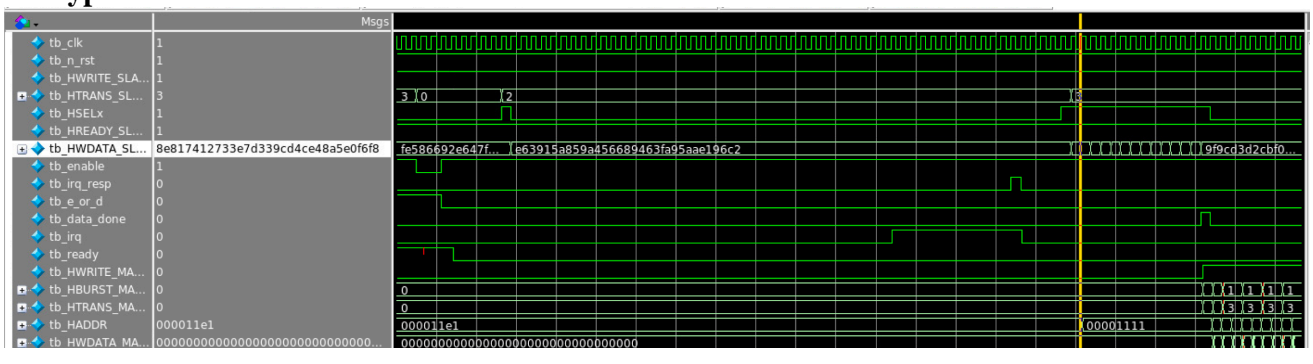
Figure 5e

Figure 5e shows the encrypted key packet on the "tb_HWDATA_MASTER" output lines. This data was calculated several clock cycles after the assertion of the tb_data_done signal due to the time required to expand the master key and encrypt the runtime key. The plaintext data in this case is the 546861747…. key from earlier, but the key used in the encryption is now the internal master key. In this test bench this value is hardcoded into a ROM module as "626565706265657060657474756 3652e". Show in figure 5f below, the output from figure 5e matches the calculations performed by an online calculator.

Figure 5f



Decryption:



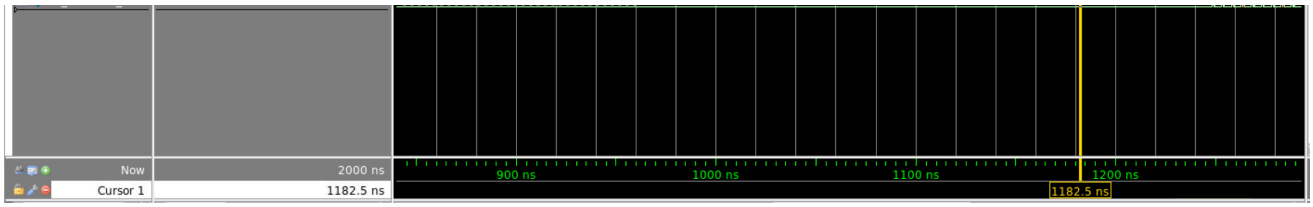


Figure 5g

Figure 5g shows the start of the decryption cycle. Notice how once the address packet is passed into the device, there is a longer delay before the normal packet stream starts. This is due to the need to intake the encrypted auxiliary key, expand the master key, decrypt the auxiliary key, and finally expand the auxiliary key. Also notice how the first data packet in is the previous first encrypted output from figure 5c. Finally, notice how the irq signal is used to send an interrupt to the CPU to let it know the device has finished all of the key decryption and manipulations and is ready to receive data. The CPU can then wait a random amount of clock cycles before sending an irq response to the device to complete the handshake.

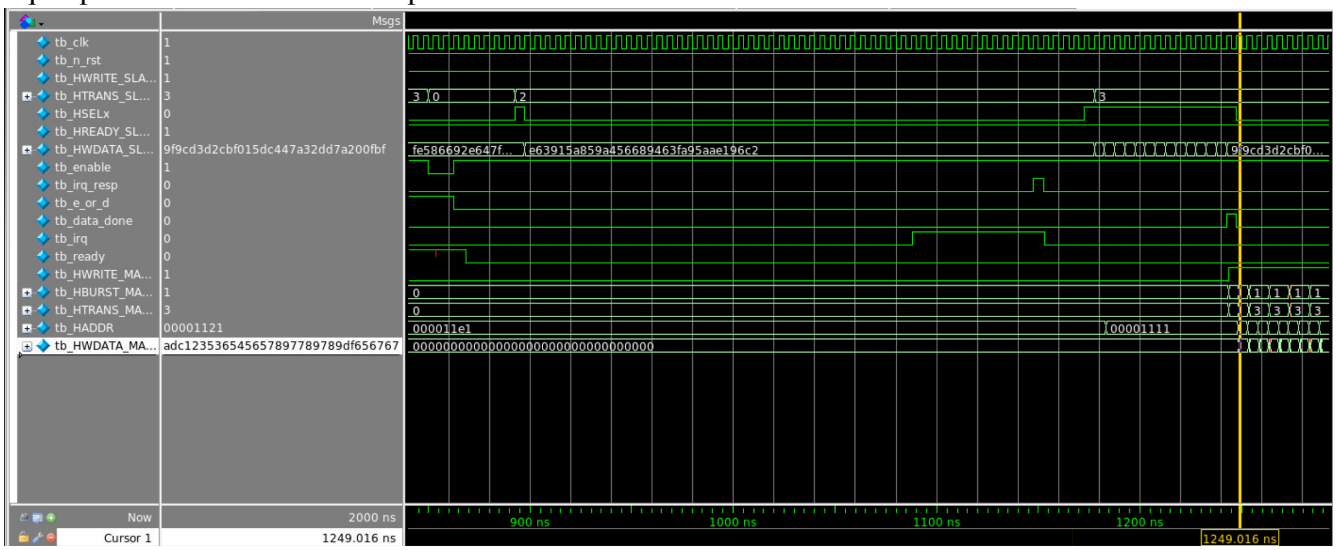


Figure 5h

Figure 5h shows the decrypted output (on the "tb_HWDATA_MASTER" lines) matching the first packet originally sent to the device in figure 5b.

Correctness of AES Core Encryption/Decryption (Top Level)

- Shown in Demo: Yes
- DSSC(s) Proved: 2, 3 4 and 6
- Highest Level of Design Module(s) involved:
  - Total Design/Chip
- Test bench Expectations/Requirements:
  - Have temporary data registers to capture encryption results for reuse in decryption request
  - Check each encrypted/decrypted block out of the TCM for correctness using the website referred.
  - Compare the original data to the loopback data out of the decryption cycle to verify correctness
- Use https://aesencryption.net/ as reference to validate aes correctness
- No pre/post processing is needed
- Main Verification Test Steps:
  1. Check each encrypted block from the loopback test using the above website AES implementation

2. Check each decrypted block from the loopback test using the above website AES implementation

Proof:

- This has already been proven with the above screenshots.

AMBA AHB-Lite Slave Interfacing (Top Level)

- Shown in Demo: Yes
- DSSC(s) Proved: 5
- Highest Level of Design Module(s) involved:
    - Total Design/Chip
- Test bench Expectations/Requirements:
    - Have test vector of samples of each type of bus transaction and correct chip response information
    - Chip response should be verified offline by checking against standards
- No external or premade references are needed
- No pre/post processing is needed
- Main Verification Test Steps:
    1. Emulate bus transaction sample from test vector
    2. Check chip response against the correct response values in the test vector
    3. Repeat Steps 1-2 for all entries in test vector

Proof:

- This has already been proven with the above screenshots.

Backup and Sub-Module Tests

Correctness of AES Core Encryption/Decryption (AES Core Block)

- Shown in Demo: Only if not shown using Top Level
- DSSC(s) Proved: 2, 3 and 6
- Highest Level of Design Module(s) involved:
    - AES Cryptographic Core
- Test bench Expectations/Requirements:
    - Check each encrypted/decrypted block out of the TCM for correctness using the website referred. Or,
    - Loop the encrypted data back in to decrypt and then check.
- No external or premade references are needed
- No pre/post processing is needed
- Main Verification Test Steps:
    1. Supply a key and data to be encrypted
    2. Check the encrypted data using the data from the website implementation above.
    3. Capture the encrypted data, and loop it back to be decrypted.
    4. Compare the decrypted data to the original data or the data from the website implementation above for correctness.

Proof:

- This has already been proven with the above screenshots.

AMBA AHB-Lite Slave Interfacing (AHB Slave Block)

- Shown in Demo: Only if not shown using Top Level
- DSSC(s) Proved: 5
- Highest Level of Design Module(s) involved:
    - AHB-Lite Slave Block
- Test bench Expectations/Requirements:
    - Have test vector of samples of each type of bus transaction and correct chip response information
    - Chip response should be verified offline by checking against standards
- No external or premade references are needed
- No pre/post processing is needed
- Main Verification Test Steps:

- Main Verification Test Steps:
    1. Emulate bus transaction sample from test vector
    2. Check chip response against the correct response values in the test vector
    3. Repeat Steps 1-2 for all entries in test vector

Proof:

- This has already been proven with the above screenshots.

Correctness of RCU (Receiver Control Unit Block)

- Shown in Demo: Only if not shown using Top Level
- DSSC(s) Proved: 4
- Highest Level of Design Module(s) involved:
    - Receiver Control Unit Block
- Test bench Expectations/Requirements:
    - Have input vector of key bits to check against output key bits
    - Have input vector of data bits to check against output data bits
- No external or premade references are needed
- No pre/post processing is needed
- Main Verification Test Steps:
    1. Emulate data input from AHB Slave block
    2. Check key output response against key passed in
    3. Check data output response against data block passed in

Proof:

- None of this design would have worked without the RCU reading the key and data packets into internal registers to be routed to the AES Core. See screenshots above for proof of working design.

Correctness of CCU (Cryptographic Control Block)

- Shown in Demo: Only if not shown using Top Level
- DSSC(s) Proved: 1
- Highest Level of Design Module(s) involved:
    - Cryptographic Control Unit
- Test bench Expectations/Requirements:
    - Use different dummy data for transaction payloads to check that relevant signals are being received/sent
- No external or premade references are needed
- No pre/post processing is needed
- Main Verification Test Steps:
    1. Assert normal operation signals in order
    2. Obverse that outputs are triggered in the correct order
    3. Iterate through all possible state transition paths

Proof:

- None of the top level timing would work without this control unit handling the io signals (signals other than bus signals) as well as activating certain state machines on time. See screenshots above for proof of working design.

Correctness of Encryption Logic Block (sub-module)

- Shown in Demo: Only if not shown using Top Level
- DSSC(s) Proved: 2
- Highest Level of Design Module(s) involved:
    - Encryption Logic
- Test bench Expectations/Requirements:
    - Use random key and data as sample inputs to the block
    - Check output for correctness using the output from the website referenced.
- Use https://aesencryption.net/ as reference to validate aes correctness
- No pre/post processing is needed
- Main Verification Test Steps:

1. Send data and key to the block to be encrypted.
2. Compare the encrypted data to the output from the website implementation for correctness.

Proof:

- This has already been proven with the above screenshots.

Correctness of Decryption Logic Block (sub-module)

- Shown in Demo: Only if not shown using Top Level
- DSSC(s) Proved: 2, 6
- Highest Level of Design Module(s) involved:
    - Decryption Logic
- Test bench Expectations/Requirements:
    - Use encrypted data and key from the website referenced as inputs.
    - Check the decrypted data using the original data that was encrypted using the website.
- Use https://aesencryption.net/ as reference to validate aes correctness
- No pre/post processing is needed
- Main Verification Test Steps:
    1. Get encrypted data and key from the website implementation or the encryption logic block and use it to decrypt data.
    2. Compare the decrypted output to the original data to verify.

Proof:

- This has already been proven with the above screenshots.

Correctness of Key Expansion Block (sub-module)

- Shown in Demo: No
- DSSC(s) Proved: 2
- Highest Level of Design Module(s) involved:
    - Key Expansion
- Test bench Expectations/Requirements:
    - Use encrypted data and key from the website referenced as inputs.
    - Check the decrypted data using the original data that was encrypted using the website.
- To validate key expansion correctness, use expanded key examples on:

○ https://kavaliro.com/wp-content/uploads/2014/03/AES.pdf
○ http://www.cs.siue.edu/~tgamage/S17/CS490/L/WK05.pdf
○ https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf

- No pre/post processing is needed
- Main Verification Test Steps:

1. Get key from the website implementation and send it to the key expansion block.
2. Compare the expanded key to the expanded key given in the reference.
3. Repeat for multiple examples from the reference.

Proof:

- This has already been proven with the above screenshots.

6 Project Management

Ryan Devlin:

- Responsibilities:
    - State diagrams: CCU, ECU, DCU, AHB Slave (initial)
    - Combinational logic design: AES Core
    - Overall logic timing optimizations
    - Overall area optimizations

    - Several test benched and debugging

Samuale Yigrem:

- Responsibilities:
    - AES Core: Encryption & Decryption Logic with test benches
    - AES Core: Key Expansion Logic with test bench
    - AES Core: Overall test bench
    - RCU: State Transition Diagram Logic with test bench
    - DCU: State Transition Diagram Logic with test bench
    - AHB Slave: Test bench
    - Incremental test benches for the integration of submodules of the design

Dhairya Agrawal:

- Responsibilities:
    - State diagrams: CCU, RCU, ECU, DCU, AHB Slave/Master
    - Combinational logic design: AES Core
    - Overall logic timing optimizations
    - Overall area optimizations
    - Several test benches and debugging

Samanth Mottera:

- Responsibilities:
    - State diagrams: ECU, DCU, AHB Slave
    - AHB slave test bench and top level test bench work
    - debugging

7 Appendix

Account and directory of where all of the files are located: mg125/ASIC_Design/Project

- **Design Verilog Code modules**
    - **Top Level structural Verilog Code:** source/top_level.sv
    - **CCU:** source/ccu.sv
    - **RCU:** source/RCU.sv
    - **AHB SLAVE:** source/AHB_slave.sv
    - **AHB MASTER:** source/AHB_master.sv
    - **AES CORE:** source/aes_core.sv
    - **KEY EXPANSION:** source/key_expansion.sv
        - source/rcon.sv
        - source/s_box_4.sv
        - source/keyScheduleCore.sv
    - **ECU:** source/ECU.sv
    - **DCU:** source/DCU.sv
    - **ENCRYPTION:** source/encryptionFull.sv
        - source/multiply2.sv
        - source/mixCol32.sv
        - source/mixCol128.sv
        - source/s_box_16.sv
        - source/shiftRows.sv
        - source/keyAdd.sv
        - source/encryptionRound.sv
        - source/round10.sv
    - **DECRYPTION:** source/decryptionFull.sv
        - source/multiply4.sv
        - source/multiply8.sv
        - source/multiplyC.sv
        - source/invMixCol32.sv

- source/invMixCol128.sv
- source/invShiftRows.sv
- source/inv_s_box_16.sv
- source/decryptionRound.sv
- source/round0.sv

- **Test Benches**
  - **Top level test:** source/tb_top_level.sv
  - **CCU:** source/tb_ccu.sv
  - **RCU:** source/tb_RCU.sv
  - **AHB SLAVE:** source/tb_AHB_slave.sv
  - **AHB MASTER:** source/tb_AHB_master.sv
  - **AES CORE:** source/tb_aes_core.sv

- **Mapped Verilog Code**
  - **Top level mapped file:** mapped/top_level.v

- **Reports generated during synthesis and layout**
  - **Geometry Check Layout:** top_level.conn.rpt
  - **Connectivity Check Layout:** top_level.conn.rpt.old
  - **Mapped Top Level report file:** reports/top_level.rep
  - **Mapped Log File (Latch Check):** top_level.log

- **Final Report Files**
  - **Final Report:** final_report.pdf