Image Edge Detection with RC4 cipher using AHB Lite-Based SoC Design

Final Report

Submission date: 5/4/2017

Dimcho Karakashev, Ruchir Aggarwal, Gautam Rangarajan, Ribhav Agarwal
Wednesday 11:30 am Lab (Project Team 4)
TAs: Eric & Cooper

1. Executive summary

With each passing day, our world becomes more and more fast paced. To keep up with this exponential growth, the demand for quick and easy-to-use technology/services has grown. A simple example of this is the upsurge of apps like Uber and PayPal. However, a critical concern with respect to these on-demand-services, is the enormous lack of strong security systems. Hacking of consumers' personal details has become almost an everyday occurrence. This poses a serious threat to both the consumer, as well as the company that offers these services. The only way to get past this hurdle is to invest in technology that provides top-of-the-line security, without compromising on convenience or simplicity.

Our proposed design is an Image Edge Detection system with an RC4 cipher that uses an AHB-Lite-Based intra-system communication. The chip will receive RC4 encrypted images, and perform edge detection on the decrypted file. This edge detected image will then be outputted for further image processing and image matching to be done by other external modules. As mentioned earlier, rapid services, combined with thorough security protocol has become a necessity in our world. Our design aims to be a piece of the puzzle that helps achieve this. It will cater in specific to the banking industry.

One of the new ventures in the sphere of banking is instant authorization of payments. To be more precise, banks have begun implementing fingerprint authorization of transactions. This will allow for faster payment methods than ever seen before. However, the issue that is preventing banks from putting out this technology is problem of ensuring sufficient security. Our chip aims to get rid of this obstacle. Our SoC will accept RC4 encrypted fingerprint images, which will then be decrypted and sent into an edge detection module. This will in turn perform edge detection on the image, and put it on the bus, so that it can be sent to an external system that will do the fingerprint matching. If the fingerprint was found to be matching, the payment will then be authorized.

Designing an ASIC to perform this function will help ensure that these processes are completed as fast as possible. The hardware will be optimized to allow for rapid decryption and edge detection. Our chip will also have a shared AHB-Lite bus for intra system communication. This will serve as how the chip will receive and transmit the encrypted and edge detected fingerprint image.

Successful implementation of the design will require the following resources-

- AHB-Lite SoC bus standard documentation
- Verilog HDL Simulation and Design Synthesis Tool Chain
- RC4 algorithms
- Edge Detection implementation algorithms

The following document content will describe:

- Intended usage expectations and constraints for the design.
- Intended main implementation architecture.
- Implementation of the entire design module.

2. Design specifications

2.1. System Usage

2.1.1. System Usage Diagram

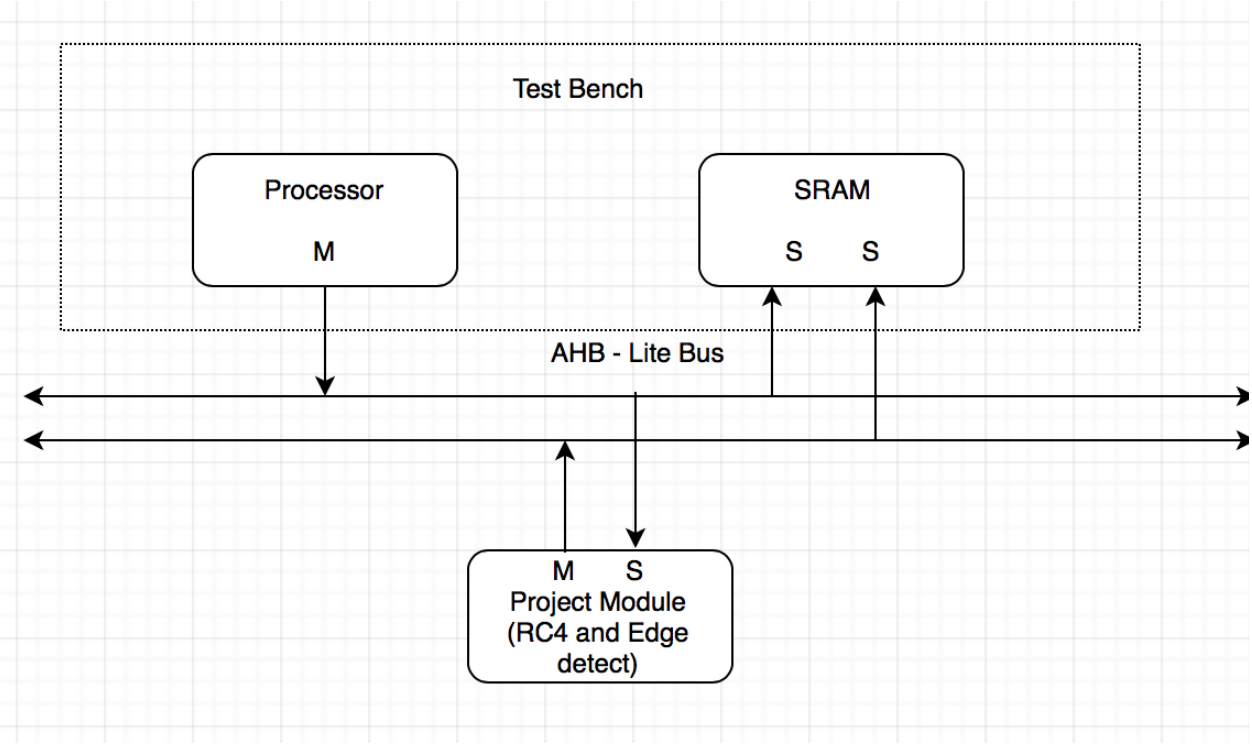


Figure 1. Example Usage of Image Edge Detector

The diagram above presents how would the Edge Detector interact typically in a bigger system as shown in Figure 1. The main processor would be used for general-purpose tasks such as running software which will also need non-volatile storage. Edge detector and the RC4 would need SRAM for quickly accessing the received images. Ideas behind what would the processor do:

1. Configure any settings needed on the Project Module.
2. Clear any images that are temporary stored in the SRAM.
3. Provide with the information on addresses of the image on the SRAM.
4. Provide the key for RC4.
5. Provide with the image height and image width.
6. Wait until the RC4 decryption and Edge detection is complete.
7. Use the Edge Detected image for further use.

2.1.2. Implemented Standard(s) and Algorithm(s)

- **Edge Detection**
  - Edge Representation
    - Identify and recognize ideal step edge
    - Differentiate between linear and nonlinear edges
    - Verify position with respect to pixels
    - Get corresponding matrix (3X3) for the pixel
  - Sobel Edge detection
    - Generate the masks for X and Y direction gradient.
    - Edge strength calculation
    - Edge detection and localization
    - Calculate threshold values
- **AHB-Lite Memory Mapped Interface**
  - 32 bit read and write data bus
  - Read and Write transfers
  - Burst transfers supported
  - Pipelined transfers
- **RC4 Cipher**
  - Generate a state Array
    - Array is of size based on image and needs
    - Generate a key for the array
    - Determine the size of the key
  - KSA function
    - Takes the state and key
    - Permutes them
    - Input - character state array
    - Input - character key
    - Input size
  - PRGA function
    - Generates keystream
    - Each byte of state is permuted
    - 4 byte of output for one call
  - XOR the PRGA and KSA function outputs

2.2 Design Pinout

Table 1: Miscellaneous Pinout Table

Signal Role	Direction (IN/OUT/ birdirc)	Number of bits	Description
gnd	GND		Ground Pin
pwr	PWR		Power Pin
HCLK	IN	1	Our System clock Note: Also used as AHB-Lite bus clock
HRESETn	IN	1	System asynchronous reset (Active Low)

Table2: AHB Memory Mapped Slave Interface Pins

Signal Role	Direction	Number of bits	Description
HADDR	Master → Slave	32	Slave namespace 32-bit word address
HSIZE	Master → Slave	2	Indicates the size of the transfer, that is typically byte, halfword or word. Ex - “00” : Single byte transfer “01” : Halfword(2 byte) transfer “10” : Word(4 byte) transfer
HWDATA	Master → Slave	32	The write data bus transfers data from the master to the slaves during write operations.
HWRITE	Master → Slave	1	Indicates the transfer direction. When HIGH this signal indicates a write transfer and when LOW indicates a read transfer.
HRESP	Slave → Master	1	Tells whether the transfer was okay or there was an error.
HRDATA	Slave → Master	32	32-bit data bus for read transfers
HREADY	Slave → Master	1	When HIGH, the HREADY signal indicates to the master and all slaves, that the previous transfer is complete.
HTRANS	Master → Slave	2	Indicates the transfer type of the current transfer. This can be: • IDLE • BUSY • NONSEQUENTIAL • SEQUENTIAL.
HCLK	Global Signal		Clock for the bus interface
HRESETn	Global signal	1	Reset for the bus interface

Table 3: Local Address mapping for RC4/ Edge-detection

SRAM Address	Read / Write	Data Size (B)	Description
0x00 -> 0x02	R/W	3	Starting address for Image stored in the SRAM
0x03 -> 0x06	R/W	4	RC4 decryption key supplied by the user
0x07 -> 0x08	R/W	2	Image Width size
0x09 -> 0x10	R/W	2	Image Height Size

2.3. Operational Characteristics

2.3.1 Total Design Macro Behavior

Normal usage of the design:

1. Initialize the state vector for RC4 with the key
2. Read one byte of data
3. Decrypt the byte of data
4. Replace the byte in SRAM
5. Repeat from step 2 until the decryption is done for the entire image
6. Initialize the Gradient array
7. Read four bytes of data from the SRAM of the decrypted image into the gradient edge detection algorithm to populate the pixel matrix.
8. Repeat step 7 to populate a 3X4 matrix.
9. Calculate the gradient for the two-central pixel of the matrix.
10. Compare the gradient with respect to a threshold value.
11. Replace the edge detected gradient in the original data site for the decrypted image.
12. Repeat steps 7 to 11 for the entire image.
13. Send back two bytes of the new decrypted edge detected image.

2.3.2 Encrypted Edge detector operation

The algorithm discussed below describes the Sobel edge detection algorithm that is going to detect the edges of the image. The algorithm follows the method of calculating the gradient of each pixel position in the image. The edge detected is basically replacing the pixels of the image with the gradient found or with a threshold value while comparing them at each pixel position. The steps include the follows for the edge detection:

```
Sobel gradient and edge detection:
always_comb (initilize_edgedetect)
    I = 0;
   J = 0; //values that need to be used in gen decryption byte
   K = double (size of image in grayscale)
   L = size (K)
   For i = 0 from 1 to size(L, 1) - 2:
        For j = 0 from 1 to size(L, 2) - 2:
            Gx = gradient in x direction = ((2*L(i+2, j+1) + L(i+2,j) + L(i+2, j+2) - (2*L(i, j+1) + L(i,j) + L(i, j+2)));
            Gy = gradient in y direction = ((2*L(i+1, j+2) + L(i,j+2) + L(i+2, j+2) - (2*L(i+1, j) + L(i,j) + L(i+2, j)));

   B(i,j) = absolute value of (Gx + Gy)
   B(i,j) = square root ( Gx ^2 + Gy ^2)
End

Replace image with new characteristics of sobel gradient
```

A threshold value is also defined for the Edge. We will use threshold value of a 100 therefore B in the above algorithm will be B = max(B, Threshold).

If the sobel gradient values are lesser than the threshold value then replace them otherwise keep the gradient values.

2.3.3 RC4 Decryption

The presented algorithm describes how the design is going to decrypt an RC4 encrypted image. First the module needs to be initialized by Key-Scheduling algorithm, as described below (we assume that the key will be hardcoded into our design and we are not going to ask the user for input. The key is going to be “unbreakable”)

```
Key-Scheduling Algo:
always_comb (initilize_rc4)
    I = 0;
    J = 0; //values that need to be used in gen decryption byte
    Key_to_ini = “unbreakable”
    For i = 0 from 0 to 255:
        S[i] = i
    For i = 0 from 0 to 255:
        Key[i] = Key_to_ini[ i % 12) //12 is the length of the key
    For i = 0 from 0 to 255:
        j = (j  + S[i] + key[i]) % 256
        Temp = S[i]
        S[i] = S[j]
        S[j] = Temp
end
```

After we have initialized the RC4 module, then it generates a byte value which is going be used to be xored with the input byte. The result would be a decrypted byte of the the image.

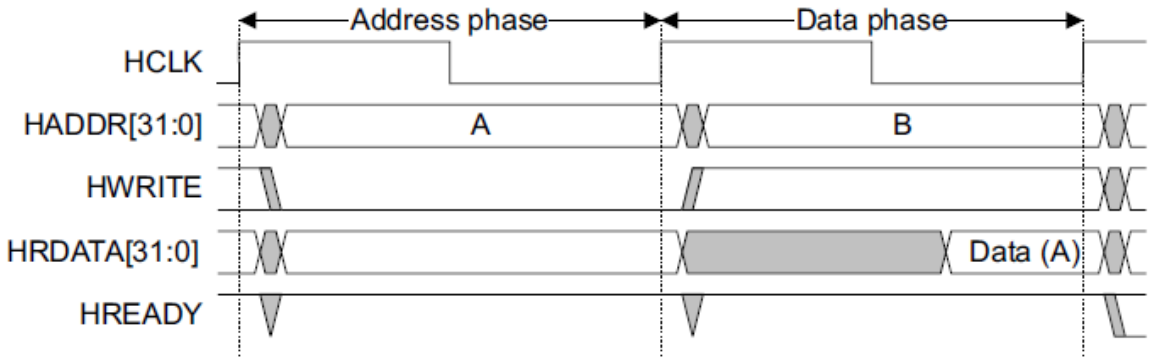
```
Generation of decryption byte:
Reg i;
Reg j;
Always_comb (gen_byte_dec)
    If gen_byte_dec == 1
        i = i + 1 % 256
        j = (j + S[i]) % 256
        Temp = S[i]
        S[i] = S[j]
        S[j] = Temp
        Output_byte = S[(S[i] + S[j] ) % 256]

Always_comb (decrypt)
    If initilize_rc4 == 0
        Initilize_rc4 = 1
    Gen_byte_dec = 1
    Decrypted_byte = input_byte ^ output_byte
    Gen_byte_dec = 0
```

2.3.4 Supported AHB-LITE SoC Bus Transactions

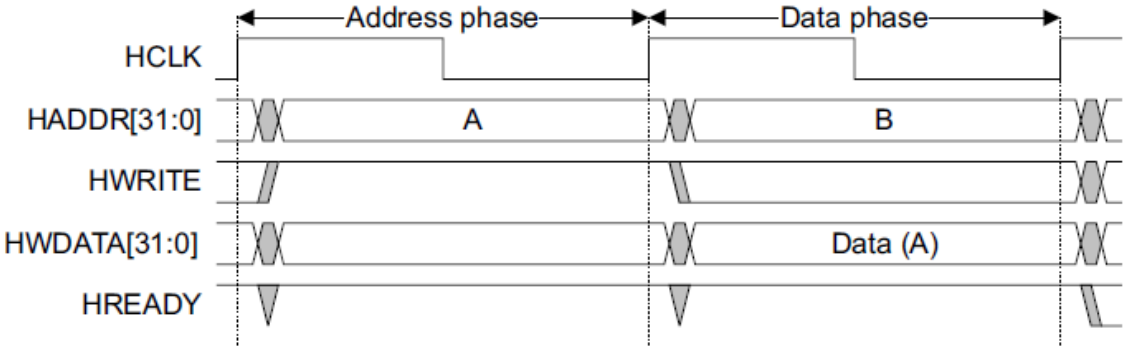
2.3.4.1. Pipelined Single Pixel Transfers

Read Transfers



The read transfer is the reading of the address and data from the slave to the master. During single word read transfers with no wait states the master drives the address and control signals onto the bus after the rising edge of HCLK. The slave then samples the address and control information on the next rising edge of HCLK. After the slave has sampled the address and control it can start to drive the appropriate HREADY response. This response is sampled by the master on the third rising edge of HCLK. The slave, at address A, uses the control signals and the address from the master, and puts the data onto the HRDATA bus for the master to read after HWRITE is asserted after a clock cycle.

Write Transfers

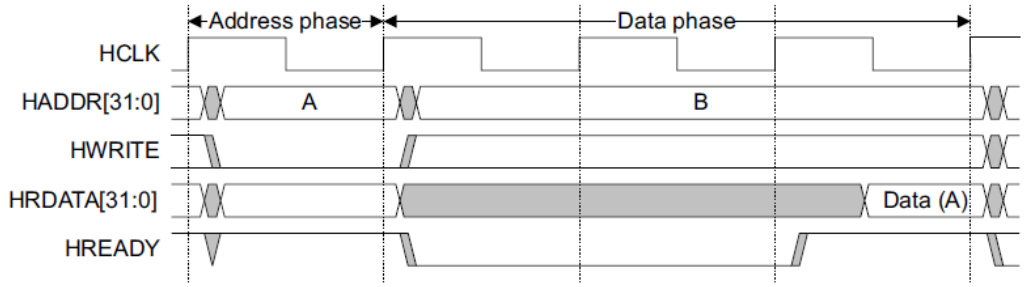


The write transfer is the writing of the address and data from the master to the salve. During single word write transfers with no wait states the master drives the address and control signals onto the bus after the rising edge of HCLK. The slave then samples the address and control information on the next rising edge of HCLK. After the slave has sampled the address and control it can start to drive the appropriate HREADY response. This response is sampled by the master

on the third rising edge of HCLK. The master, at address A, sets the control signals and the address for the slave. The master writes the Data at that that address based on the rising edge sensitivity after HWDATA is asserted after a clock cycle.

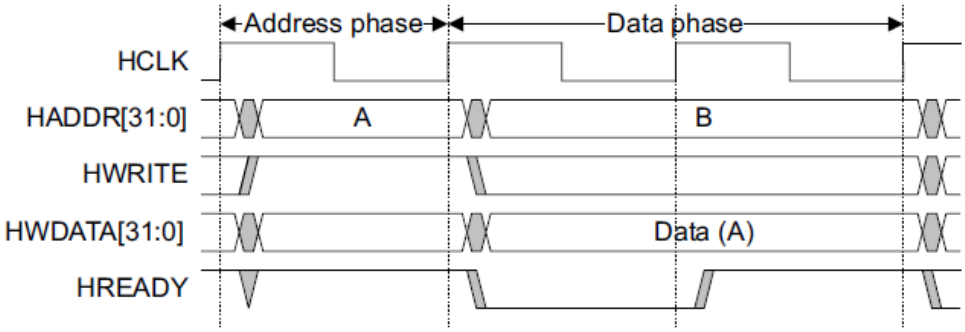
2.3.4.2. Pipeline Single Pixel Transfer with wait states

Read Transfers



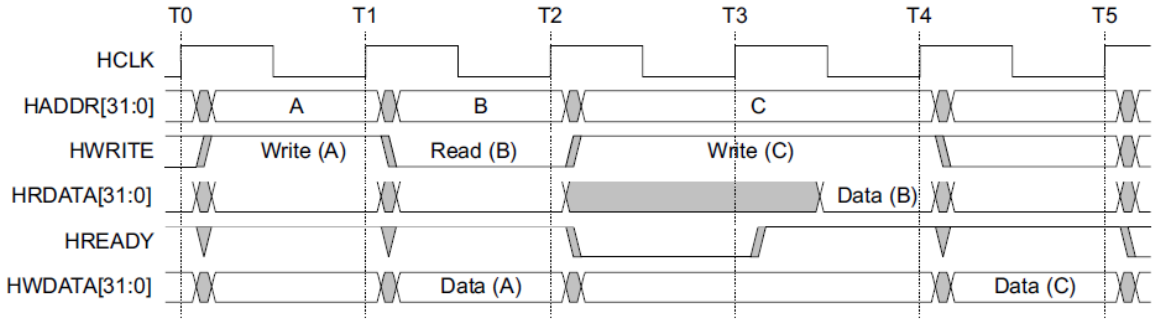
The master first drives the address and control signals onto the bus after the rising edge of the clock. The slave A then samples the address and the control information on the next rising clock edge. After the slave A has sampled the address and control, it deasserts the HREADY signal to insert wait states into the read transfer, in this case two wait states are inserted. At the same time, the master changes the address so that it can give another slave an instruction. Once the slave A is done with the transfer, it sets the HREADY signal to 1 again indicating that the transfer is complete. For read transfers the slave does not have to provide valid data until the transfer is about to complete.

Write Transfers



The master first drives the address and control signals onto the bus after the rising edge of the clock. The slave A then samples the address and the control information on the next rising clock edge. After the slave has sampled the address and control, it de-asserts the HREADY signal to insert wait states into the write transfer; in this case only one wait state is inserted. At the same time, the master changes the address to provide read or write instruction to other slave. Once the slave is done with the transfer, it sets the HREADY signal to 1 again indicating that the transfer is complete. For write operations the master holds the data stable throughout the extended cycles.

2.3.4.2. Pipeline Multiple Pixel Transfers



The master drives the address and the control signals on the bus. The slave samples the address and at the upcoming clock cycle asserts HREADY for the proper write of the data from the corresponding address. On the next clock cycle the master then reads the data from the slave. These transactions are overlapped thus finishing in 3 clock cycles. On the next write cycle the master sets the condition for writing address of C but in this transaction the previous read cycle inserted a wait state in the data phase so that it could complete the reading properly. After the wait is completed, the data is written while asserting the HREADY signal. These transactions indicate the multiple word transfer with pipelined AHB Lite Bus interface.

2.4 Requirements

The reason behind implementing decryption and edge detection on an ASIC, rather than through software, was to make the process faster. Hence the special requirement that we decided to focus on was computational latency. To ensure that the algorithm runs efficiently, we need to focus on the design organization, and its effect on the critical path.

One method by which we plan on optimizing for time is to make use of a fifo buffer for both reading and writing from the edge detection module. We will implement a burst read and burst write for the edge detection module to allow for higher data throughput.

The input format accepted by our chip will be smaller or equal to 640x480 grayscale Bitmap (.bmp) images. Our target is to achieve a clock speed of 28.5 MHz. Our target area should be under 5mm x 5mm. We have a sample rate of 10Mbit / second.

3. Design Implementation

3.1 Design Architecture

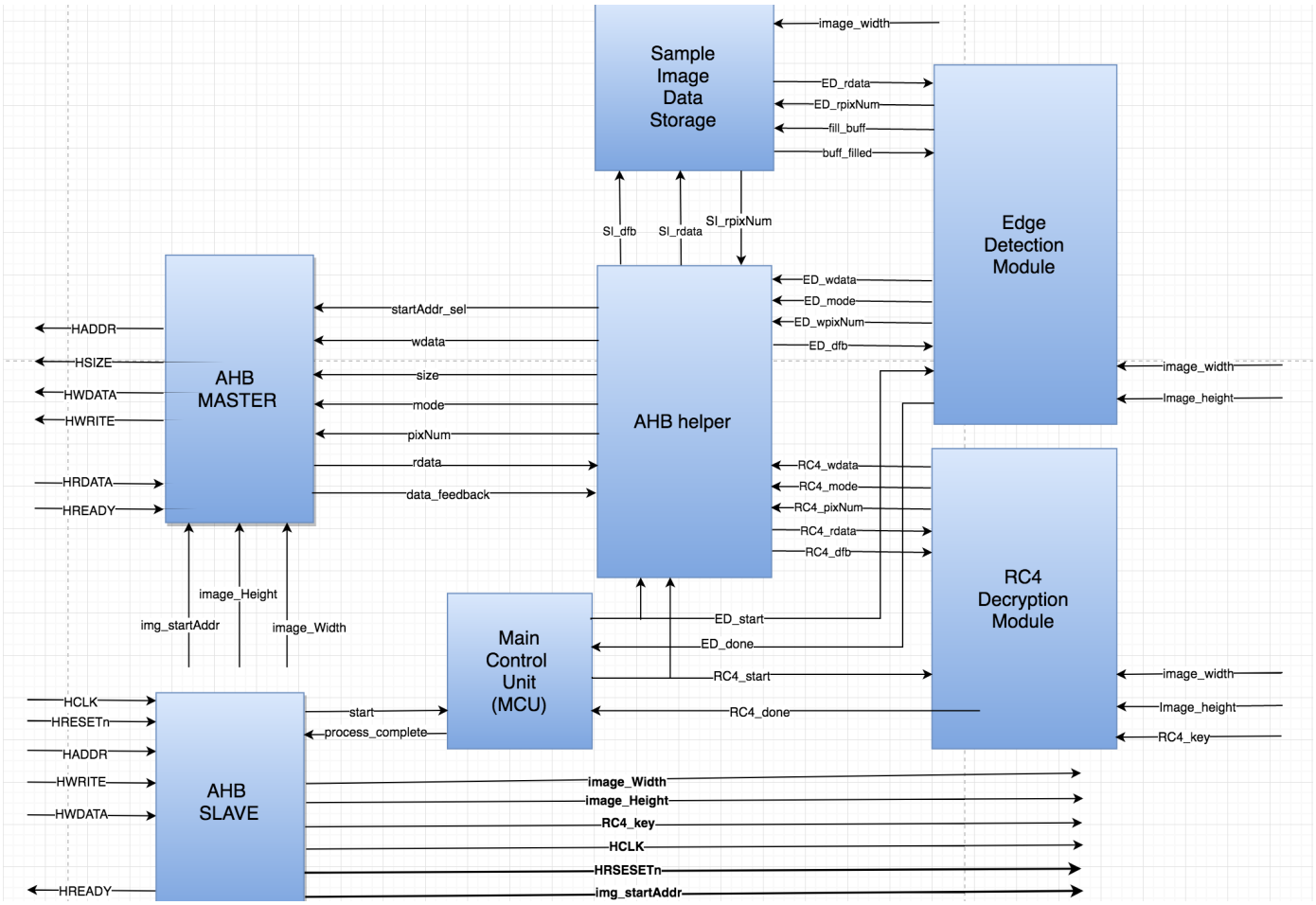


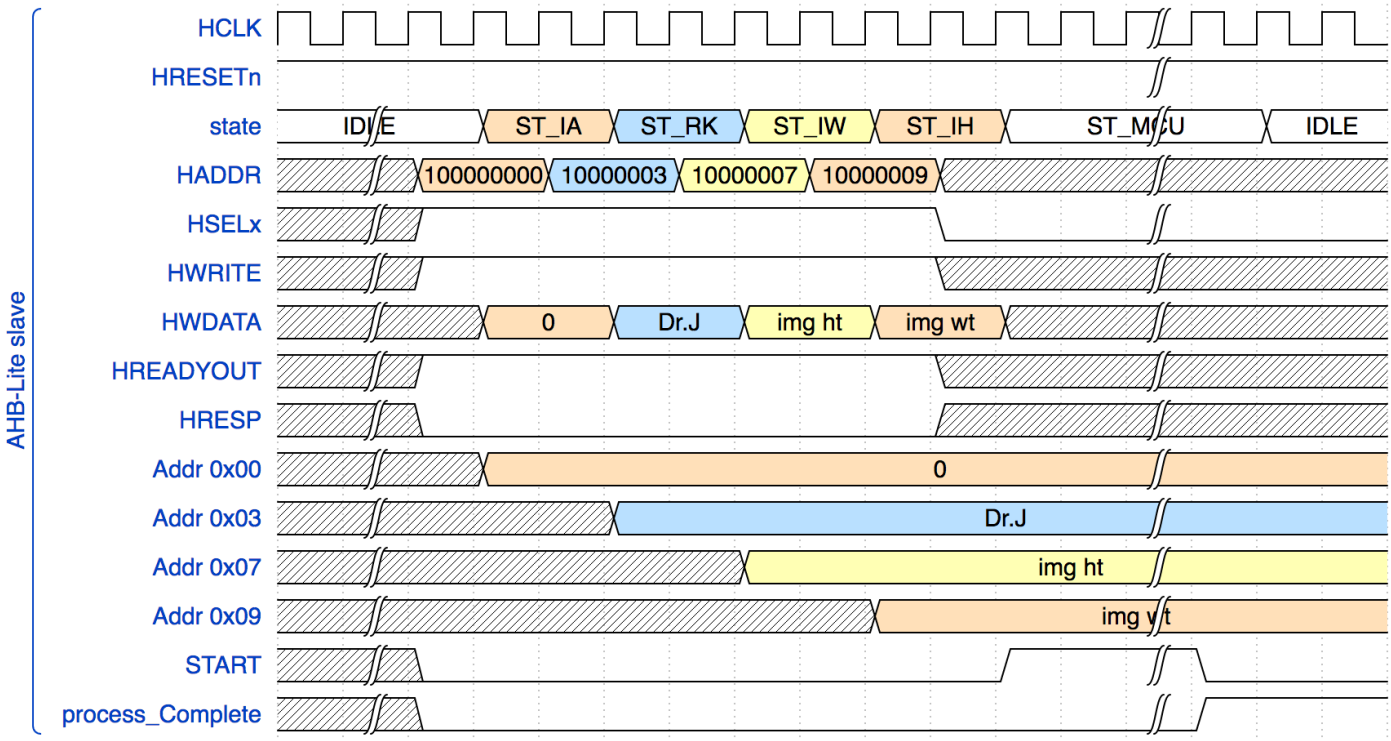
Figure 2: RC4 and Edge-Detection Architecture Diagram

The intended implementation architecture is depicted above in Figure 2. As seen in the diagram, there will be three modules that will handle the implementation of the AHB Lite interface. These are the AHB master, the AHB slave and the AHB Helper. They will serve as how the inputted encrypted image will be accepted, and the processed image will be outputted. It will also serve as how the different modules within the SoC will access different locations in memory. The Main Control Unit will oversee deciding when to activate the various other modules inside our project module. The MCU will be an FSM with multiple states, each containing instructions to the various modules. When an encrypted image is inputted, it will be received through the AHB Slave Interface. On receiving the image, the MCU will send a signal to the RC4 module, instructing it to begin its process. Once the entire image file has been read, and decrypted, the image is sent through the AHB Master using the AHB Helper which helps in storing the image in

'Image Storage'. Then the MCU instructs the 'Edge Detection Module' to begin its process. To do this, the edge detection module will need to access image storage, which it will do through the Sample Image Data Storage which in turn uses the AHB Helper interface again. Once the edge detection is complete, the outputted image is stored in 'Image storage' (the old file gets overwritten) using the AHB Helper. The AHB Helper is a common interface between the RC4 module and the Edge Detection module to understand the flow of data and read and write the correct data to the memory using the AHB master.

Per Operation Timing Waveform Diagrams

1. Starting the MCU

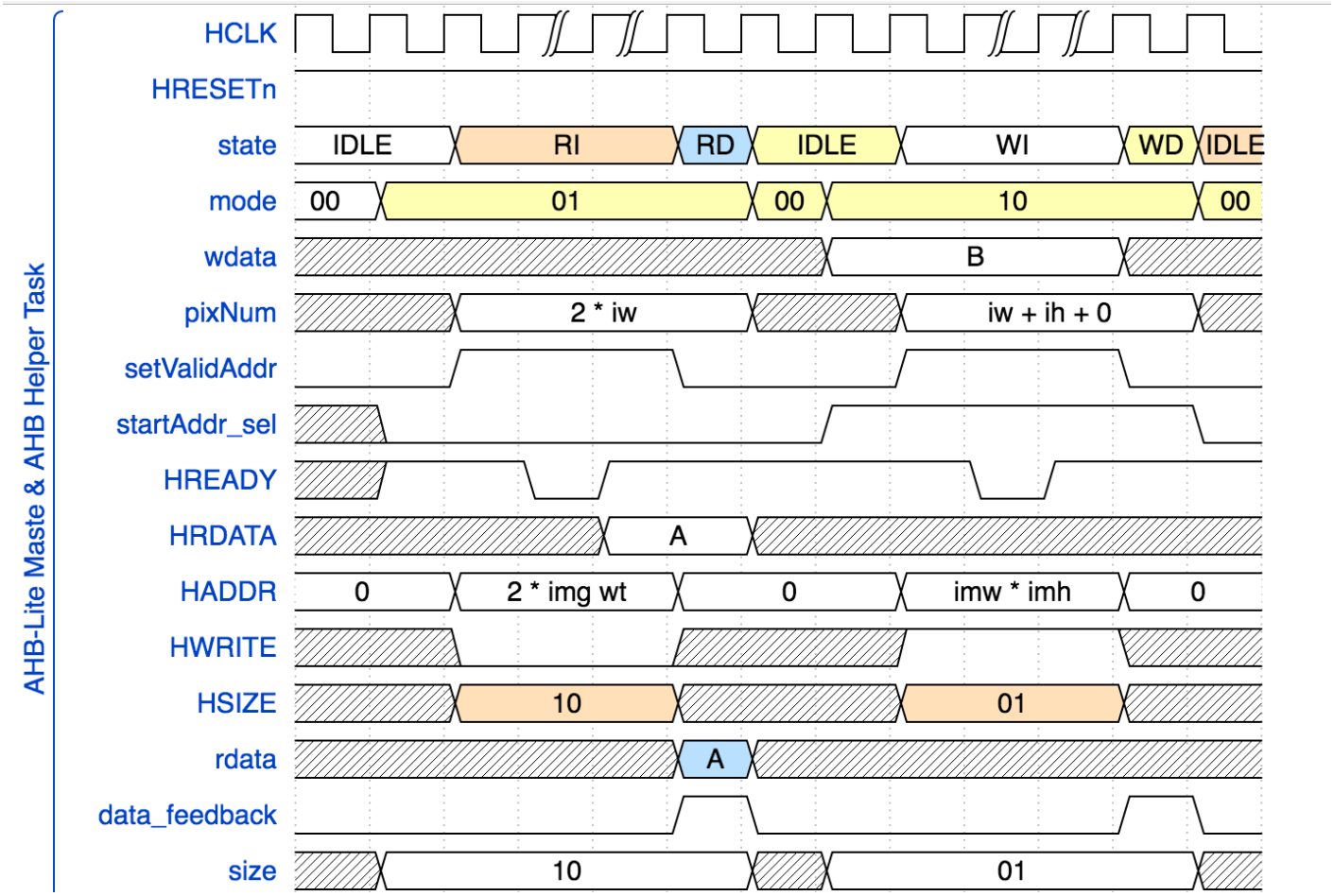


State names in the timing diagram above -
ST_IA: STORE IMG_ADDR
ST_RK: STORE RC4 Key
ST_IW: STORE IMG_WID
ST_IH: STORE IMG_HGT
ST_MCU: START MCU

The above timing diagram reflects the starting of the MCU. The slave is in the IDLE state and goes into ST_IA state only when the CPU provides the correct HSELx for choosing our slave module. In this state, the value for the starting address of the image is stored within the internal mapped registers(0x00). Since, the write instruction has an address

phase and a data phase, we decided to do a pipeline transfer to our AHB-Lite Slave. During the address phase, the write instruction is recognised and in the next clock cycle i.e. the data phase, the data is reflected in Addr 0x00 as 0. However, during the time it is writing in the Addr 0x00, we have changed the address for the next write instruction. So, the address phase of new instruction and the data phase of previous write instruction take place in the same clock cycle. It is followed by three more write instructions, following the same pattern as the one talked above. After ST_IH state, the data for the image height is written, the slave goes into ST_MCU state which sets the START signal to 1. This signal is seen by the MCU and it starts the entire process of decryption and edge-Detection. This task is important since without receiving a START signal, our design won't even start the RC4 decryption or the Edge-Detection.

2. Sending the right signals/data for Read or Write instruction to AHB-Lite Master

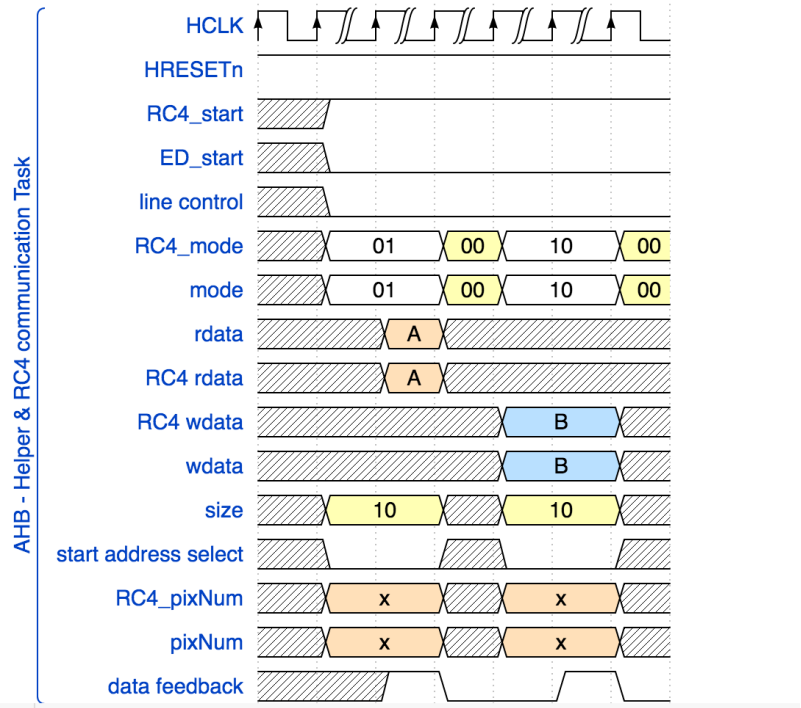


State names in the transition diagram:
RI: Read Instruction
RD: Read Done
WI: Write Instruction

WD: Write Done

The above timing diagram reflects the communication between the AHB Helper and the AHB Master. The master is in the IDLE state and goes into RI state only when the mode gets changed to 01. Here mode = 01 means that a read request is sent to AHB-Lite master. This read request can come from either RC4 or the Edge-Detection module. During the first clock cycle, only the instruction request for 4 bytes is sent. The address is sent in the next clock cycle which marks the address phase of the read instruction. The data regarding the transfer are sent during the next clock cycle since this will be the data phase of the instruction. The setValidAddr line decides when the correct HADDR is set. Once the read instruction is done, a mode = 00 is sent by the AHB-helper to make sure that AHB-master does not perform the read or write instruction over the same SRAM address again. It is basically put into IDLE state by setting the HADDR to 0 which makes sure the HSELx = 0 which is not a valid address of the SRAM as a slave. Here the startAddr_sel line comes from AHB-Helper that decides whether we need the offset of image size. This will be used only for writing the edge-detected data to get the new image since we can't overwrite the old image. Also, a data feedback is sent back to AHB-Helper so that the module which requested the read/Write request can know that the request was completed, and the AHB-master is ready to accept a new read or write request.

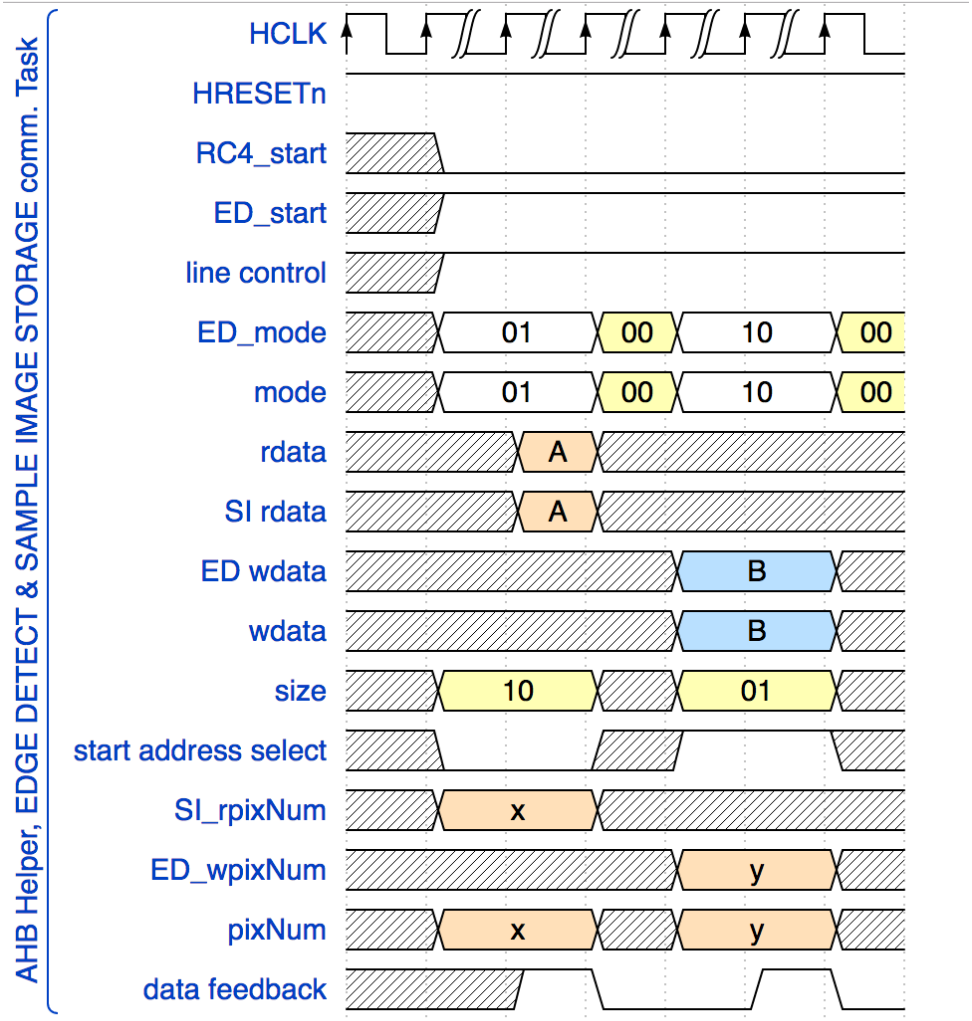
3. RC4 communicating with AHB-Helper



The above figure shows the communication between the RC4 module and the AHB-helper to set the right signals and send them to AHB-Lite Master for read or write instructions. Since the AHB-Helper is just a combinational block, we see that the change in signals from RC4 are reflected during the same clock cycle in the AHB-Helper module. The signals like RC4_mode, RC4 wdata, size, RC4_pixNum are the information that RC4 decides based on its current state.

These signals and data are send to AHB-Helper module, which then relays this information to AHB-Master. Here the start address select line goes to 0 for both read and write instructions because this line is used to give an offset of image size. But since the decrypted image is written over the encrypted image, we don't need to give that offset. However, that signal will be used by the Edge-Detection module later on. One important thing to note here is that once the read request from RC4 is done, the data from AHB-Master is put onto the rdata line and sent back to RC4 rdata. Also, a data feedback is sent back to RC4 so that RC4 can know that the request was done, and the AHB-master is ready to accept a new read or write request.

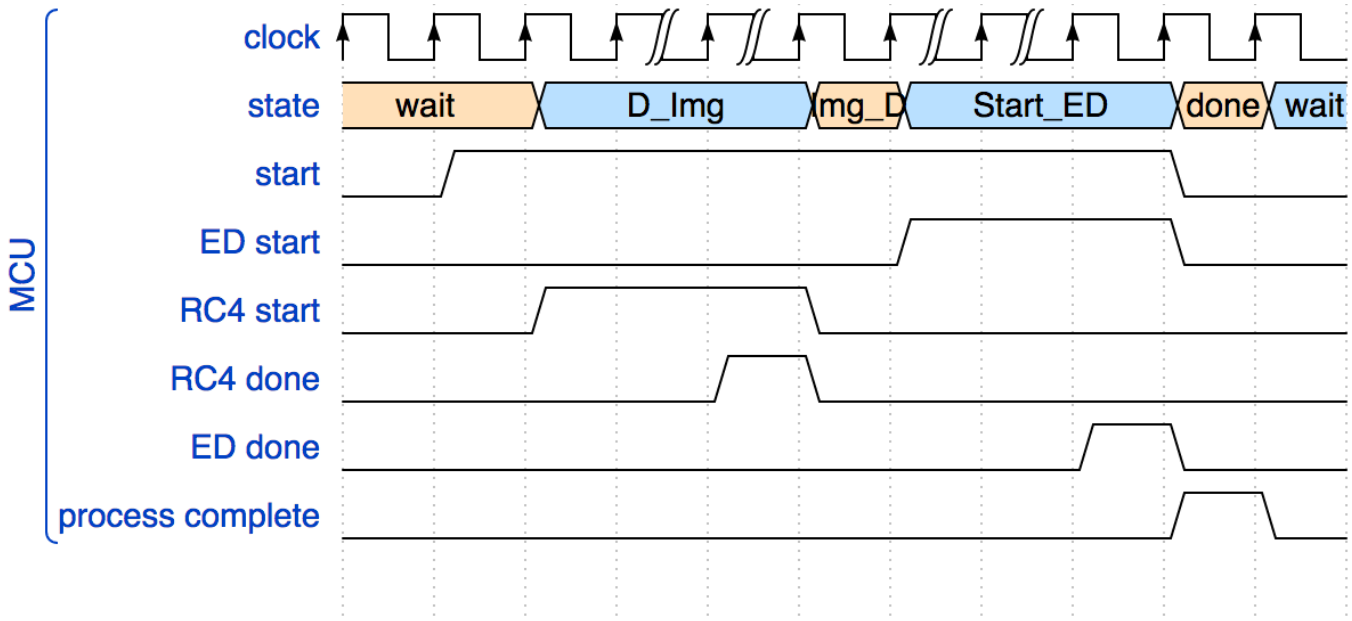
4. AHB Helper, edge Detection and Sample Image Storage Communication



The above image shows the timing waveform diagram for the inter-module communication between the AHB-Helper, Edge-Detection and the Sample Image Storage modules. The mode is set by the Edge-Detection module which reflects what kind of instruction it needs to perform. First the Edge-Detection module asks the Sample Image storage module to fill the buffer of 12 pixels worth of data. This data is used to perform edge-detection for 2 pixels. It is set by using line

control which is set by ED_Start signal. So now the AHB-helper knows that the read/write instruction will only be performed by the Edge-Detection module. Now, the sample Image Storage module sends the pixel number for read through SI_rpixNum and EDge-Detection Module sends the pixel number for write through ED_wpixNum. For the initial instruction, mode is set to 01, and the data from SRAM is put on rdata by the AHb-Lite master and sent to AHB-helper. This data is passed onto Sample Image storage module through the line SI rdata. For the write instruction, the mode is set to 10 and the data to be written is put onto the ED wdata by the Edge-detection module. AHb-helper puts this data onto wdata and sends the data to AHB-master to write it to SRAM. Note that start address select is set to 1 only fro write instruction since writing the edge-detected data, we need to write it to new image location. We cannot write it over the decrypted image since it would corrupt the image and would result in the incorrect output edge-detected image. A data feedback is sent back to AHB-Helper so that the Edge-detected or the Sample image storage module which requested the read/write request can know that the request was completed, and the AHB-helper can be used to send a new read or write request.

5. MCU - Ending of Operations



D_Img: Decrypt image state
Img_D: Image decryption done state
Start_ED: Start Edge-Detection

As described earlier, the MCU is responsible for all order of operations. On completion of all tasks, the MCU is responsible for sending out a signal, ‘process complete’, to both the AHB slave and the external CPU, indicating that the inputted image has been decrypted and edge detected, and is ready for new data.

When the MCU receives ‘ED_done’, the MCU proceeds to its next state ‘done’, in which the MCU asserts process complete. This allows AHB slave to reset to its wait state. One clock cycle later, the MCU itself resets to its wait state where process complete is set to 0.

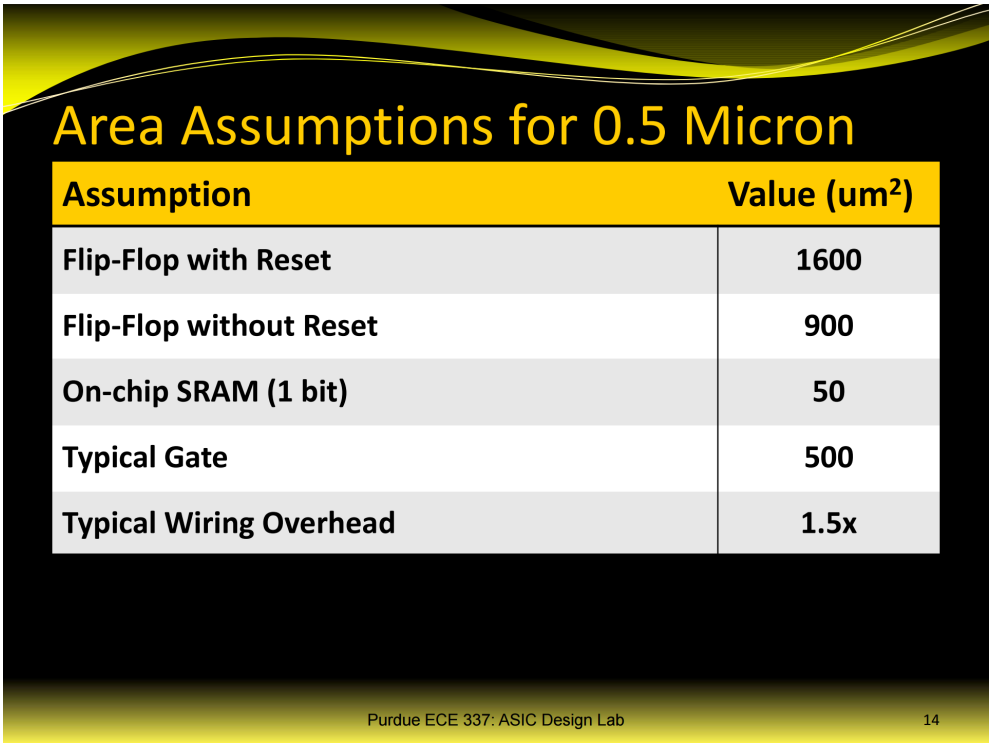
Hence to summarize, the assertion of process complete implies that the encrypted image has been received, decrypted, edge detected, and saved in the SRAM.

Architectural Level Area Estimation Table and Discussion:

Core Area Calculations				
Name of Block	Category	Gate/FF Count	Area (um2)	Comments
AHB-Lite Slave Input Logic	Combinational	114	85,500	
AHB-Lite State Registers w/ Reset	Reg. w/ Reset	3	7,200	
AHB-Lite Slave Output Logic	Combinational	10	7,500	
AHB-Lite Memory Mapping	Reg. w/ Reset	80	192,000	10 registers of 8-bits each
MCU Input Logic	Combinational	15	11,250	
MCU State Registers w/ Reset	Reg. w/ Reset	3	7,200	
MCU Output Logic	Combinational	6	4,500	
RC4 PSEUDO Input Logic	Combinational	60	45,000	
RC4 PSEUDO State Registers	Reg. w/ Reset	5	12,000	
RC4 PSEUDO Output Logic	Combinational	84	63,000	
RC4 PSEUDO Combinational Logic	Combinational	2108	1,581,000	
RC4 PSEUDO (3 2:1 Mux)	Combinational	24	18,000	
RC4 PSEUDO 2072 regs	Reg. w/ Reset	2072	4,972,800	Regs for State Array(255*8) + LocSafe + OutputToXOR + Temp + CounterJ
RC4 PSEUDO 8-bit Flex Counter	Reg. w/ Reset	8	19,200	
RC4 PSEUDO 255:1 Mux	Combinational	1020	765,000	
RC4 PSEUDO 1:255 Demux	Combinational	765	573,750	
RC4 Core Input Logic	Combinational	160	120,000	
RC4 Core State Registers	Reg. w/ Reset	4	9,600	
RC4 Core Output Logic	Combinational	150	112,500	
RC4 Core Flex Counter (20-bit)	Reg. w/ Reset	20	48,000	
RC4 (2 2:1 Mux)	Combinational	8	6,000	
RC4 Core regs	Reg. w/ Reset	62	148,800	
RC4 Combinational Logic	Combinational	1558	1,168,500	
AHB-Lite Master Input Logic	Combinational	15	11,250	
AHB-Lite Master State Registers w/Reset	Reg. w/ Reset	3	7,200	
AHB-Lite Master Output Logic	Combinational	6	4,500	
AHB-Lite Master In-Module Muxes (20 2:1)	Combinational	80	60,000	
AHB-Lite Master In-module Muxes (32 2:1)	Combinational	128	64,000	
AHB-Lite Master (2 20-bit Adders)	Combinational	200	150,000	(Worst Case 20-bit Riple Carry Adders) but we would implement carry look ahead adder
AHB-Lite Master (12x12 Bit Multiplier)	Combinational	144	108,000	(if we use binary array multiplier)
AHB Helper Muxes (58 2:1)	Combinational	232	116,000	
AHB Helper DeMuxes (33 1:2)	Combinational	132	66,000	
AHB Helper (2 2 bit comparator)	Combinational	20	15,000	
Edge-Detection Input Logic	Combinational	18	13,500	
Edge detection 24 8 bit adders	Combinational	1440	720,000	
Edge Detection 2 8 comparator	Combinational	60	45,000	
Edge-Detection State Registers	Reg. w/ Reset	3	7,200	
Edge-Detection Output Logic	Combinational	14	10,500	
Edge-Detection Counters	Reg. w/ Reset	2	4,800	
Edge-Detection (16x16 Bit Multiplier)	Combinational	256	192,000	(if we use binary array multiplier)
Edge-Detection (20 bit comparator)	Combinational	170	127,500	(about 30 gates for 1 4-bit comparator) So we use 5 4 bit comparators and add 20 registers for precautio
Sample Storage (25 3:1 Mux)	Combinational	200	150,000	
Sample Storage (1 20 bit adder)	Combinational	100	75,000	(Worst Case 20-bit Riple Carry Adders) but we would implement carry look ahead adder
Sample Storage (96 3:1 Demuxes)	Combinational	768	576,000	
Sample Storage State Registers w/ Reset	Reg. w/ Reset	3	7,200	
Sample Storage Input Logic	Combinational	12	9,000	
Sample Storage Output Logic	Combinational	8	6,000	
Sample Storage Registers w/Reset	Reg. w/ Reset	96	230,400	
Total Core Area			12,754,350	
Chip Area Calculations (units in um or um2)				
Number of I/O Pads:		-		
I/O Pad Dimensions:	-	by	-	
I/O Based Padframe Dimensions:	-	by	-	
Core Dimensions	-	by	-	
Core Based Padframe Dimensions:	-	by	-	
Final Padframe Dimensions:	-	by	-	
Final Chip Area (1.5 x):		19,131,525		

Table: *Area Estimation of System level architecture*

The table above describes our Area estimation for the architectural level diagram. It includes the name of the modules in our architecture followed by the type of logic used for that section of the module. Further we elaborate on the number of gates for each type of logic and finally the translation of the gates into area in micrometers squared. This area is based on the table below.



Area Assumptions for 0.5 Micron

Assumption	Value (um^2)
Flip-Flop with Reset	1600
Flip-Flop without Reset	900
On-chip SRAM (1 bit)	50
Typical Gate	500
Typical Wiring Overhead	1.5x

Purdue ECE 337: ASIC Design Lab 14

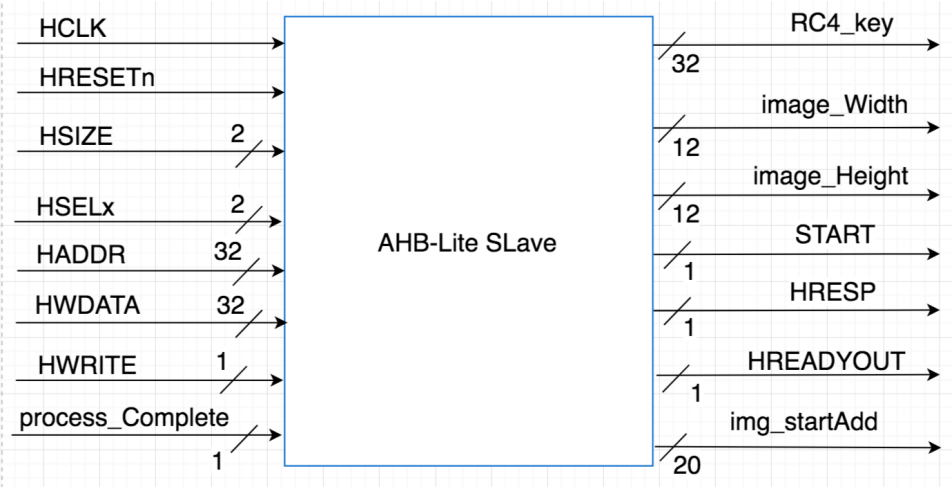
For each block, we found the number of gates based on the calculation of the number of bits of input of a sub block of that module. This was based on the block diagrams and RTL's of each block. Subsequently to the calculation we took in account the number of bits of output of the same sub blocks based on their Block diagram and RTL's. With these two factors known now for our calculation, the area is calculated by multiplying the number of output bits to one less than the number of input bits. This calculation holds true for finding the number of gates from input to output as it is based on all possible combinations of output bits and input bits and all these combinations require gates for operation. This calculation held true for all the possible blocks and their categories of registers and combinational block in the sub module. For other parts of the module that required a fixed number of registers or were combinational logic separate from the input and output logic of the module have their own gate count written separately. As we have an SoC we do not require including the area for the I/O pads. There is also an inclusion of adders, comparators, multiplexers and demultiplexers with their separate gate count and area estimation. With all these area calculations done, the core chip area is simply the sum of all these areas. Our chip area is estimated to be **12,754,350 micrometer squared**. This area is also multiplied with a 1.5 wiring overhead assumption and this brings the overall chip area to **19,131,525 micrometer squared**.

3.2 Functional Block Diagrams

AHB-Lite Slave:

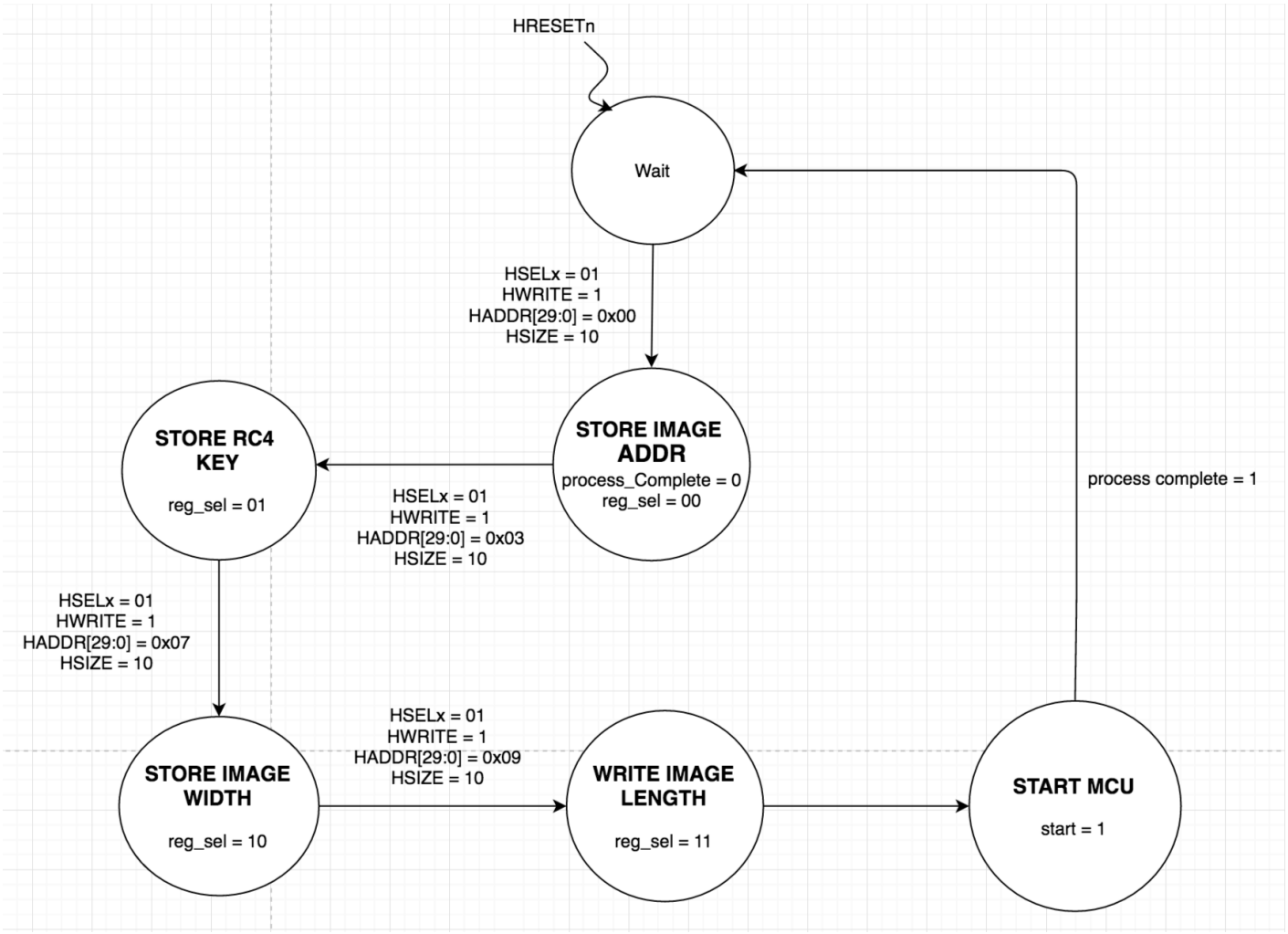
The AHB-Lite Slave is used to communicate with the CPU to get the correct data regarding the image, the key for the RC4 decryption and the dimensions of the image - the image height and the image width. All of this data is stored within a memory mapped array within the slave and this data is made available to all the other modules which saves the need to store the data in each individual module. It is important to note that it is the AHB-Lite slave that decides when the MCU needs to start the entire process.

- **Block Diagram –**



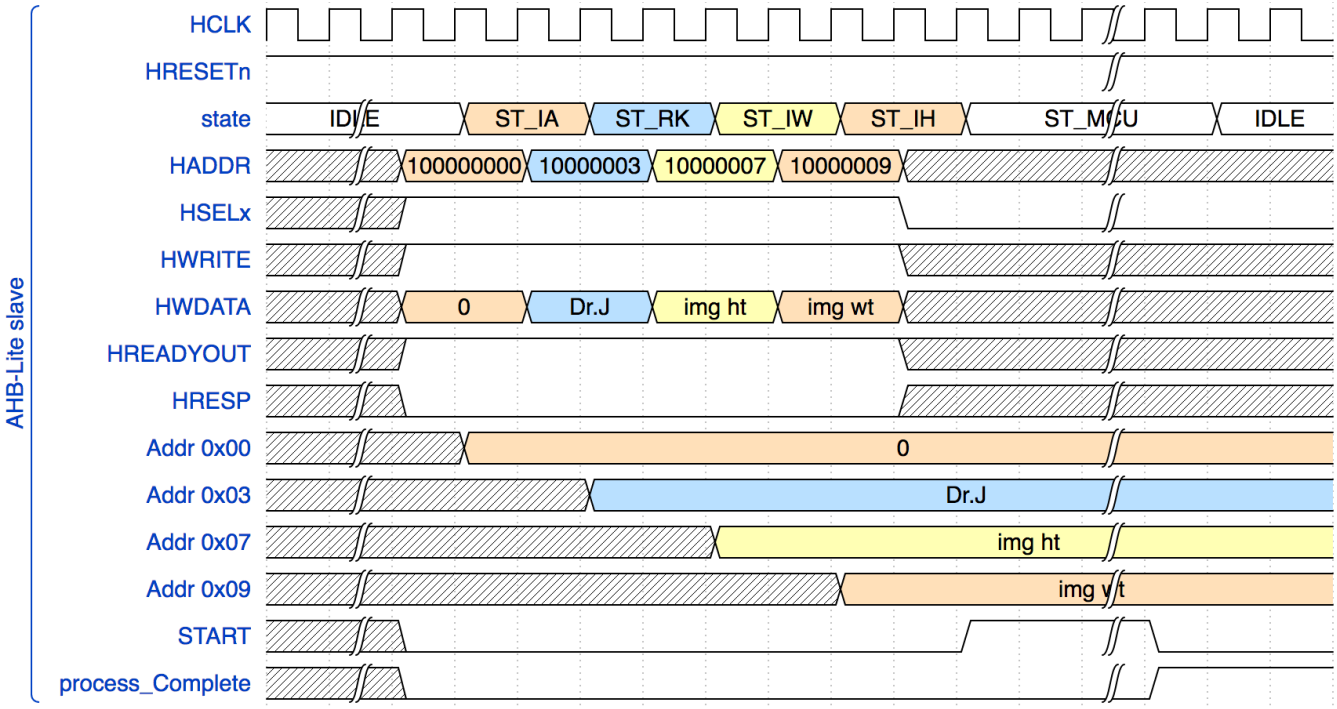
The image below above shows the block diagram for the AHB-Lite slave within our design. It is just a single block with address and control signals coming from the CPU(Master) and the process_Complete signal coming from the MCU that tells that the RC4 decryption and the Edge-Detection has been done. The output signals are the response signals to the CPU and a START signal to the MCU that marks the start of the RC4 decryption. The reg_Sel signal selects in which memory mapped register the data will be written from the HWDATA line from CPU.

- **State Transition Diagram -**



The image above shows the state transition diagram for the FSM used in AHB-Lite Slave. Initially it is in the IDLE state. Once it receives the HSELx from the CPU to be the correct address of the slave, our AHB-Lite slave activates. Then, the state transitions for the write transaction. It is important to note that if the CPU tries to give a read instruction, the FSM won't change state since there is nothing to read from our slave. One by one, the COU will give four write instructions, each for starting address of the image in SRAM, the RC4 decryption key, image width and the image height respectively. All this data is written in the internal registers by setting the reg_Sel line in store states in the state-transition diagram. Once all the data is acquired, START signal is asserted to tell MCU to start its process. The FSM then waits for the process_Complete signal to be asserted by the MCU telling that the entire process of decryption and edge-detection is complete.

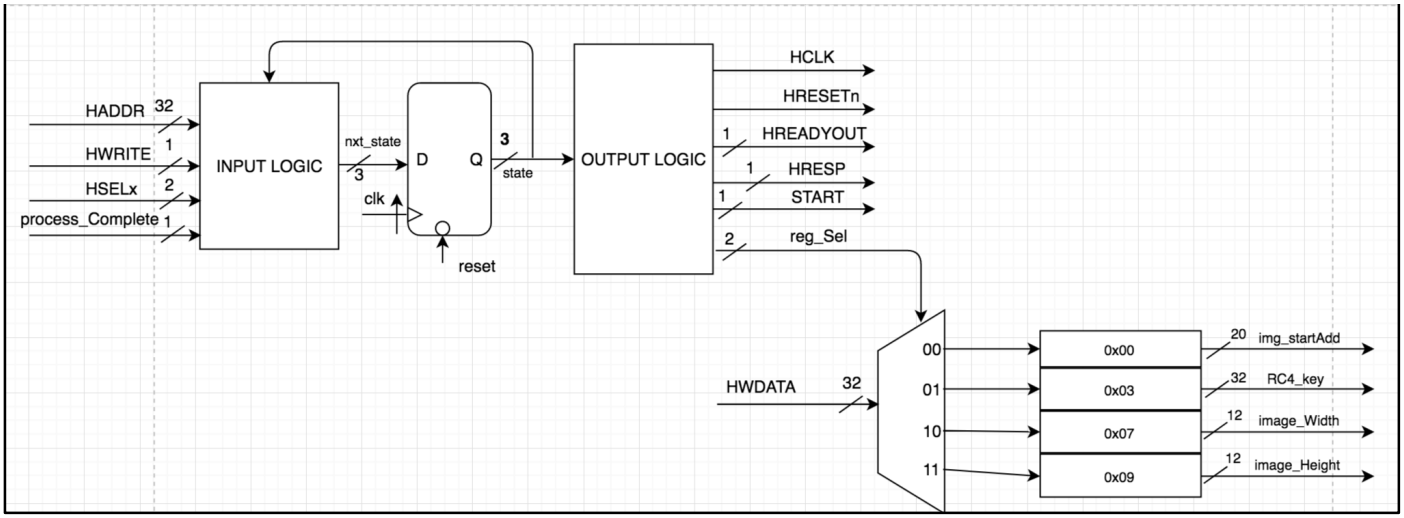
- **Waveform -**



ST_IA: STORE IMG_ADDR
ST_RK: STORE RC4 Key
ST_IW: STORE IMG_WID
ST_IH: STORE IMG_HGT
ST_MCU: START MCU

The above image shows the timing waveform for the AHB-Lite Slave. The read or write instruction consist of two phases, one address phase and one data phase. During the address phase, the CPU sends the address of right register in the HADDR signal, selects the slave, and sets the HWRITE signal to 1 to do a write instruction. During the next clock cycle, which is the data phase, the CPU broadcasts the data that needs to be written in the slave's internal register on the HWDATA line and the slave selects the reg_sel line to out the data into the right register. It is wise to note here that although we haven't made any wait states in the timing diagram, it is totally possible to introduce wait states in between instructions. However, since we are optimizing time, all these write instructions could be completed without states as well which we have showed here. Once the ST_IH state is completed, the next state it goes into is ST_MCU. In this state, a start signal is asserted for the MCU to start operating on the image stored in the SRAM. After, sometime (we don't know the exact time yet), MCU will send an asserted process_Complete signal which will make the FSM transition to its IDLE state again.

- **RTL Diagram -**



The image above depicts the RTL Diagram of the AHB-Lite slave module. It comprises of a FSM.
The input logic decides which state it is going through based on the incoming signals, the register stores the which state it is in, and the output logic sets the signal based on the state the FSM is in. The most important aspect is the reg_Sel line which decides the data flow within the memory mapped registers through a De-multiplexer. Also, the HCLK and HRESETn are sent to other modules as well so that they have the same clocking speeds and the FSMs in other modules are able to reset there FSMs as well.

- **Area Estimation -**

Core Area Calculations				
Name of Block	**Category**	**Gate/FF Count**	**Area (um2)**	**Comments**
AHB-Lite Slave Input Logic	Combinational	114	85,500	
AHB-Lite State Registers w/ Res	Reg. w/ Reset	3	7,200	
AHB-Lite Slave Output Logic	Combinational	10	7,500	
AHB-Lite Memory Mapping	Reg. w/ Reset	80	192,000	10 registers of 8-bits each

The above table gives the area estimation of the AHB-Slave module. The area breakdown is as follows:

1) Input logic combinational area
2) Output logic combinational area
3) State register area
4) Area for memory-mapped Registers

This makes the total area of the module come to **292,200 micrometer squared**. The area is calculated based on estimating the number of input and output bits of each block mentioned above. With finding all the possible combinations of the inputs and outputs (input bits - 1) X (output bits), we find the number of gates for each block. With

the gates of the block the area is calculated based on their category of combinational block and registers and found using known values.

- **Actual Area Comparison with original Estimation -**

Estimated Area = 292,200 um^2

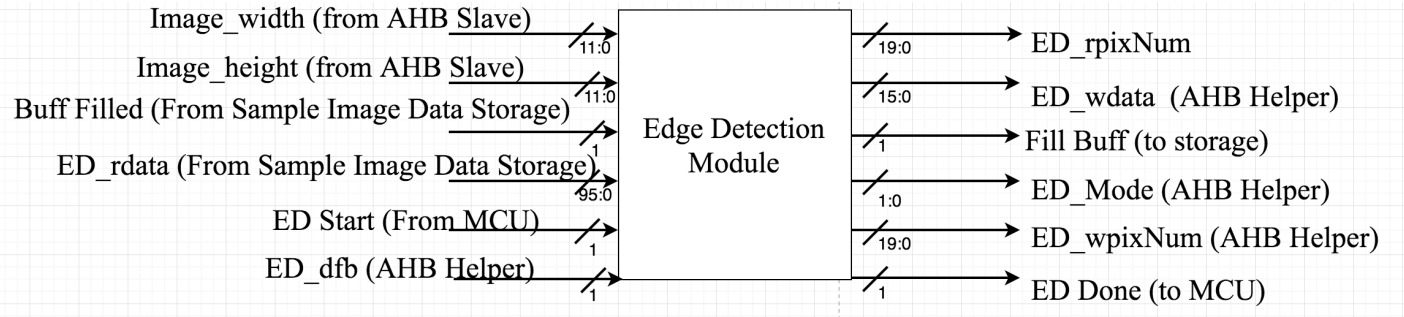
Actual Area = 200,646 um^2

The Area we estimated for the AHB-slave was indeed very close to what we actually got after synthesizing our design. The combinational area in the AHB-slave was 66,006 um^2 which is much less than our anticipated combinational area of 92,500 um^2. In addition to it, our estimation for non-combinational area in AHB-slave was 199,200 um^2 but we actually got it to be 134,640 um^2. This says that our estimation was for the worst case scenario but synthesizing the design resulted in area less than we predicted, which reduces the chip size.

Edge Detection Module:

The edge detector block performs the edge detection on the decrypted image. The main block diagram consists of a single block that has six inputs and six output signals. These correspond to the signals that it receives from the sample image data storage and the AHB helper which determine that it writes the calculated gradient correctly to the memory through the AHB Helper which in turn is communicating with the AHB master. As the MCU tells the module to start it communicates with the Sample Image Data Storage to make it fill a buffer of pixels so as to calculate the gradient of two of the central pixels of that buffer. As that buffer matrix is 3x4 this is the reason why we calculate the gradient of two central pixels at one time. Once it gets the buffer filled of the Image storage, it receives that buffer and then calculates the gradient of the pixel using Sobel Edge Detection on that matrix. Once the gradient has been calculated and compared with the threshold the module with the help of the AHB Helper writes the two bytes of pixels to the memory.

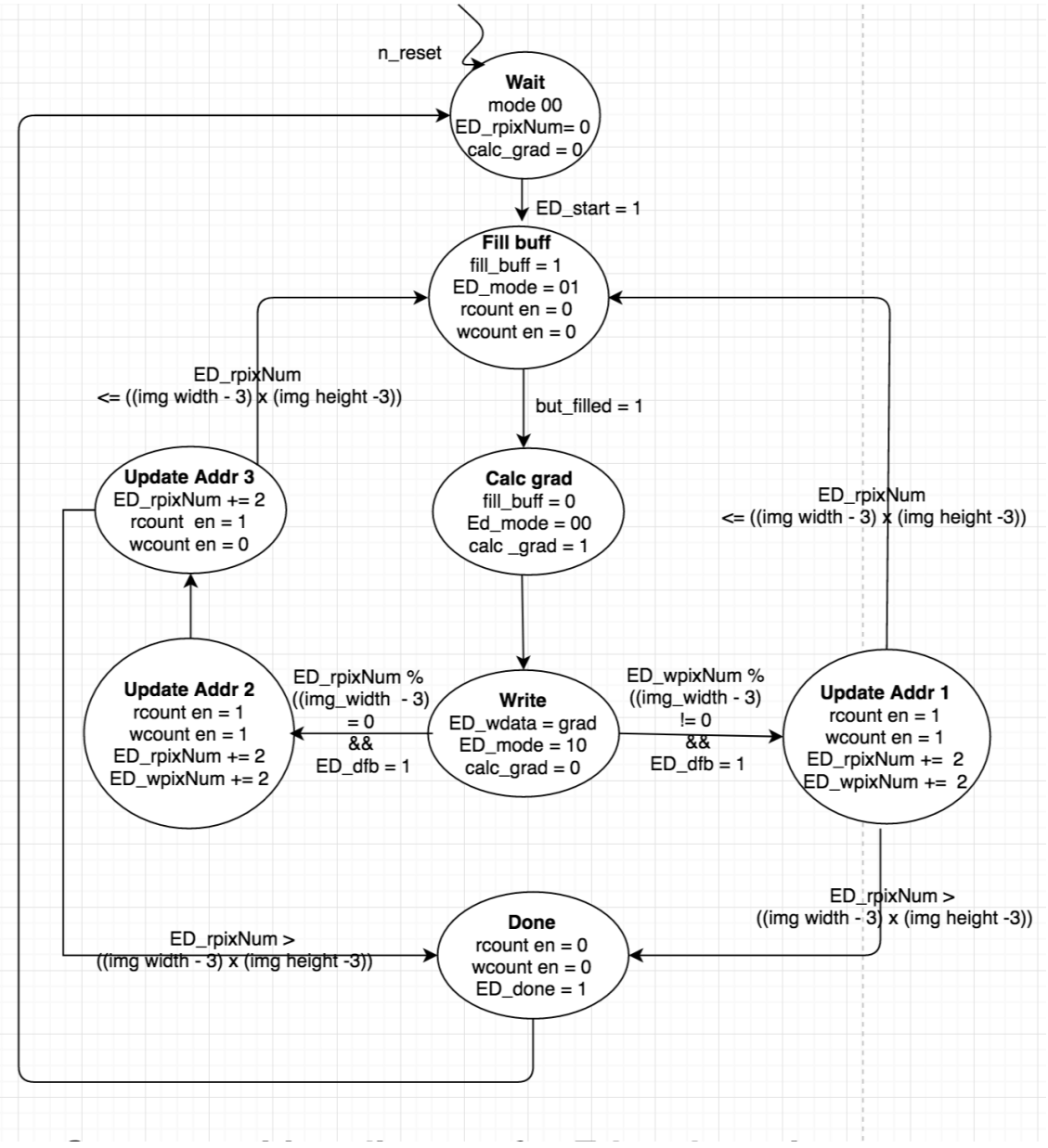
- **Block Diagram -**



The image below shows the Block diagram of the Edge-Detection Module. The signals shown below demonstrate the number of bits on the signal line, and the to or from communication with the module. The block receives the 3x4 pixel data matrix from the sample image data storage block and then using sobel edge detection, calculates the gradients in the x and y direction. This is done by masking of the pixel matrix and finding the determinant of the x and y direction

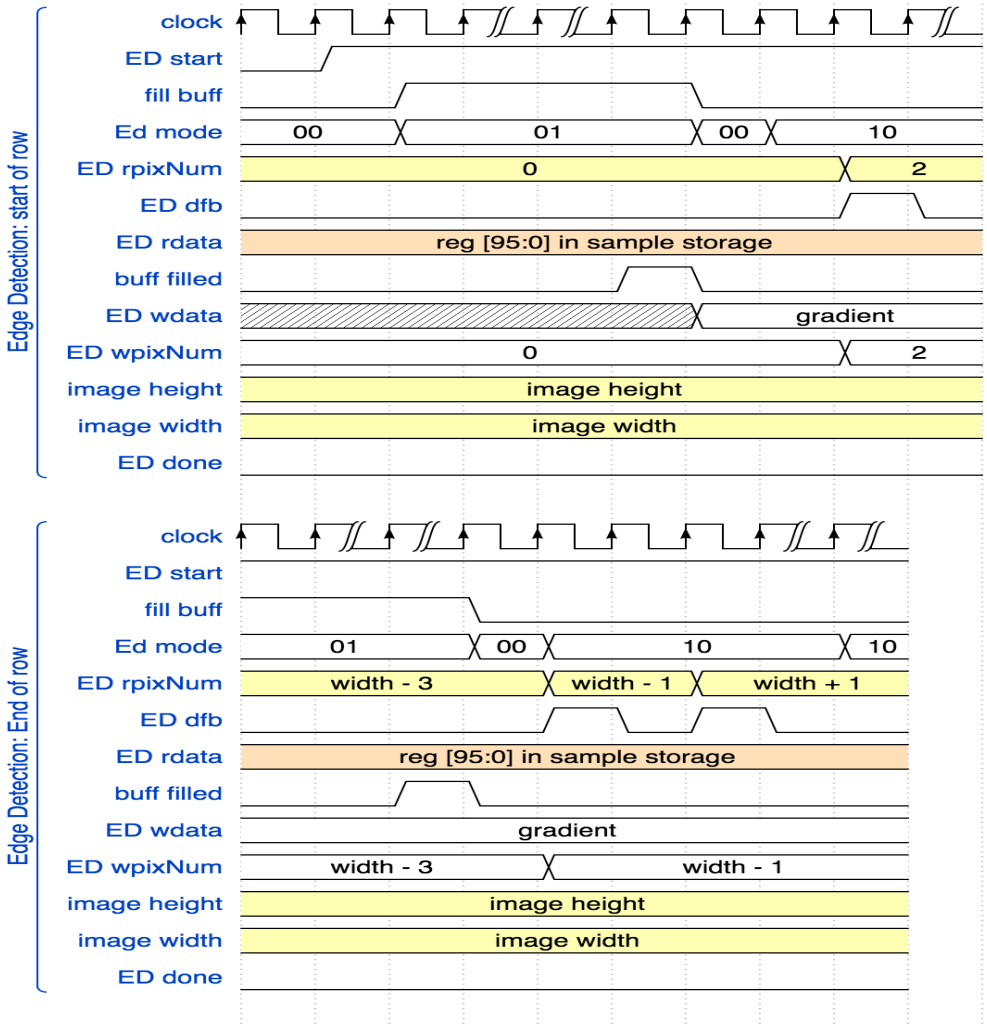
mask. The values are then squared and added and consecutively their root is taken. This value is compared with a threshold value and then if it is greater than the value, it become the edge for the central pixel of the matrix. The signals below ensure communication with the AHB helper and the Sample Image Data Storage as these are the two modules responsible for the edge detection module to get the buffer filled with the correct pixels and the writing of the correct gradient pixel to the new position.

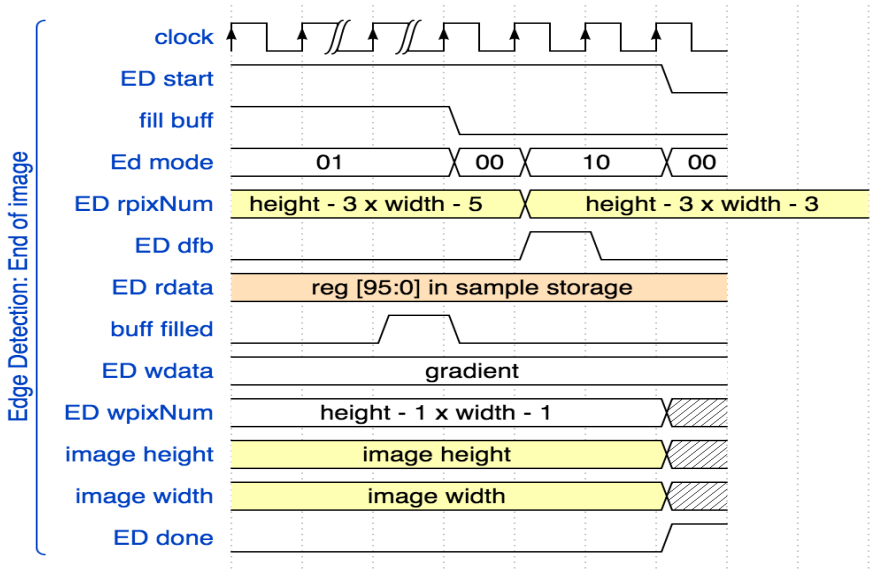
- **State Transition Diagram -**



The above diagram is the state transition diagram for the edge detection module. The module starts from the wait state. Once the MCU signals the module to start it goes to the Fill buff state where it makes the Sample image data storage fill the 3x4 matrix of pixels that must read to do the calculations. In the next state, it perform the calculation on the pixel data buffer. Once the calculation is complete it writes the pixels for the gradient and uses the AHB Helper to do so. Once the write is done before proceeding it checks whether the end of the row has been reached and if not it updates the new read address so that the matrix is positioned in such a manner that the consecutive gradients can be calculated. Once the end of the row is reached the state is to Update Addr 2 and Update Addr 3 where it jumps the buffer to the next row and updates the write to just one so that there is no empty pixel in the storage. At the final write, when the image buffer reading is happening at the last possible pixel ((image height - 3) x (image width - 3)) according to buffer size, the state Update Addr 3 checks for whether it's reached the end and if it has it moved to Done state.

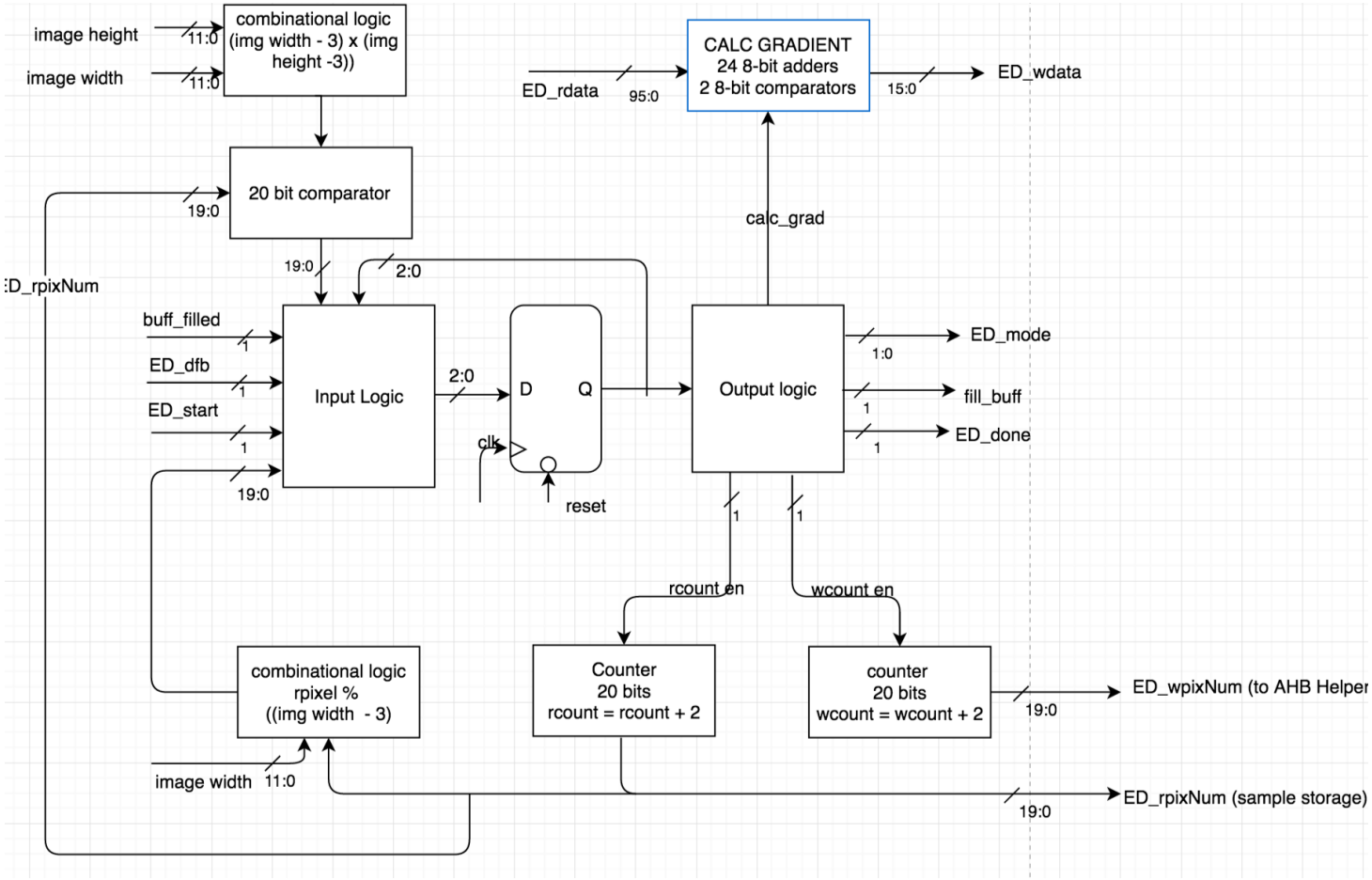
- **Waveform -**





The waveform diagram is distributed into three corresponding parts. These are when it is reading the buffer in a row, when it changes row and when it as the end of the image. The first waveform corresponds to the state transition diagram of the part when the module interacts with the image data storage to set its reading position and performs the calculation. After the calculation, the write pixel address is updated so that the the calculated pixels are written at the correct position. The same waveforms occur for a jump or change in row, but these are accompanied by to update address states with just one state updating the write address. At the end of the image when the whole image is read the same waveform proceeds the same way for one read except that it would signal the MCU that edge detection is done.

- **RTL Diagram -**



Below is given an RTL diagram for the Edge detection block. For the edge detection process the algorithm used is the sobel edge detection algorithm that uses the technique of masking a subset matrix of the image (this is a 3x3 matrix) and then calculating the gradient along the x and y direction for that matrix. The gradients are then squared and added and then rooted. This value is compared with a threshold value and then based on the comparison the edge is assigned either the gradient value or the threshold value. This calculation is performed in the Calc Gradient block that comprises of 24 bit adders and 2 8 bit adders. The RTL follows the use of a combinational block to feed in the size of the image or the last pixel position of the read by multiplying image height - 3 x image width - 3. There is also presence of 2 counters that basically monitor and update the pixel to be written or read and send this information to the data storage or the AHB helper.

- **Area Estimation -**

Core Area Calculations				
Name of Block	Category	Gate/FF Count	Area (um2)	Comments
Edge-Detection Input Logic	Combinational	18	13,500	
Edge detection 24 8 bit adders	Combinational	1440	720,000	
Edge Detection 2 8 comparator	Combinational	60	45,000	
Edge-Detection State Registers	Reg. w/ Reset	3	7,200	
Edge-Detection Output Logic	Combinational	14	10,500	
Edge-Detection Counters	Reg. w/ Reset	2	4,800	
Edge-Detection (16x16 Bit Multiplier)	Combinational	256	192,000	(if we use binary array multiplier)
Edge-Detection (20 bit comparator)	Combinational	170	127,500	(about 30 gates for 1 4-bit comparator) So we use 5 4 bit comparators and add 20 registers for precautio

The above table gives the area estimation of the Edge Detector module. The area breakdown is as follows:

5) Input logic combinational area
6) Output logic combinational area
7) State register area
8) Counters areas (for updating and keeping check on the pixel being read and written)
9) Multiplier, Adders and Comparators

This makes the total area of the module come to **1,120,500 micrometer squared**. The area is calculated based on estimating the number of input and output bits of each block mentioned above. With finding all the possible combinations of the inputs and outputs (input bits - 1) X (output bits), we find the number of gates for each block. With the gates of the block the area is calculated based on their category of combinational block and registers and found using known values.

- **Actual Area Comparison with original Estimation -**

Estimated Area = 1,120,500 um^2

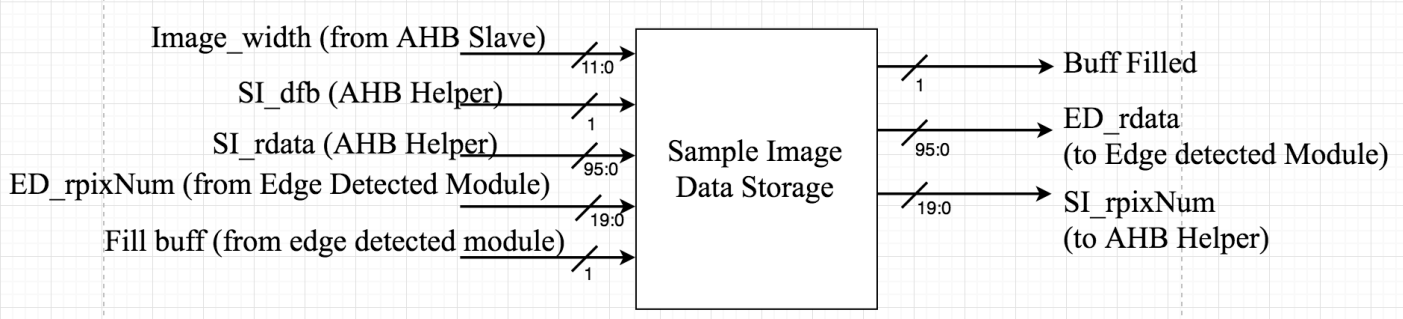
Actual Area = 2,038,743 um^2

The actual area of the Edge-Detection module was found to be 2,038,743 um^2 which is about 22.6% of the area of our design. Our predictions were 1,120,500 um^2 which is way less than the actual area. We think that the difference in area is because of how the comparator, counters, adders and multipliers are synthesized. Although the area for Edge-detection module is more than our estimations, since the overall chip area is within our predictions, it more than compensates for it.

Sample Image Data Storage:

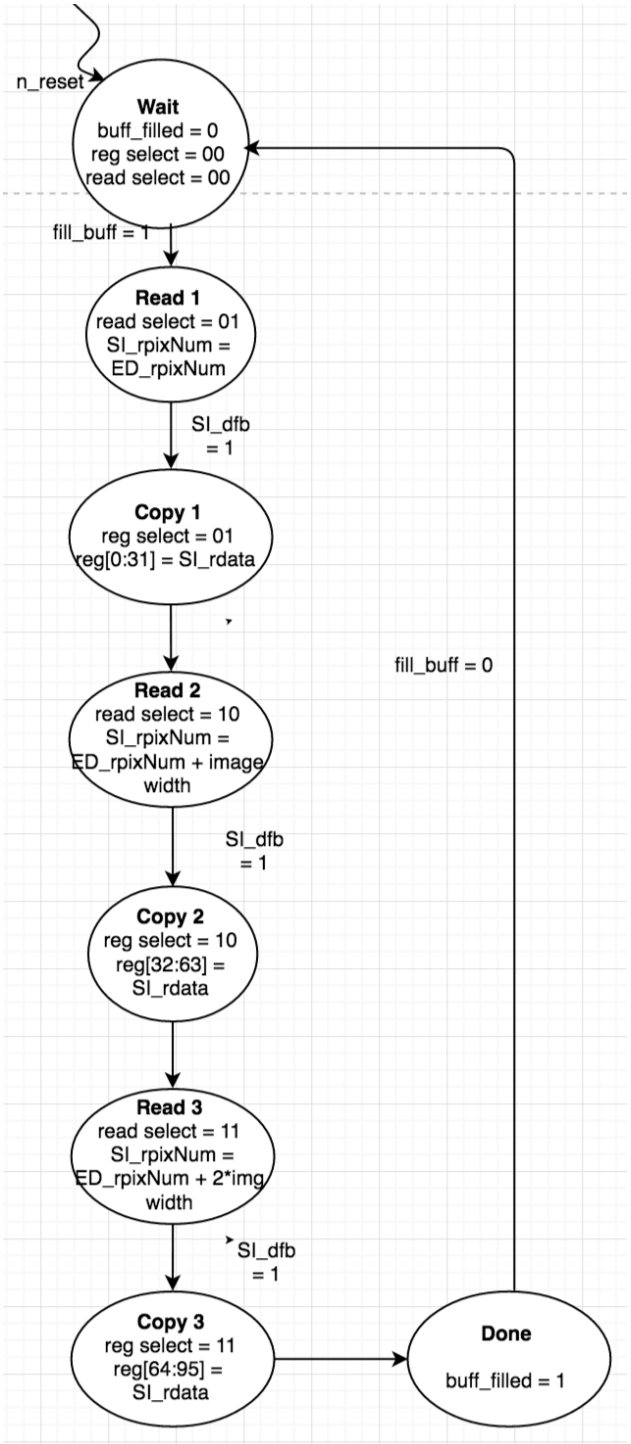
The Sample Storage block is used for getting a fixed size matrix of byte sized pixels from the image. The size of the matrix is 3x4. The Storage block follows the Edge detection controls to read a row into the matrix. The Edge detection module sets which row to be read based on the pixel address updated in its module.

- **Block Diagram -**



The block diagram above shows the interaction of the AHB Helper and the Edge detection module to Sample Image Data Storage. The number of inputs to the block are 5 which originate from the AHB Helper with the data and the feedback of the data read with also inputs from the EDM to specify to start Sample Image Data Storage and pixel position to read from. The module outputs the data from the AHB Helper to the EDM and also that it has done filling the buffer. The pix num is also provided from the EDM to the AHB Helper which it then communicates to the AHB Master so that the read occurs correctly from the SRAM.

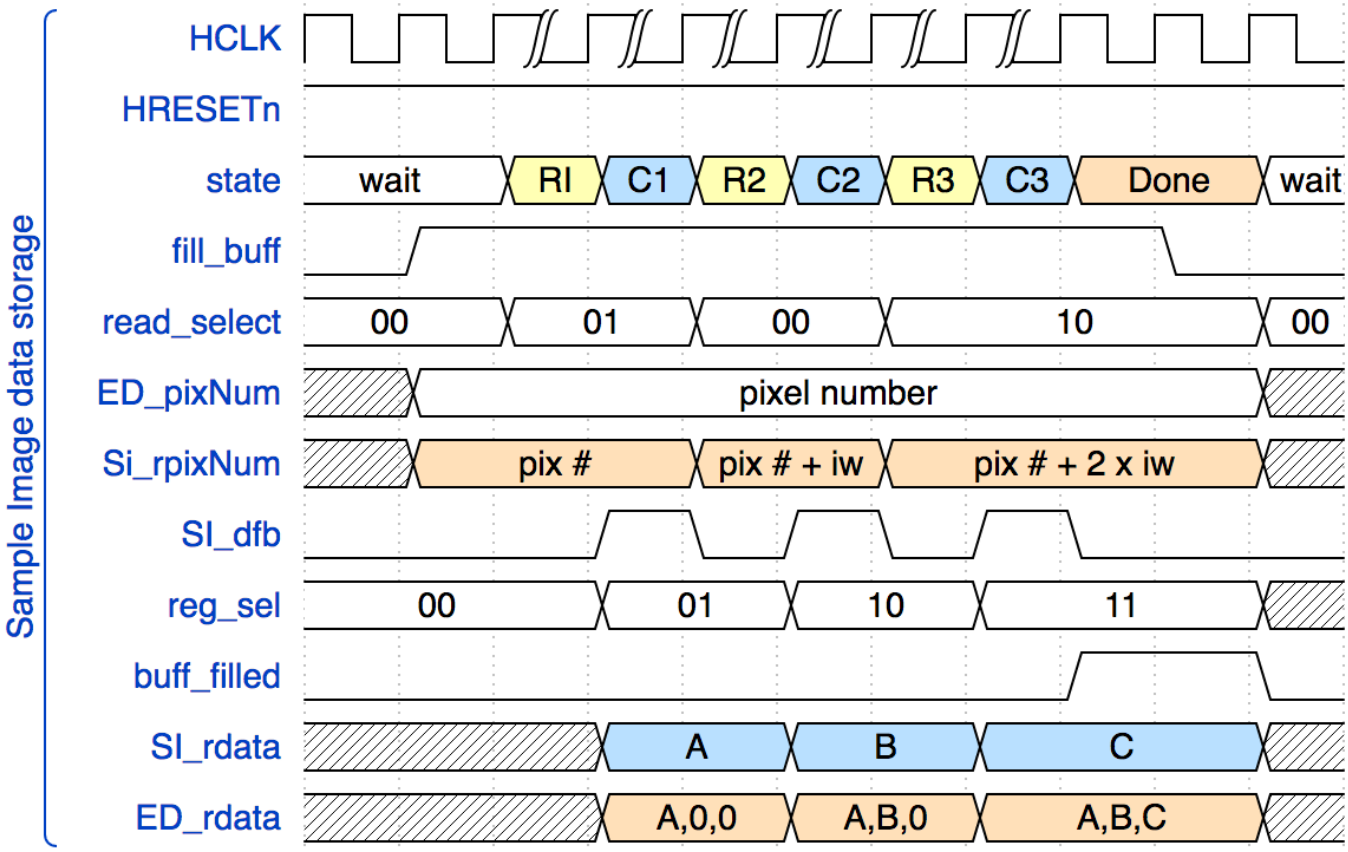
- **State Transition Diagram -**



The state transition diagram above is for Sample Image Data Storage. It has 8 states. It starts with the wait state and then when fill buff is asserted from the Edge detector module it moves to the first Read state where it reads the first

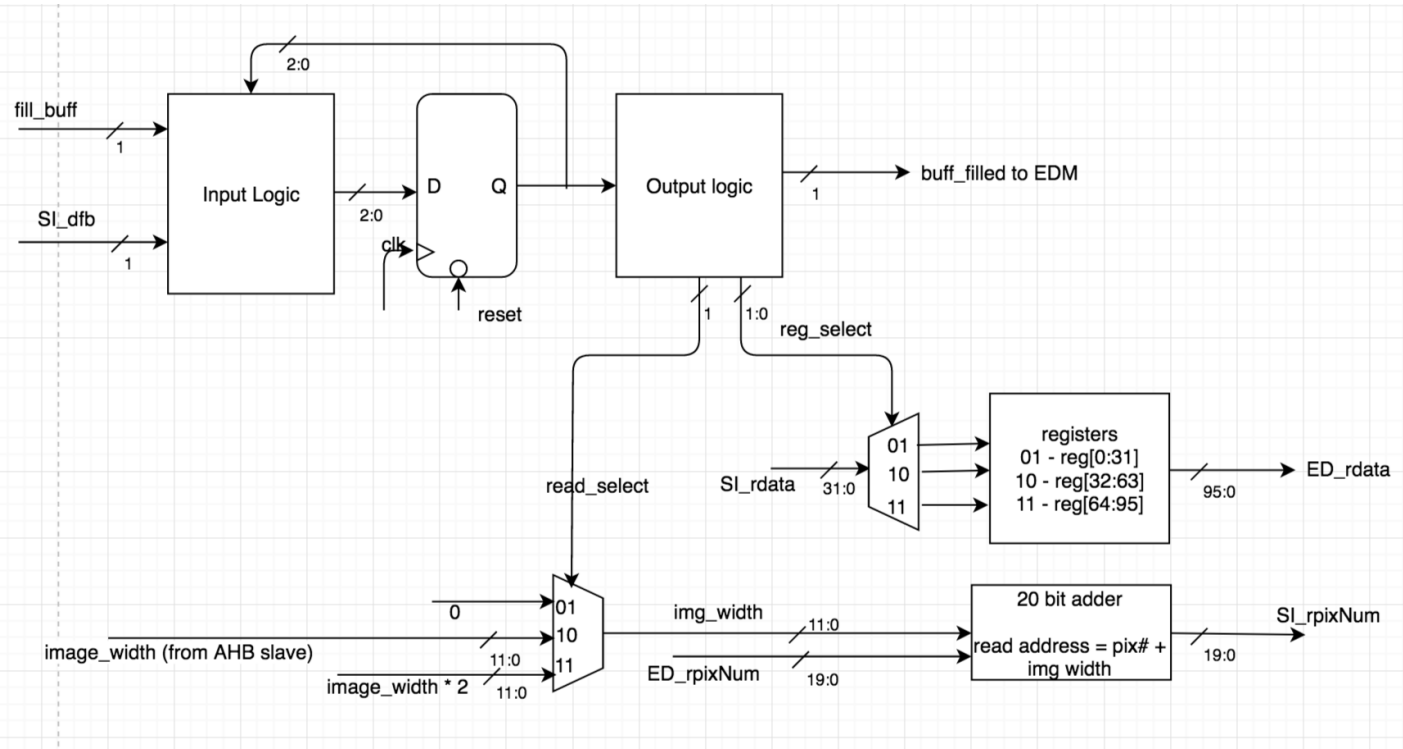
row of data from AHB Helper which reads the data from the master based on the read pixel number set by Edge Detection module. Then the state moves to Copy 1 where the read data is copied to data line that is to be supplied to Edge Detection. These two steps are repeated 2 more times with the update in address in the two reads by the Sample Image Storage itself. This update is to move to the next row of the buffer. After the 3x4 matrix is full or when there are 12 bytes of data copied to the data line to be sent to the edge detection module the storage indicates that the buffer is filled and moves to the done state.

- **Waveform -**



The waveform shows the transition of all the states from the state transition diagram. Once the fill buff is asserted then the state is moved to Read 1. Here the read select is set to 01 to show that the first row has to be read. After this when the data feedback is received, this shows that the data has been read correctly and then the state Read 2 is reached. In between states, Copy 1 asserts and copies the data to the data line that has to be read by the Edge detection module. This follows until all the three rows of the 3x4 matrix fill. After the matrix is filled and the data is correctly copied and all the states have occurred successfully, the next state reached is done which tells the Edge detector module that the data line is ready with all the pixels needed for calculating the edge gradient.

- **RTL Diagram -**



The diagram above shows a demultiplexer used to select which row of the 4X3 matrix to be populated with the data on the line from the AHB Helper which is talking to the AHB Lite interface which is fetching data from the SRAM. After the row is selected, each row is stored in registers for the read of the Edge detection module. Also there is a multiplexer to update the address of the next read of the row. This done by the multiplexer followed by the adder to get the right address to be read from. The main inputs to the input logic is to start filling the registers in the Sample Image Storage Module. The other input is feedback from the AHB helper to check if the data is read successfully because if it is not there will be more wait states added which will ensure the correct read of data.

- **Area Estimation -**

Core Area Calculations				
Name of Block	Category	Gate/FF Count	Area (um2)	Comments
Sample Storage (25 3:1 Mux)	Combinational	200	150,000	
Sample Storage (1 20 bit adder)	Combinational	100	75,000	(Worst Case 20-bit Riple Carry Adders) but we would implement carry look ahead adder
Sample Storage (96 3:1 Demuxes)	Combinational	768	576,000	
Sample Storage State Registers w/ Reset	Reg. w/ Reset	3	7,200	
Sample Storage Input Logic	Combinational	12	9,000	
Sample Storage Output Logic	Combinational	8	6,000	
Sample Storage Registers w/Reset	Reg. w/ Reset	96	230,400	

The above table gives the area estimation of the Sample Image Data Storage. The area breakdown is as follows:

1) Input logic combinational area
2) Output logic combinational area
3) State register area
4) Multiplexers and demultiplexers
5) Adder

This makes the total area of the module come to **1,053,600 micrometer squared**. The area is calculated based on estimating the number of input and output bits of each block mentioned above. With finding all the possible combinations of the inputs and outputs (input bits - 1) X (output bits), we find the number of gates for each block. With the gates of the block the area is calculated based on their category of combinational block and registers and found using known values. The area follows that the input logic used 12 gates with 3 state register and 8 gates for the output logic. There were 96 registers in the module to store the 12 byte matrix bit by bit. The demuxes and the muxes consumed 768 and 200 gates respectively. There were 200 gates for the 20 bit adder. This was due to the fact that the adder was considered a binary adder and also that it could be written in terms of 1 bit adders.

- **Actual Area Comparison with original Estimation -**

Estimated Area = 1,053,600 um^2

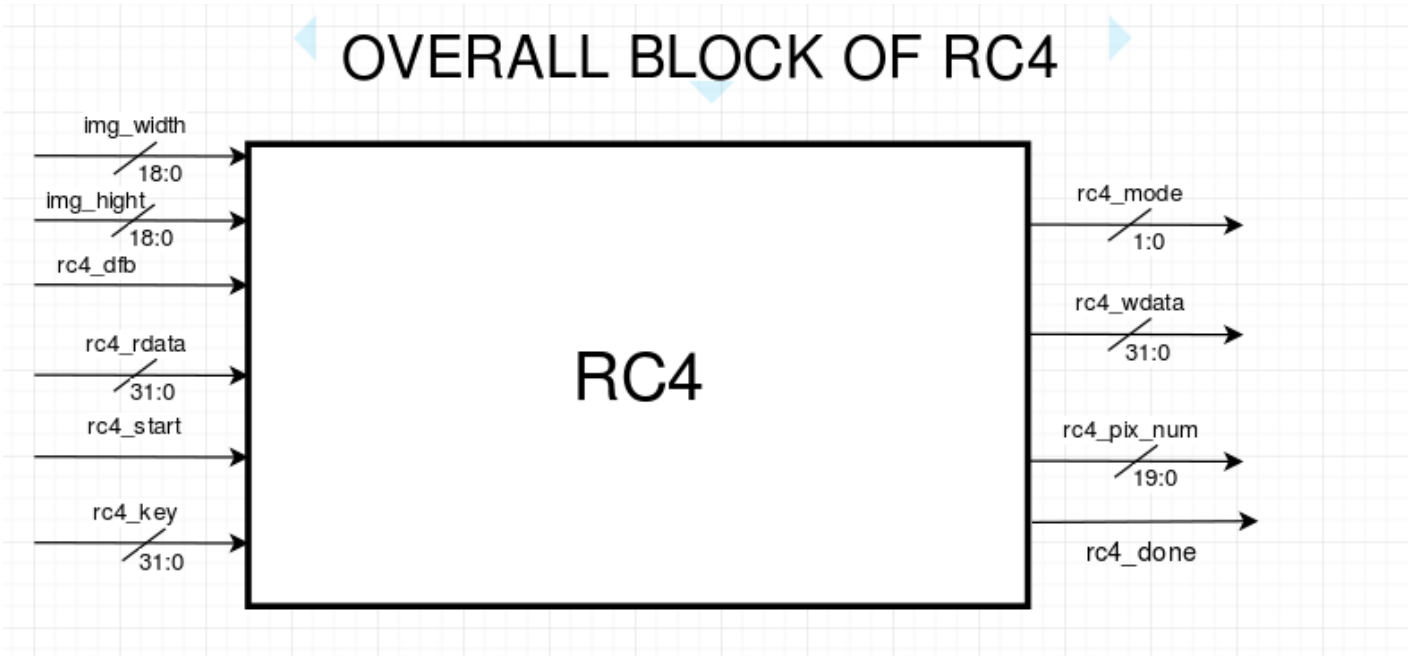
Actual Area = 265,788 um^2

The actual area of the Edge-Detection module was found to be 265,788 um^2 which is about 3% of the area of our design. Our predictions were 1,053,600 um^2 which is way more than the actual area. The area difference is because while synthesizing our design, there is some level of optimization which reduces the area and then how the adders are synthesized is different then how we interpreted it to be. Considering all these reasons, the actual area came to be much more than what we anticipated.

RC4:

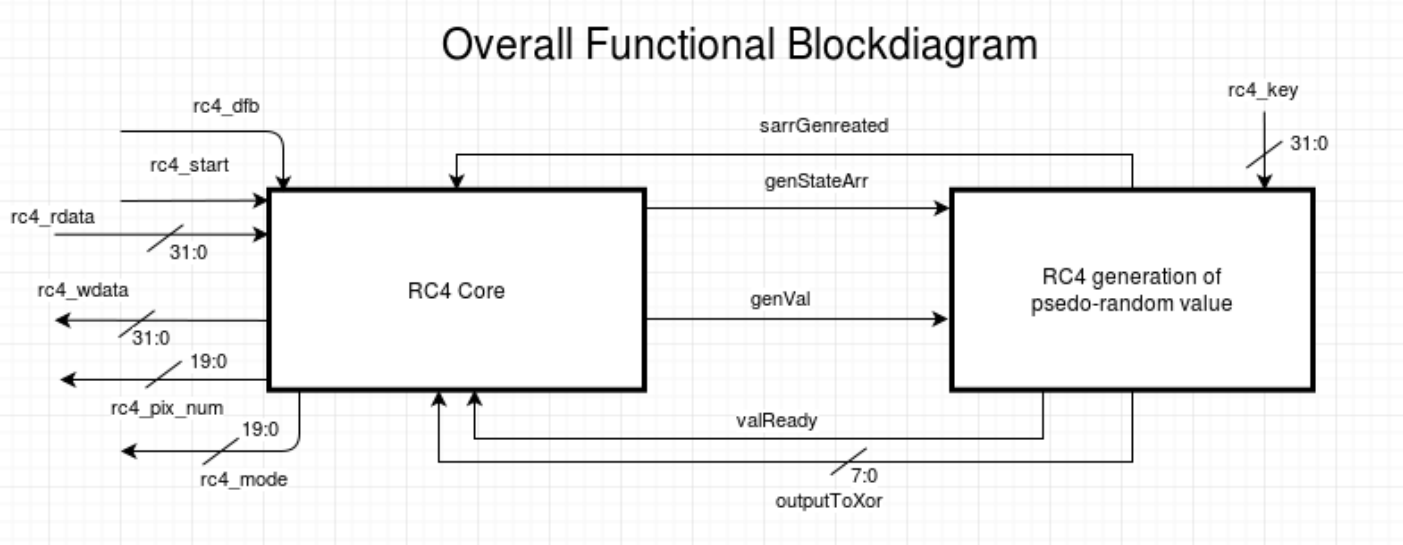
The RC4 block decrypts 4 bytes at once of the encrypted image in the SRAM until the whole image is decrypted. The whole block is controlled by a module called “RC4 Core”. As shown in the state transition diagram, before decryption begins the module needs to initialize a state array. This is accomplished in RC4 generation of psedo-random value module with a section of the FSM called “RC4 Initialization” as noted on the side. It first initializes the state array which is located internally in the module with values from 0 to 255 and performs the operations specified by the RC4 standard using the key “Dr.J” to prepare the state array(those operations could be seen in the pseudo-code). After the array is ready, the next state in the state transition diagram in “RC4 CORE” tells AHB Helper to read 4 bytes from SRAM. Then RC4 generation of pseudo-random value module performs operations on the State Array to generate value for decryption. After each value is generated, it is used to decrypt one of the 4 bytes read by AHB Helper. All those operations are accomplished with the part of the FSM called “RC4 gen Xor.” The “RC4 Core” waits until the “RC4 generation of pseudo-random value” module has performed all the operations and generated the value. That value is used to be xor with the input data (encrypted byte) in the next state. After the xor operation, “RC4 core” continues to work on to the next byte until all four bytes are decrypted. After, it tells AHB Helper to write into SRAM.

- **Block Diagram -**

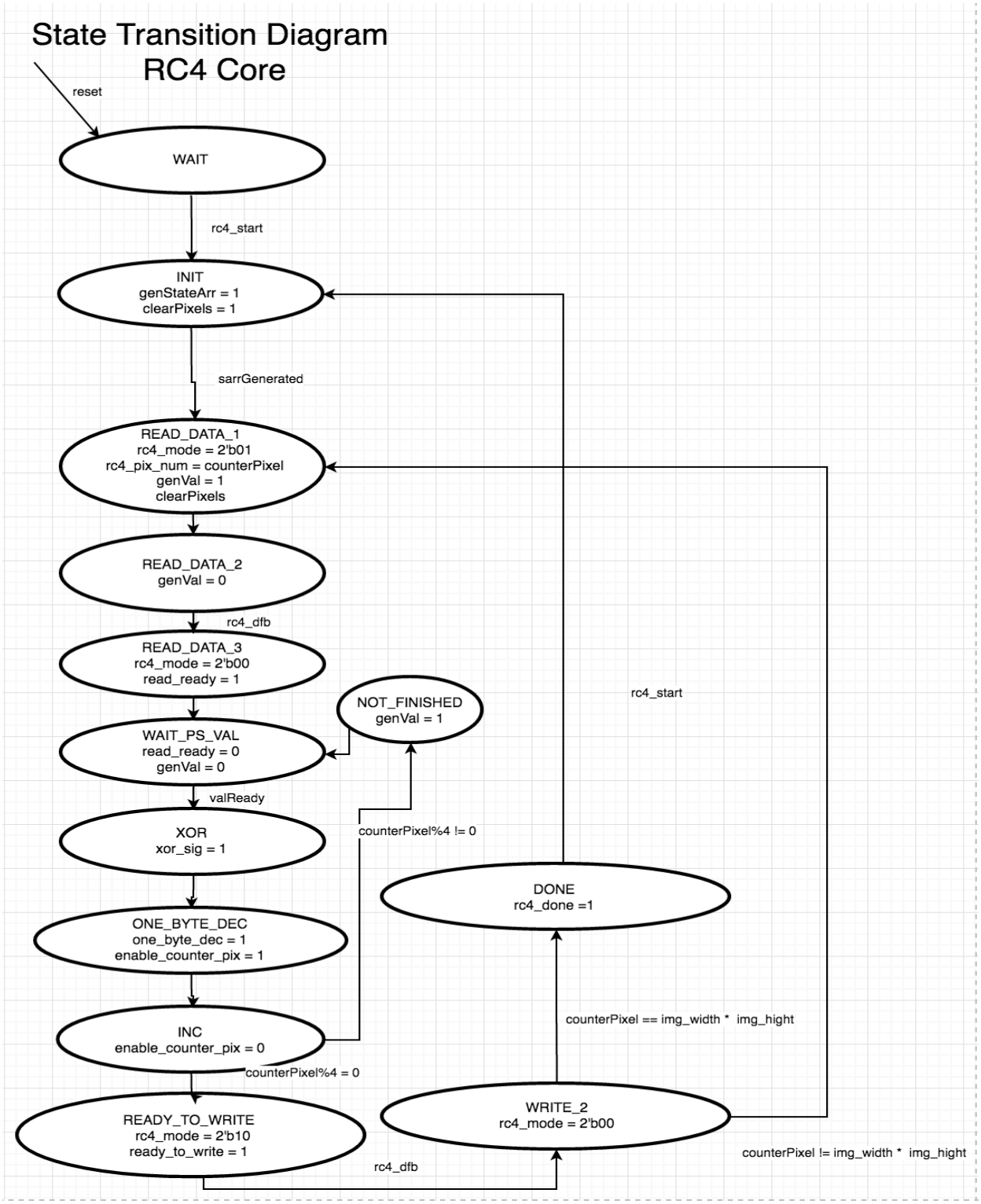


The first image above shows the block diagram of RC4 module. The module is supplied with the image width and image height to determine the number of bytes to decrypt. RC4_mode determines if the RC4 module want to read or write and the data is supplied by rc4_rdata, rc4_wdata respectively. The address is supplied by the RC4 module with rc4_pix_num. Rc4_start start the module and rc4_done is used to signal that the module is completed decryption. The

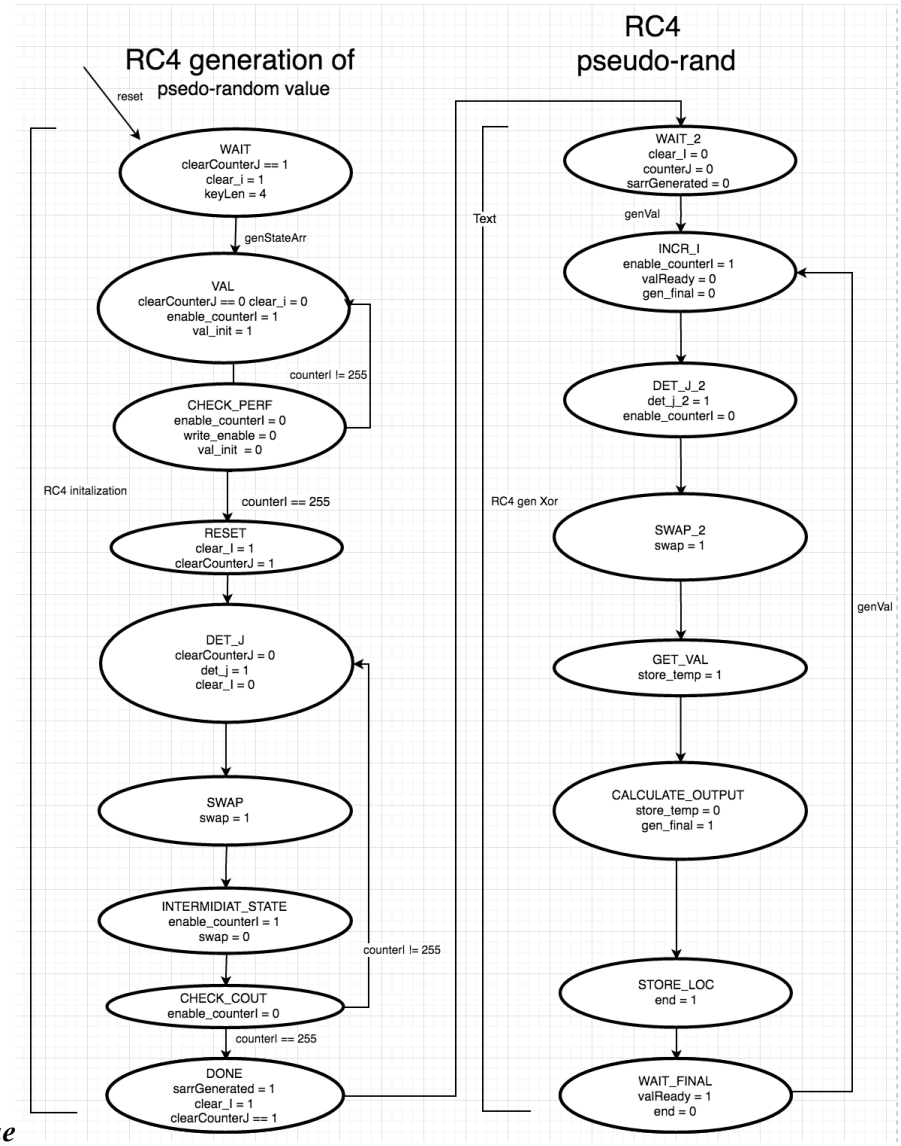
second image below demonstrates the interaction between the modules within the RC4. genStateArray from RC4 Core tells the second module to initialize the state array and genVal tells to generate just one byte used for decryption. sarrGenerated and valReady respectively give feedback to the RC4 Core.



- **State transition Diagrams -**



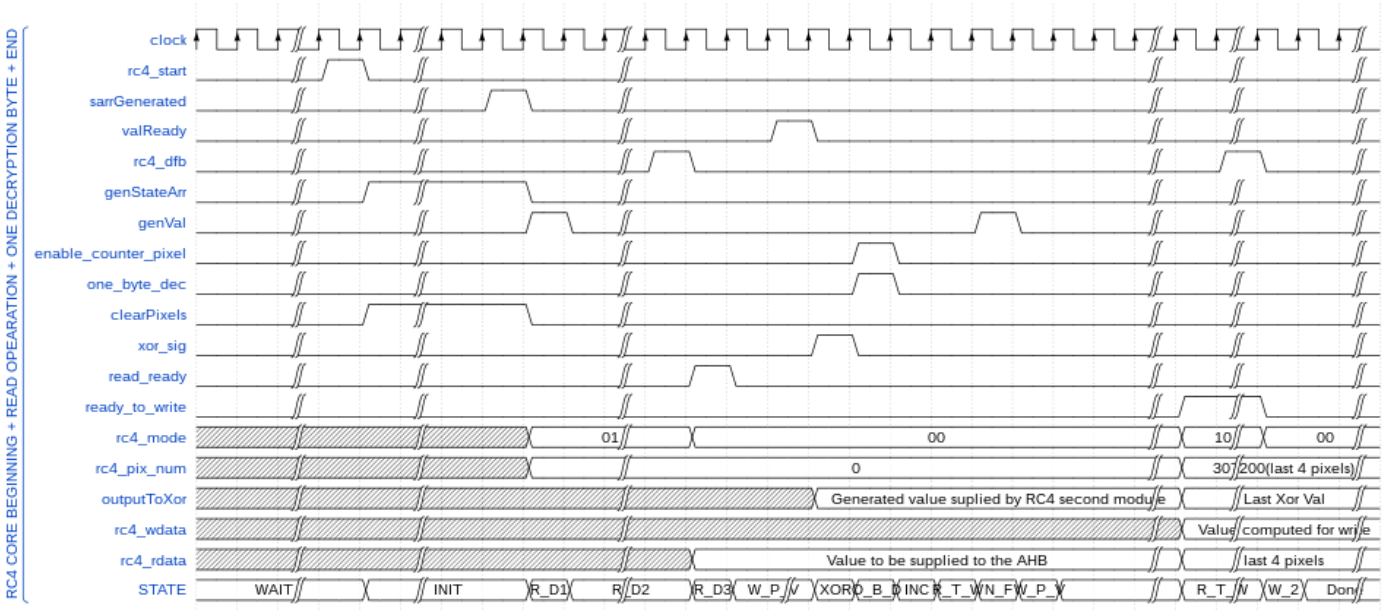
RC4 Core waits for receiving rc4_start signal. Once it receives it, it waits for RC4 generation of pseudo-random value module to initialize the array. Once RC4 controller receives sarrGenerated, it sends a command to AHB master to read four bytes of data from the SRAM. The data is put or rc4_rdata. Once the data is read (rc4 received rc4_dfb), rc4 needs to produce a value to xor the input to decrypt it. That is accomplished by the assertion of genVal signal. That state waits until valReady is asserted. Once the value is ready, it is xored with the input data and then, put in a register. Then the RC4 goes and decrypts the next byte of the 4 bytes. Once all of the four bytes are decrypted RC4 tells (using rc4_mode = 10) AHM Master to write the data stored in the registers to the SRAM. Then RC4 checks against the total count of pixels to determine if it needs to decrypt more data.



RC4 generation of pseudo-random value

RC4 generation of pseudo-random value module is basically one big FSM, which is split into two to be easier to understand. Other smaller blocks take care of complex functions like swapping values in array, adding, subtracting, etc. FSM just tells them when to perform their functions. First part of the FSM is called “RC4 initialize” which is basically responsible for initializing the internal state array (contained in Sarra block in the RTL). It starts by waiting for assertion of genStateArr signal. The next two states are responsible for going through the array and initializing each value from 0 to 255. After that, the states DET_j through CHECK_OUT perform the actual permutation of rc4 state array by using the key to prepare the array for decryption. Those steps could be found in the pseudo-code(Key-Scheduling Algo). After that the state machine asserts sarrGenerated and transfers the control to the RC4 core. Next RC4 generation of pseudo-random value module waits for genVal to become high. This will start the next part of the FSM called “RC4 gen Xor” which is responsible for generating values which are used by the RC4 Core to decrypt one byte of rc4_rdata. After the signals genVal is received the states shown in the diagram accomplish the sequence of operations specified in the pseudo-code(called Generation of decryption byte). At the end the value is outputted in the outputToXor signal and valReady is asserted.

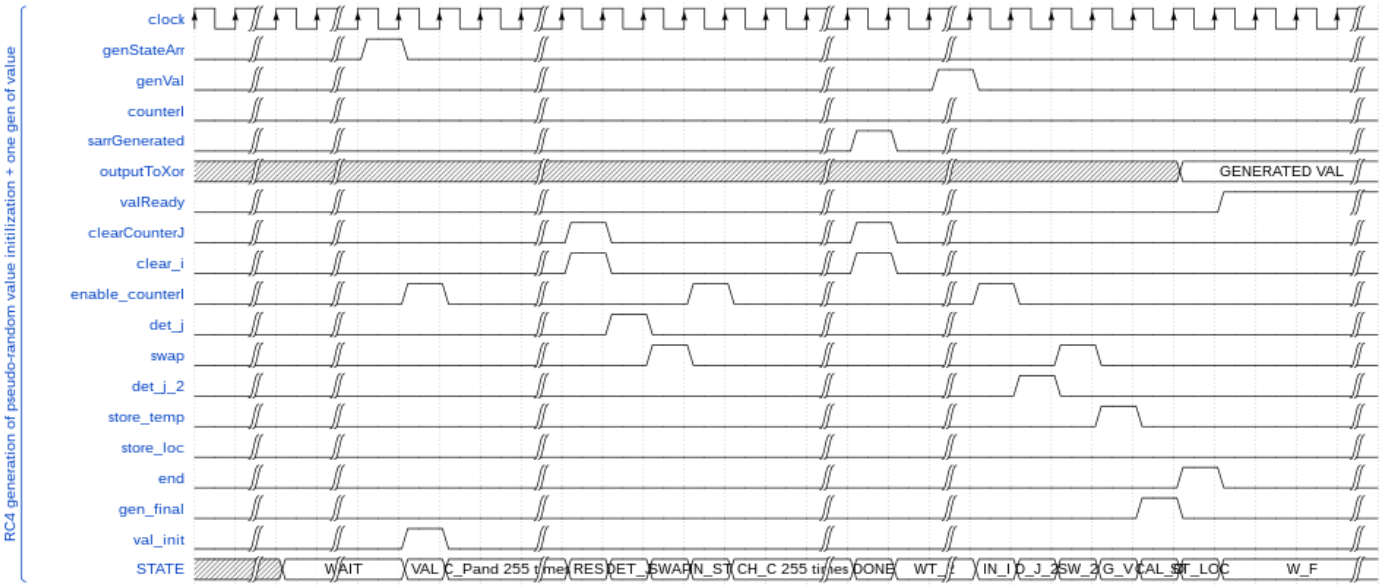
- **Waveform Timing Diagram**



The figure shows a timing diagram for both modules in RC4. On the top, you can see RC4. The waveform begins with rc4_start signal. Then the rc4 core asserts genStateArry to initialize the state array in the “RC4 generation of pseudo-random value” module. After that the rc4 core sends to the ahb master rc4_mode to 01 which means read a memory from SRAM. RC4 core supplies the exact memory as shown in the waveform by rc4_pix_num. After the request was process rc4_dfb is set high for one clock cycle which means that rc4_mode should be set as an IDLE (no requests). Then, rc4 core continues by waiting for sarrGenerated which means that the other module has generated a value to xor. Then RC4 core module xors the encrypted input and stores the value. The decryption is shown for only one byte of decryption and after that just the ending of the state diagram when all the pixels were decrypted.

Note: each state was abbreviated to fit in the waveform.

R_D1,R_D2,R_D3 - READ_DATA1,2,3
W_P_V - WAIT_PS_VAL
O_B_D - ONE_BYTE_DEC
R_T_W - READY_TO_WRITE
N_F - NOT_FINISHED
R_T_W - READY_TO_WRITE
W_2 - WRITE_2



The bottom of the waveform diagram demonstrates the states for RC4 generation of pseudo-random value. It first begins with initializing the state array with values from 0 to 255. The diagram shows only the first of the 255 values that are initialized. Then the diagram continues with showing the steps which accomplish "j:= (j + S[i] + key[i mod keylength]) mod 256". Basically, the waveform shows the signals asserted to external smaller modules which take care of operations such as addition. These are the states from DET_J until DONE. With that the initilization of the state array finishes. After that the waveform demonstrates generation of one byte to be used for xoring. Again, it asserts signals which commands other modules to take care of the calculations.

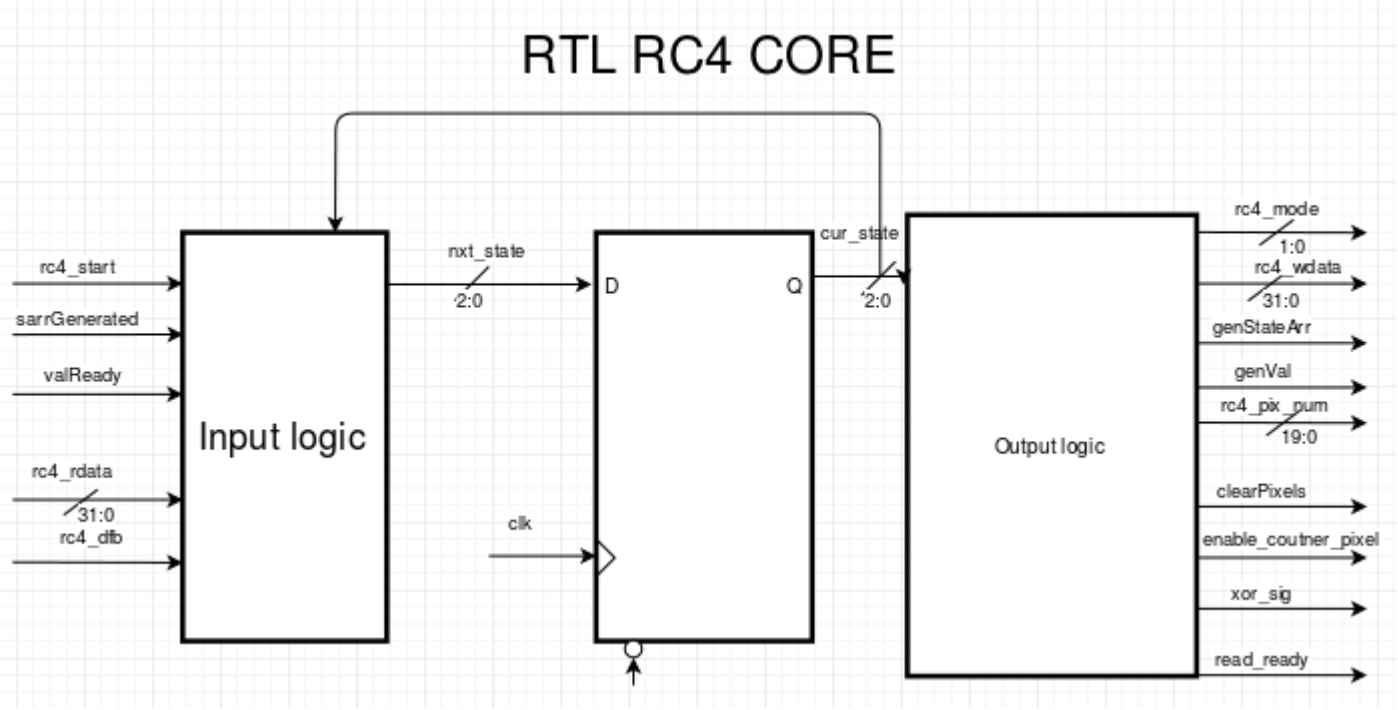
Note: each state was abbreviated to fit in the waveform.
C_Pand 255 times -
RES - RESET
N_ST - INTERMIDIAT_STATE
CH_C 255 times - REPETITION OF 255 time from CHECK_OUT
WT_2 - WAIT_2
IN_I - INCR_I
D_J_2 - DET_J_2
SW_2 - SWAP_2

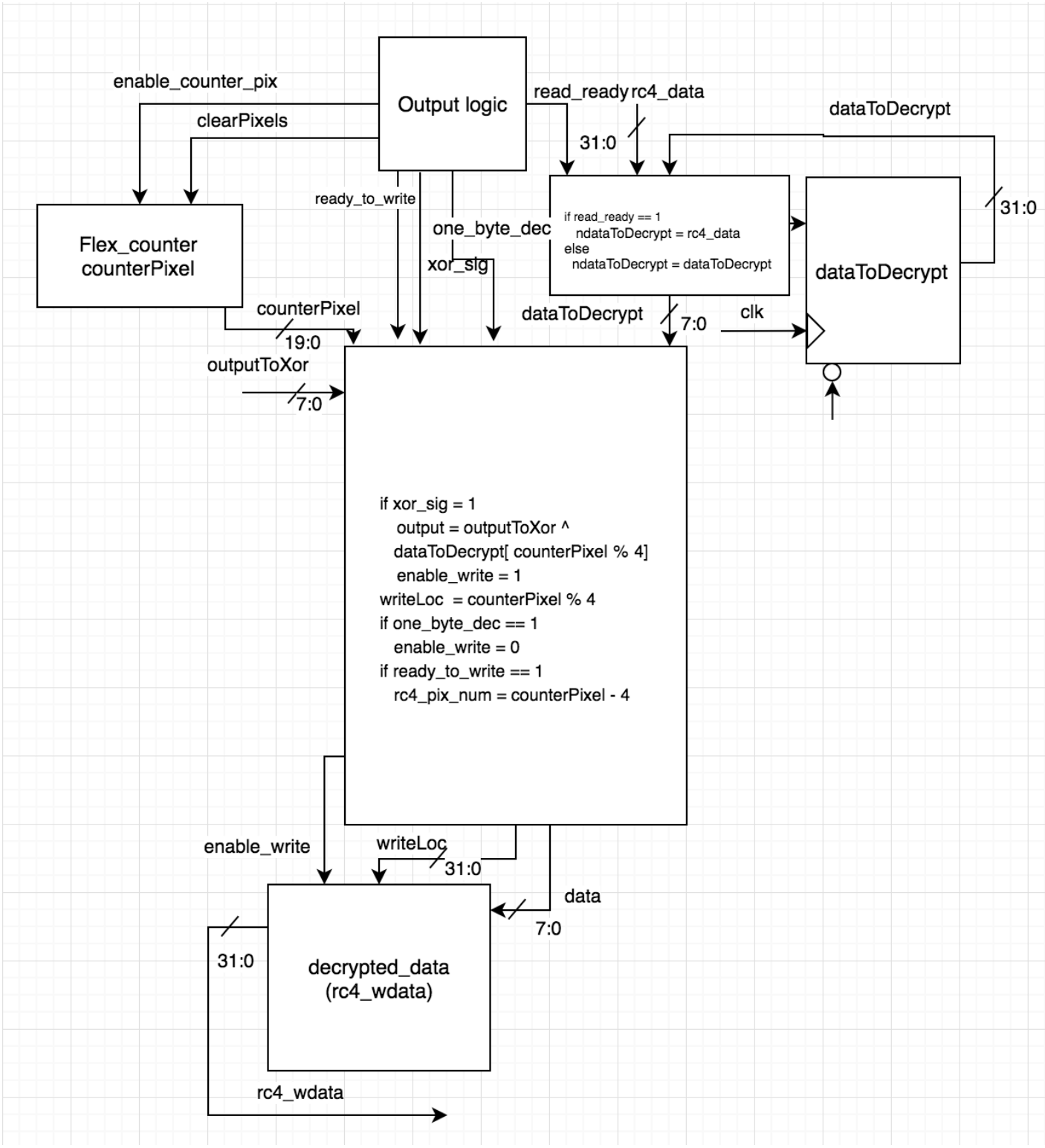
G_VAl - GET_VAL
CAL_O - CALCULATE_OUTPUT
ST_LOC - STORE_LOC
W_F - WAIT_FINAL

- **RTL Diagram -**

RC4 Core

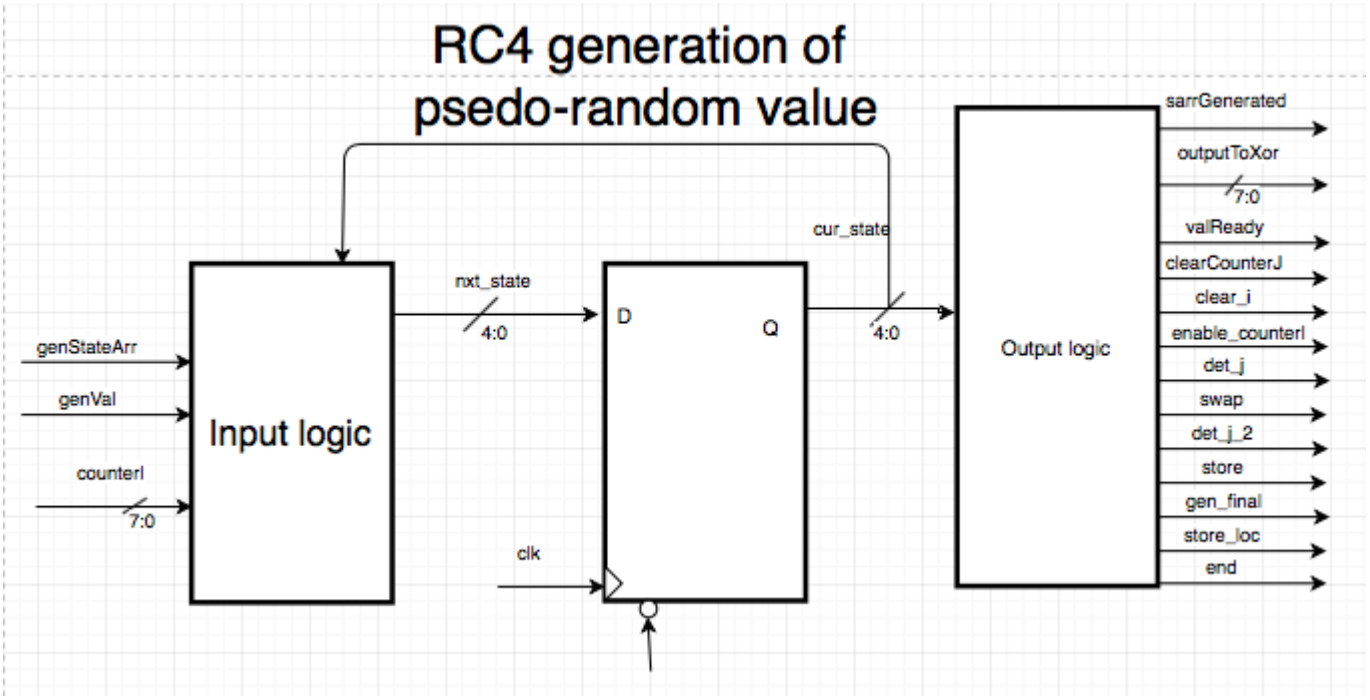
RTL below illustrates just a FSM for the “main brain” of the RC4. It contains an input and output logic which are better explain below with the help of the state transition diagram. The second image shows all the smaller blocks that are designed to help the FSM and take care of complex combinational tasks. That RTL is is easier to understand with the help of the state transition diagram later.

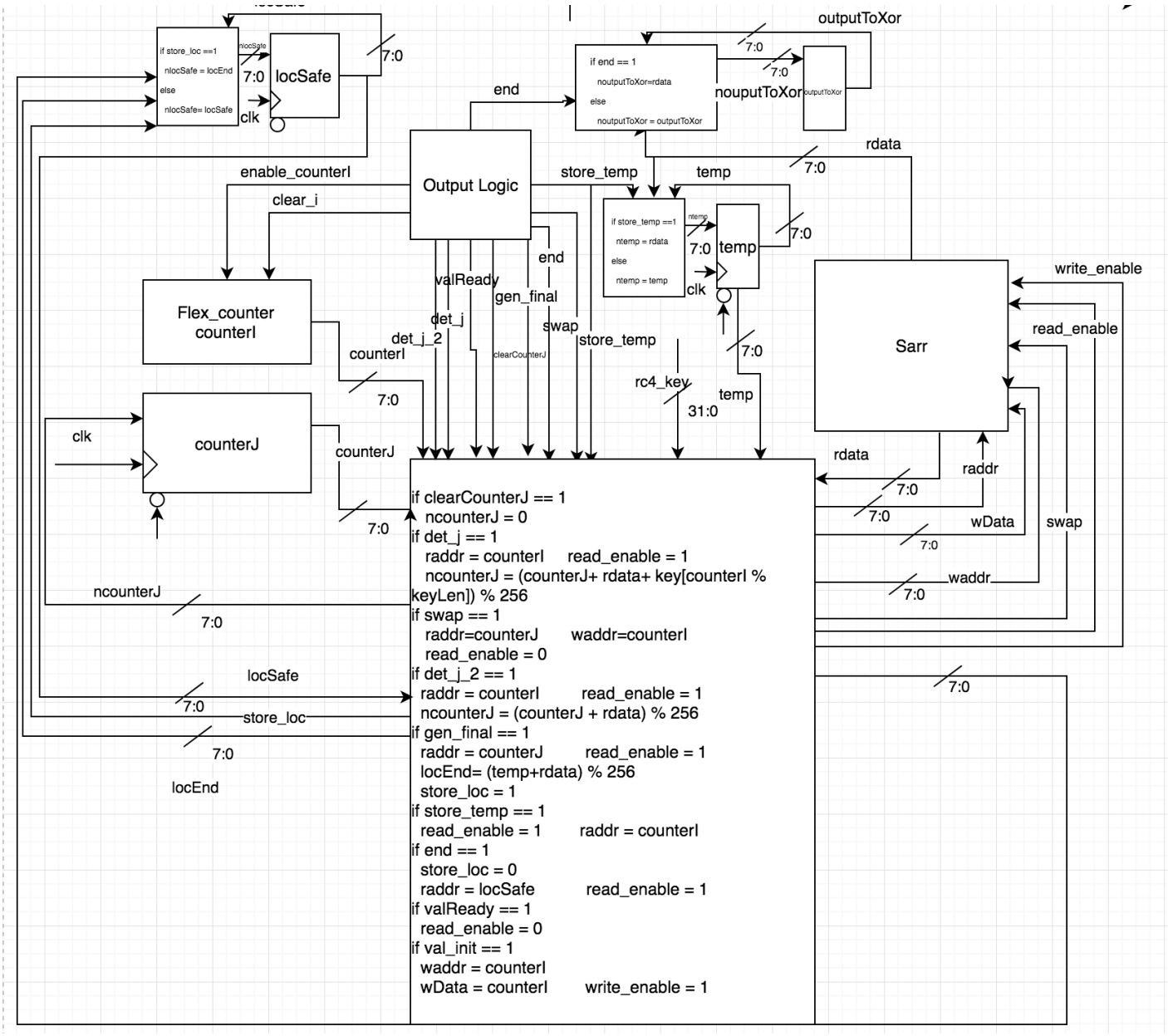



Output logic
enable_counter_pix
clearPixels
read_ready
rc4_data
31:0
dataToDecrypt
ready_to_write
one_byte_dec
xor_sig
Flex_counter
counterPixel
if read_ready == 1
ndataToDecrypt = rc4_data
else
ndataToDecrypt = dataToDecrypt
dataToDecrypt
31:0
counterPixel
19:0
dataToDecrypt
7:0
clk
outputToXor
7:0
if xor_sig = 1
output = outputToXor ^
dataToDecrypt[counterPixel % 4]
enable_write = 1
writeLoc = counterPixel % 4
if one_byte_dec == 1
enable_write = 0
if ready_to_write == 1
rc4_pix_num = counterPixel - 4
enable_write
writeLoc
31:0
data
7:0
31:0
decrypted_data
(rc4_wdata)
rc4_wdata

RC4 generation of pseudo-random value

RTL below illustrates the FSM for the module responsible for initialization of the state array in the RC4 and the generation of each random value used for decryption in the RC4. This module contains internal memory for the State Memory and the key. Below I have included the RTL including all the smaller helper blocks. The state array is contained in the Sarr block in the rtl. The rest is easy to follow with the help of the state transition diagram.




if store_loc ==1
nlocSafe = locEnd
else
nlocSafe= locSafe
nlocSafe
7:0
locSafe
clk
7:0
if end == 1
noutputToXor=rdata
else
noutputToXor = outputToXor
7:0
nouputToXor
outputToXor
outputToXor
end
rdata
7:0
enable_counterI
clear_i
Output Logic
store_temp
temp
if store_temp ==1
ntemp = rdata
else
ntemp = temp
ntemp
7:0
temp
clk
7:0
end
valReady
gen_final
det_j
det_j_2
swap
store_temp
clearCounterJ
counterI
7:0
Flex_counter
counterI
rc4_key
31:0
temp
7:0
Sarr
write_enable
read_enable
clk
counterJ
counterJ
7:0
rdata
7:0
raddr
7:0
wData
swap
7:0
waddr
7:0
ncounterJ
7:0
if clearCounterJ == 1
ncounterJ = 0
if det_j == 1
raddr = counterI read_enable = 1
ncounterJ = (counterJ+ rdata+ key[counterI %
keyLen]) % 256
if swap == 1
raddr=counterJ waddr=counterI
read_enable = 0
if det_j_2 == 1
raddr = counterI read_enable = 1
ncounterJ = (counterJ + rdata) % 256
if gen_final == 1
raddr = counterJ read_enable = 1
locEnd= (temp+rdata) % 256
store_loc = 1
if store_temp == 1
read_enable = 1 raddr = counterI
if end == 1
store_loc = 0
raddr = locSafe read_enable = 1
if valReady == 1
read_enable = 0
if val_init == 1
waddr = counterI
wData = counterI write_enable = 1
locSafe
7:0
store_loc
7:0
locEnd
7:0

- **Area Estimation -**

Core Area Calculations				
Name of Block	**Category**	**Gate/FF Count**	**Area (um2)**	**Comments**
RC4 PSEUDO Input Logic	Combinational	60	45,000	
RC4 PSEUDO State Registers	Reg. w/ Reset	5	12,000	
RC4 PSEUDO Output Logic	Combinational	84	63,000	
RC4 PSEUDO Combinational Lo	Combinational	2108	1,581,000	
RC4 PSEUDO (3 2:1 Mux)	Combinational	24	18,000	
RC4 PSEUDO 2072 regs	Reg. w/ Reset	2072	4,972,800	Regs for State Array(255*8) + LocSafe + OutputToXOR + Temp + CounterJ
RC4 PSEUDO 8-bit Flex Counter	Reg. w/ Reset	8	19,200	
RC4 PSEUDO 255:1 Mux	Combinational	1020	765,000	
RC4 PSEUDO 1:255 Demux	Combinational	765	573,750	
RC4 Core Input Logic	Combinational	160	120,000	
RC4 Core State Registers	Reg. w/ Reset	4	9,600	
RC4 Core Output Logic	Combinational	150	112,500	
RC4 Core Flex Counter (20-bit)	Reg. w/ Reset	20	48,000	
RC4 (2 2:1 Mux)	Combinational	8	6,000	
RC4 Core regs	Reg. w/ Reset	62	148,800	
RC4 Combinational Logic	Combinational	1558	1,168,500	

The above table gives the area estimation of the RC4 module. The area breakdown is as follows:

RC4 PSEUDO:

1) Input logic combinational area
2) State registers
3) Output logic combinational area
4) All possible combinational area
5) All area related to State array like 2072 registers 1 mux and 1 demux.

RC4 CORE:

1) Input logic combinational area
2) State registers
3) Output logic combinational area
4) Flex Counter
5) All other smaller modules helping RC4 area (combinational and registers)

This makes the total area of the module come to **9,663,150 micrometer squared**. The area is calculated based on estimating the number of input and output bits of each block mentioned above. With finding all the possible combinations of the inputs and outputs (input bits - 1) X (output bits), we find the number of gates for each combinational block. With the gates of the block the area is calculated based on their category of combinational block and registers and found using known values. The area for muxes are added if 1 2:1 Mux consists of 5 gates and that each bit would require 1 2:1 mux. Another estimation we made was that 255:1 Mux consist of 255 number of 2:1 muxes.

- **Actual Area Comparison with original Estimation -**

Estimated Area = 9,663,150 um^2

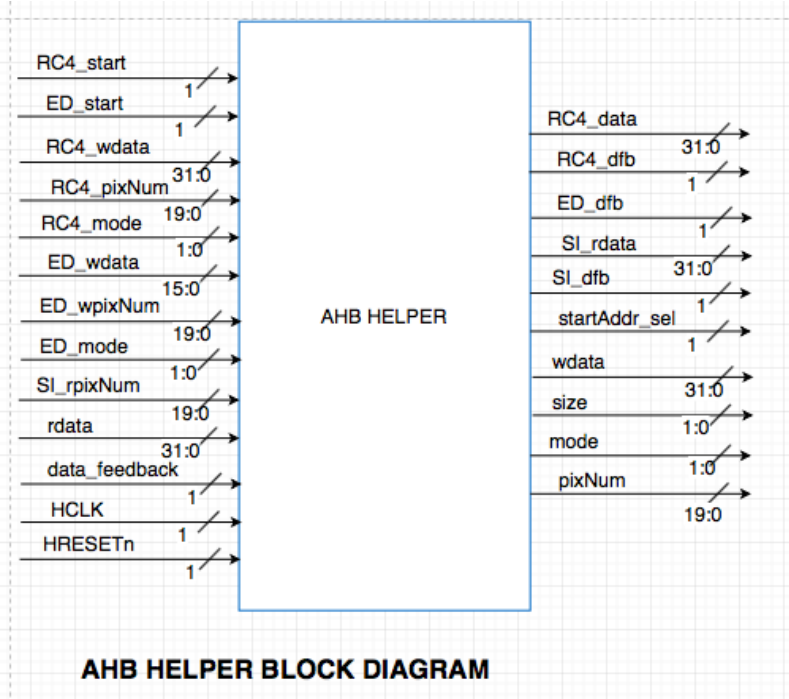
Actual Area = 6,194,619 um^2

Since the beginning, we anticipated RC4 to be most size consuming module in our design since it requires a state array of 255 registers which itself was expected to take 4,972,800 um^2 of area. However, after synthesizing the design, we found RC4 area to be 6,194,619 which is about 36% less than our predicted area. However, even after synthesizing the design and looking up the report, we found that RC4 takes up about 69% which is still the most space consuming module in our design. RC4 core takes up 528,741 um^2 while RC4 pseudo random module takes about 5,665,878 um^2 which aligns with our predictions of taking a lot of space due to the state array. However, our estimation about the area of the state array was different than the actual area primarily because of how the synthesis and optimization takes place. We estimated the area of state array to be 4,972,800 um^2 but we actually got to be 5,469,408 um^2 of area. The underestimation of area in this part is compensated by the overestimation of the area in other part of RC4 which in total still results in much less actual area than estimated.

AHB-Helper:

AHB-Helper module is present just to assist the AHB-Lite master in setting the control and address signals for the AHB interface. It basically relays the information from the module that asks for a request, whether it be RC4, Sample image storage or the Edge-Detection module. It is a purely combinational block thus adding not a significant amount of delay.

- **Block Diagram -**



AHB HELPER BLOCK DIAGRAM

The above diagram shows the block diagram of the AHB-Helper.

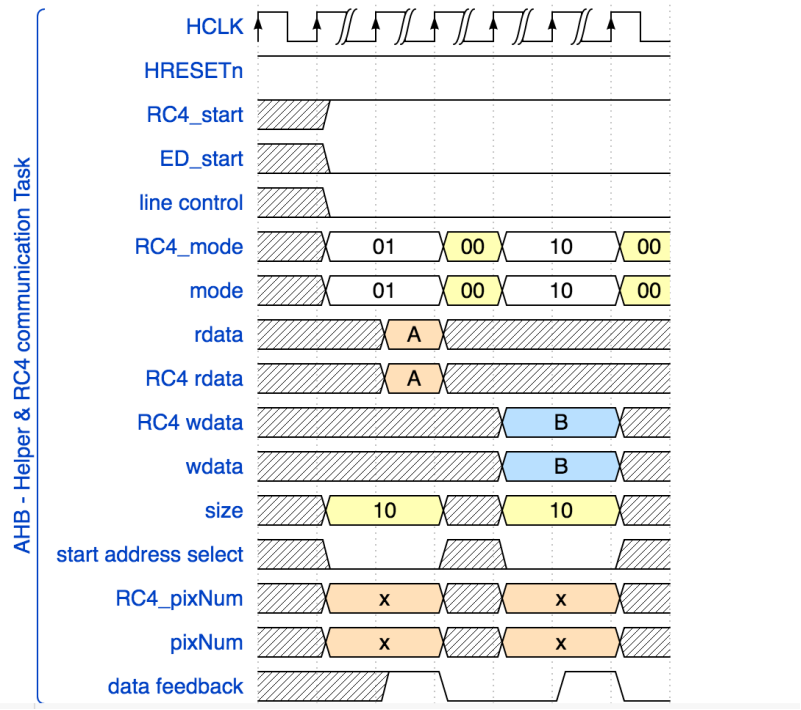
All of the inputs coming into the AHB helper are just signals that the AHB helper needs to relay to the AHB master from the various modules or vice versa. However, in order to choose the appropriate set of signals to send, the AHB helper makes use of ED_start and RC4_start.

In addition to selecting the correct inputs as outputs, it also sends out one additional signal called startAddr_sel. This signal is intended to instruct the AHB master whether to add an additional off-set while writing information from the edge detection module. This is because while writing from the edge detection module it is important to not overwrite the old image information.

- **State Transition Diagram -**

Since the AHB-Helper block is purely combinatorial, it does not have a state diagram. All outputs are determined through combinational logic.

- **Timing Waveform Diagram -**

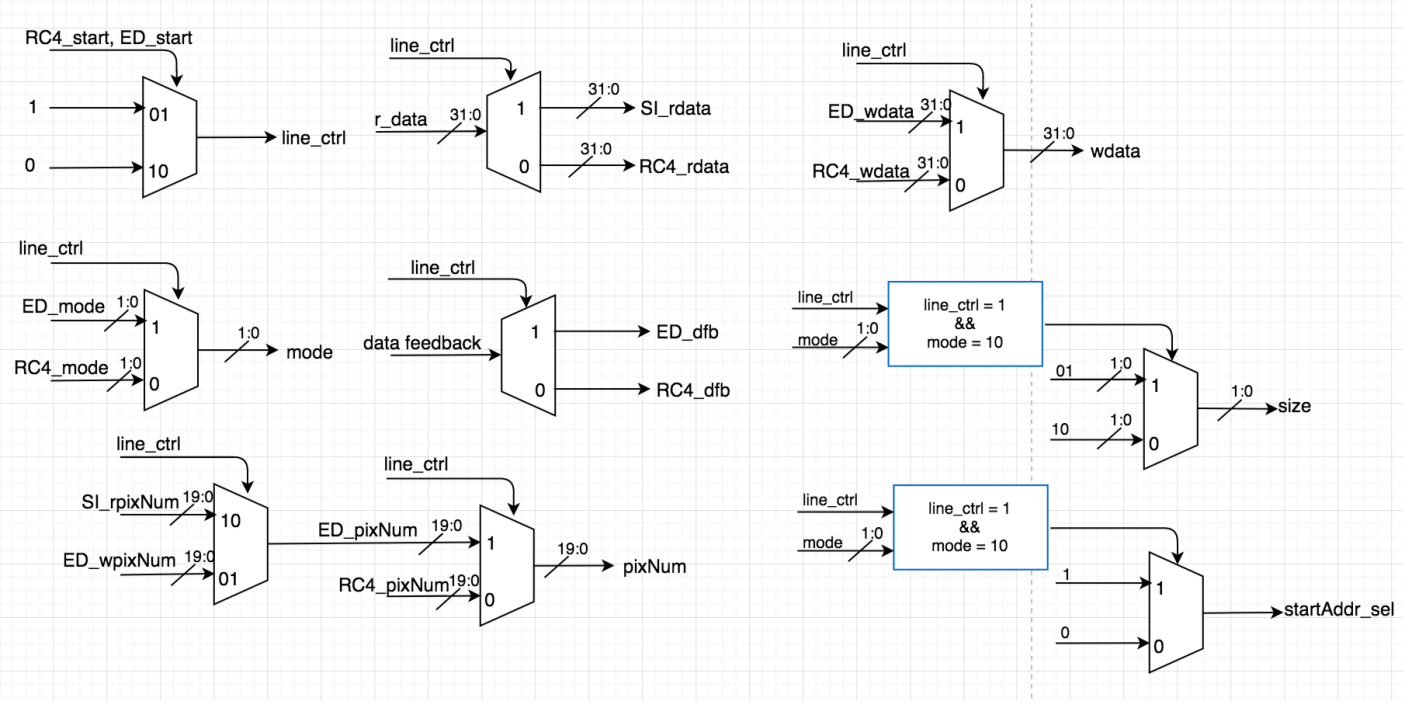


As described above, the AHB helper is purely combinatorial. As a result, the timing waveform does not have a changing state. It checks the current value of RC4_start and ED_start, and determines which inputs to make as the current outputs.

When RC4_start is asserted, the AHB helper identifies that the RC4 module is ON, implying that edge detection is OFF. This means that the AHB helper is currently relaying signals between the RC4 module and the AHB Master. When ED_start is asserted, the AHB helper identifies that the Edge detection module is ON, implying that the RC4

module is OFF. This means that the AHB helper is currently relaying signals between the Edge detection module/sample image storage module and the AHB Master.

- **RTL Diagram -**



Since the AHB helper is purely combinatorial, there is no state machine in its RTL diagram. However, there are several MUXs to choose between the various inputs.

There is one internal signal called line_ctrl which is largely responsible for choosing between all the inputs. It is kept at 0 when the RC4 is running, and is asserted to 1 when Edge detection is running. This signal decides which module's lines will be connected to the AHB Master.

Besides this, there are two signals that are internally decided within the AHB helper. These are size and startAddr_sel. This is necessary because the write state of the edge detection differs from the rest of the read/write operations in the fact that it only writes 2 bytes, and those two bytes are written to a completely new address location, beginning from the ending address of the decrypted image. This occurs when line control is equal to 1 (implying that edge detection is taking place), and mode is equal to '10'(implying that it is in a writing phase). Hence size is set to '01' which corresponds to 2 bytes, and startAddr_sel is set to 1. StartAddr_sel is sent into a mux in the AHB master, which adds an additional offset to the writing address-

- **Area Estimation -**

Name of Block	Category	Gate/FF Count	Area (um2)	Comments
AHB Helper Muxes (58 2:1)	Combinational	232	116,000	
AHB Helper DeMuxes (33 1:2)	Combinational	132	1,500	
AHB Helper (2 2 bit comparato	Combinational	20	7,500	
		Total Area -	125,000	

The above table gives the area estimation of the AHB-Slave module. The area breakdown is as follows:

1) 2:1 Muxes
2) 1:2 Demuxes
3) 2-bit Comparators

This makes the total area of the module come to **125,000 micrometer squared**. The area for muxes are added if 1 2:1 Mux consists of 5 gates and that each bit would require 1 2:1 mux. It was found that one 2-bit comparator consisted of 10 gates, hence for 2 2-bit comparators, we estimated about 20 gates.

- **Actual Area Comparison with original Estimation -**

Estimated Area = 125,000 um^2

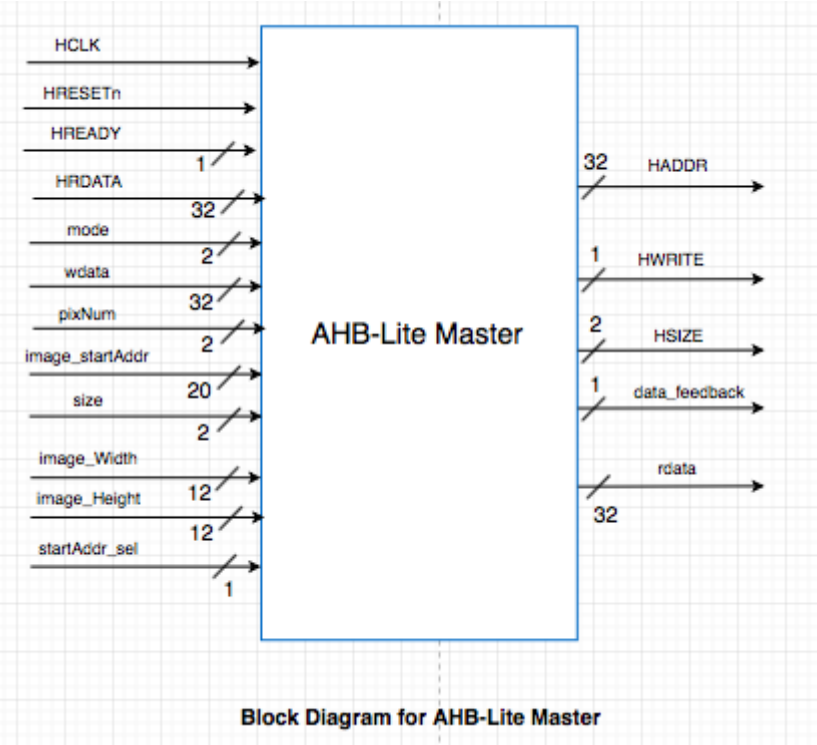
Actual Area = 55,845 um^2

AHB-Helper is the only module in our design that is purely combinational which made it is easy for us to estimate its area in comparison to other modules. Our original predictions were of 125,000 um^2 but the actual area was found to be 55,845 um^2 which is 0.6% of the total design area. The major area difference we think is because of how the comparator and muxes/demuxes are actually synthesized within the design. Of course because of some level of optimization, the actual area came out to be less than our worst case predicted area.

AHB-Lite Master:

AHB-Lite Master serves as the primary communication between our design and the off-chip SRAM. It receives the read/write instruction from RC4 or Edge-Detection module and does the instruction. Once it does the transfer, it sends a feedback to the respective module as well.

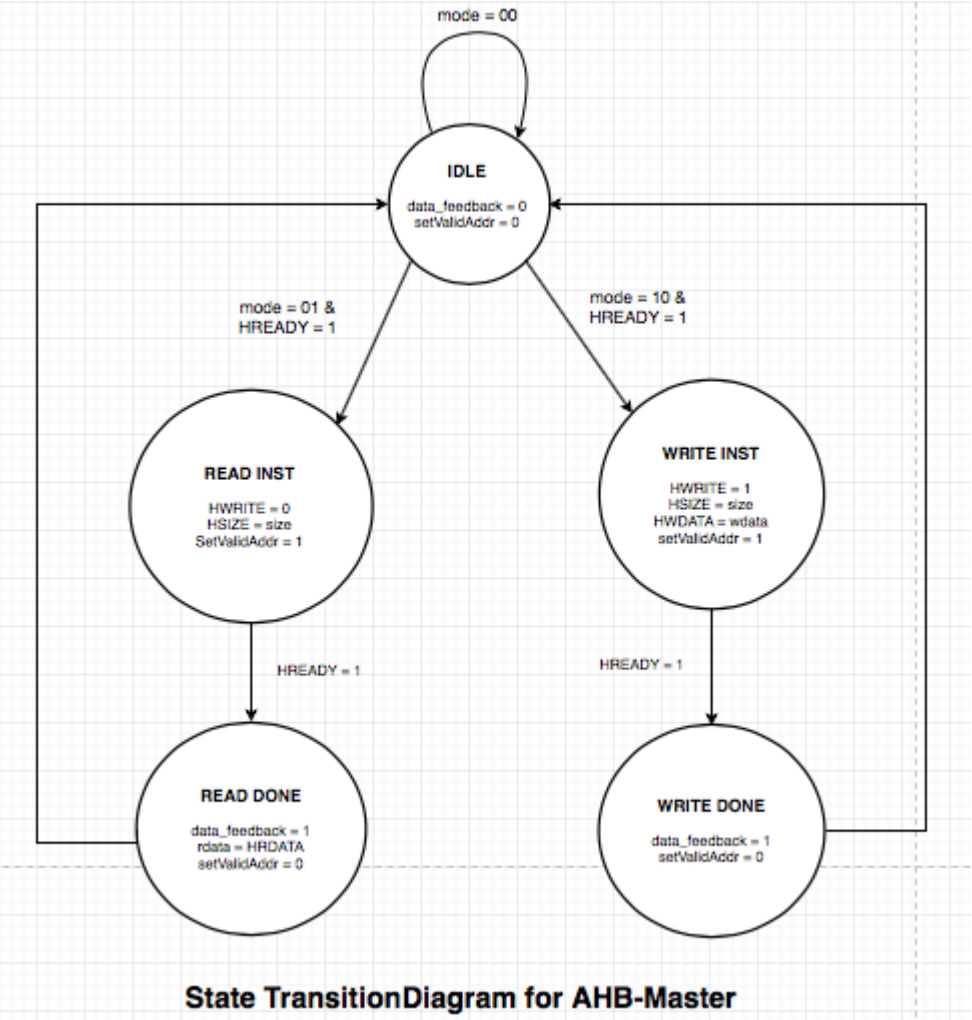
- **Block Diagram -**



Block Diagram for AHB-Lite Master

The image below above shows the block diagram for the AHB-Lite master within our design. It is just a single block with address and control signals going to SRAM and a data_feedback signal which tells that the transfer is complete. Also there is a rdata line which sends the data from read instruction to the module (RC4 or Edge-Detection) that asked for the data. Al lot of signals are coming to the block because signals like mode, pixNum, image_startAddr, size, image_Width, image_height and startAddr_sel help into setting the address and control signals. Wdata line sends the data from RC4 or Edge-detection to be written in SRAM. HCLK and HRESETn signals are coming from AHB-Lite slav3e so that our entire design is running on the same clock. The HREADY signal comes from the SRAM telling if the read or write instruction was accepted or not. The HRDATA is the line on which SRAM puts te data for AHB-Lite master to read.

- **State Transition Diagram -**



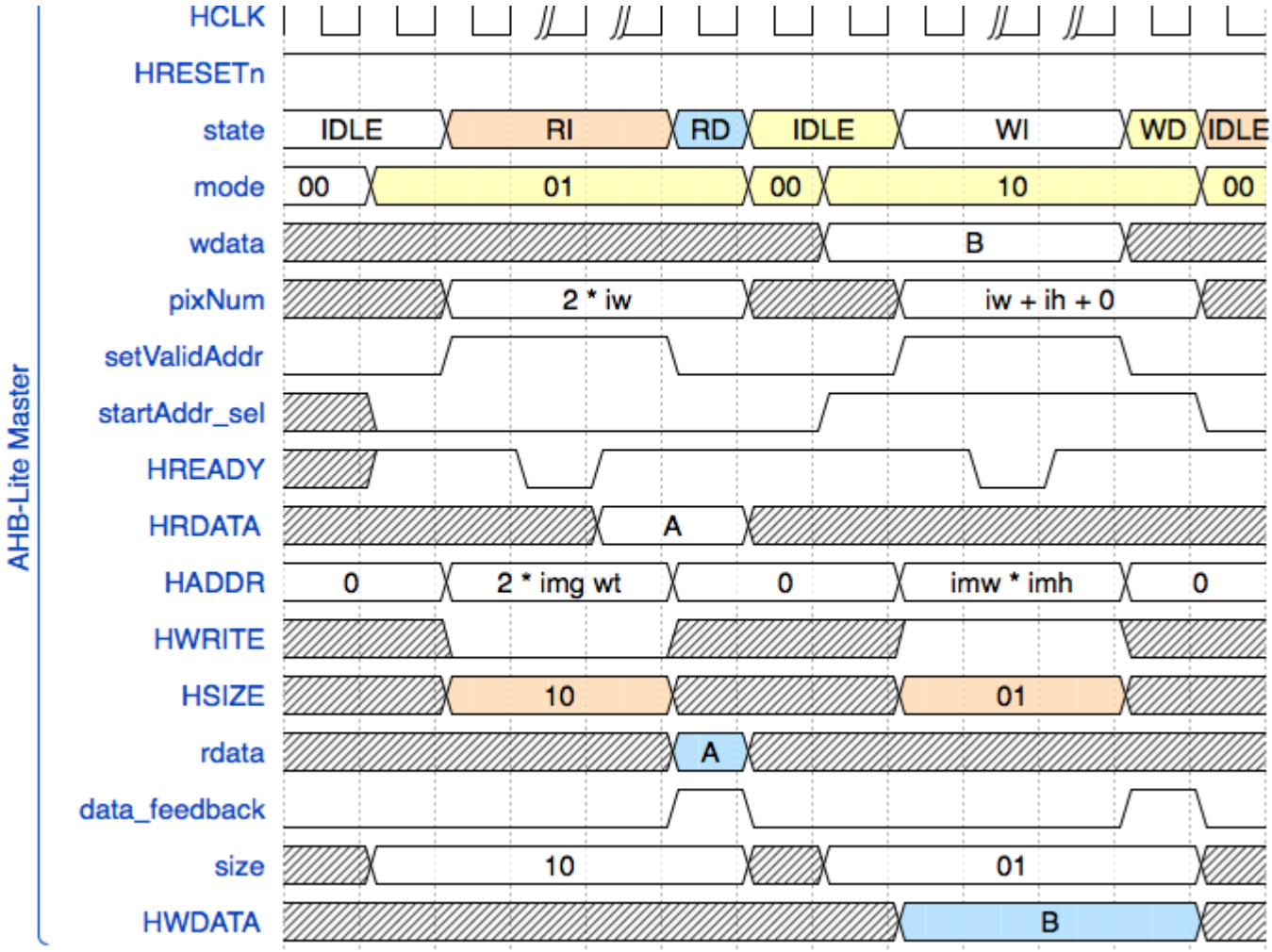
State TransitionDiagram for AHB-Master

The image above shows the state transition diagram for the FSM used in AHB-Lite Master. Initially it is in the IDLE state. The data_feedback and setValidAddr signals are set to 0 initially. If the mode is 01 and HREADY = 1 (which means the slave has not asked to be in wait state), it means one of the RC4 or Edge-Detection has asked for the AHB-Lite master to read data from the SRAM. The AHB-Lite master then sets the address and control signals for the SRAM based on the signals received from AHB-Helper module like HSIZE = size, setvalidAddr = 1 and HWRITE = 0 for read instruction. SRAM replies the Master by setting HREADY = 1 and when that happens, the FSM moves to next state called 'READ DONE' state. In this state, the AHB-Lite master sends a feedback to RC4 or Sample Image Data Storage module indirectly through AHB-Helper saying that it has acquired the required data. Note that in this case, setValidAddr selects the HADDR value but the constant offset of image size is not added since the startAddr_sel for a read instruction would take care of that.

However, if the mode is 10 and HREADY = 1, it means one of the RC4 or Edge-Detection has asked for the AHB-Lite master to write the data to the SRAM. Now, based on the signals sent by the AHB helper module, the master sets the address and control signals for the SRAM i.e. HWRITE = 1, HSIZE = size, HWDATA = wdata and setValidAddr = 1. It is important to note here that setValidAddr = 1 gives us the offset of the entire image since we don't want to write

the edge-detected image over the decrypted image already stored within the SRAM. Overwriting the decrypted would result in corruption of data and the final edge-detected image would not be as expected. Once the write instruction is sent to SRAM, if the HREADY = 1, FSM transitions to next state which is the 'WRITE DONE' done. Now the AHB-Lite master sends a data feedback to the AHB-Helper module which further passes the message to RC4 or Edge-Detection module that the write has finished and it was successful.

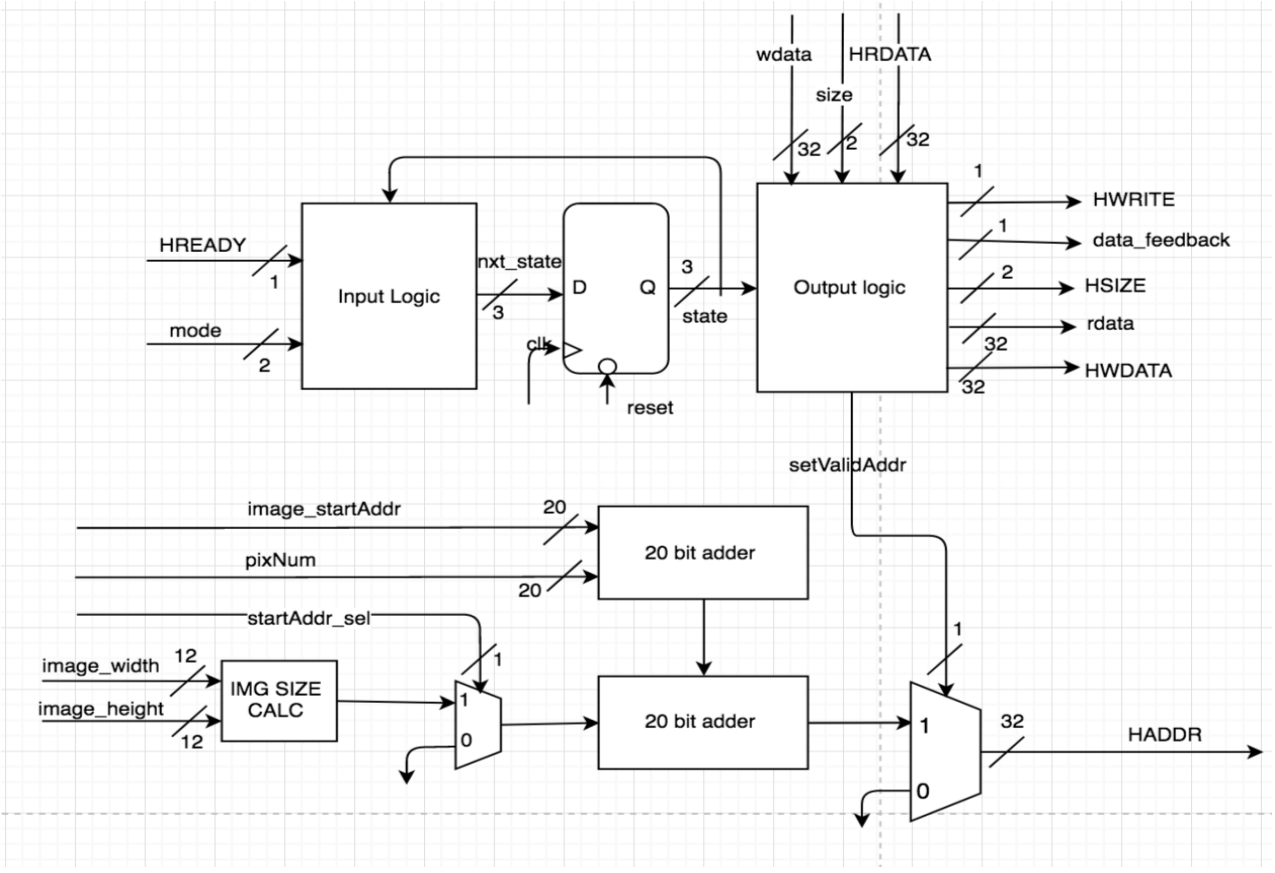
- **Timing Waveform Diagram -**



RI: Read Instruction
RD: Read Done
WI: Write Instruction
WD: Write Done

AHB-Lite master only takes care of read and write transfers to and from the SRAM. The above image shows the timing waveform diagram for the AHB-Lite master module. The address and control signals are set based on the signals sent by the AHB Helper. We have shown a read first and then a write of the data for the Edge-Detection module. The first part of the RI is the address phase while the second part of it is the data phase. In the address phase, Master selects the SRAM as slave and tells what it has to do - read or write. The size of the transaction - HSIZE is also selected based on read or write, a read is a 4 byte transaction and a write is a 2-byte transaction.
In the data phase, for the read transaction, the SRAM puts the data for the AHB-master to read ie A in the above image. For the write transaction, the AHB-Master puts the data on the HWDATA for the SRAM to write ie B in the above image. Here the startAddrSel line is set to 1 only during the write and not read is because a read transfer reads from the decrypted image but a write instruction writes to the address with an offset of image size.

- **RTL Diagram -**



The image above shows the RTL diagram for the AHB-Lite master within our design. The module has an FSM with some muxes and 2 20-bit adders to get the final HADDR value. There is also a small combinational block that does the product of image_Width and image_Height since this product would be used to get the offset in case the Edge-Detection module asks to write to SRAM. This offset is required since the decrypted image cannot be written over

because that would corrupt the image and would result in incorrect final edge-detected image. One thing to note here is that setValidAddr takes care of putting the correct HADDR for the instruction.

- **Area Estimation -**

Core Area Calculations				
Name of Block	**Category**	**Gate/FF Count**	**Area (um2)**	**Comments**
AHB-Lite Master Input Logic	Combinational	15	11,250	
AHB-Lite Master State Registers w/Reset	Reg. w/ Reset	3	7,200	
AHB-Lite Master Output Logic	Combinational	6	4,500	
AHB-Lite Master In-Module Muxes (20 2:1)	Combinational	80	60,000	
AHB-Lite Master In-module Muxes (32 2:1)	Combinational	128	7,500	
AHB-Lite Master (2 20-bit Adders)	Combinational	200	150,000	(Worst Case 20-bit Riple Carry Adders) but we would implement carry look ahead adder
AHB-Lite Master (12x12 Bit Multiplier)	Combinational	144	108,000	(if we use binary array multiplier)

The above table gives the area estimation of the AHB-Slave module. The area breakdown is as follows:

1) Input logic Combinational Area
2) Output Logic combinational Area
3) State registers w/ Reset Area
4) Area for 2:1 Muxes
5) Area for 2 20-bit adders
6) Area for 12x12 bit multiplier

This makes the total area of the module come to **348,450 micrometer squared**. The area is calculated based on estimating the number of input and output bits of each block mentioned above. With finding all the possible combinations of the inputs and outputs (input bits - 1) X (output bits), we find the number of gates for each block. With the gates of the block the area is calculated based on their category of combinational block and registers and found using known values. The area for muxes are added assuming that 1 2:1 Mux consists of 5 gates and that each bit would require 1 2:1 mux. A 20-bit adder was assumed to consist of 100 gates. This was done because adder for 1 bit would require 5 gates and considering a linear relation, 20-bit adder would require 20*5 = 100 gates. For a multiplier, the number of gates is a function of n^2. So, we assumed the number of gates to be number of bits squared. Hence, a 12x12bit multiplier would require 12^2 gates.

- **Actual Area Comparison with original Estimation -**

Estimated Area = 348,450 um^2

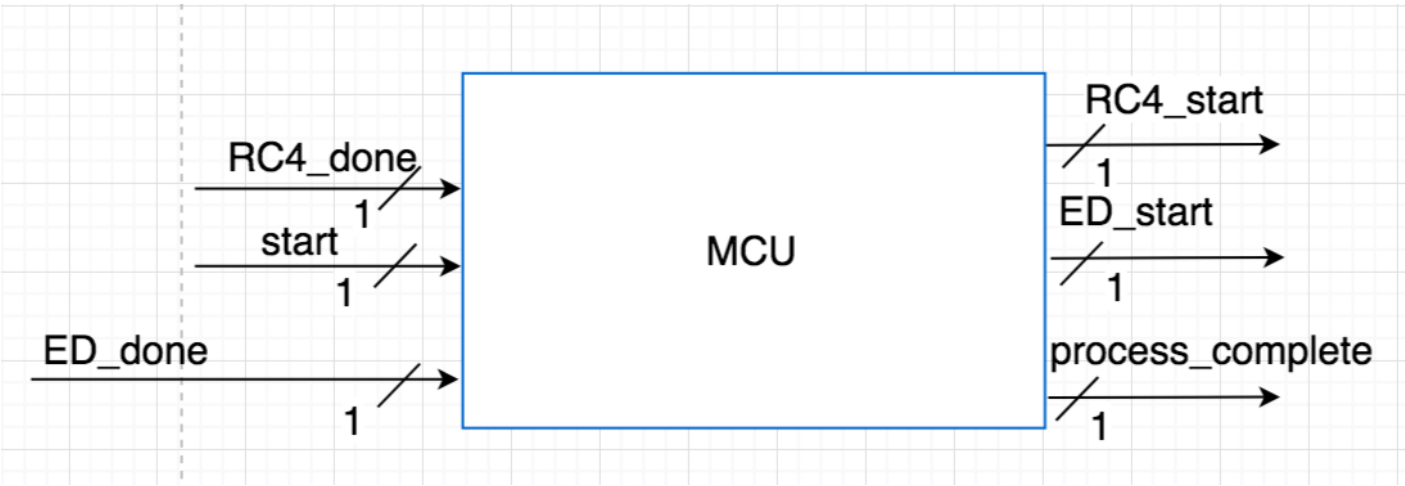
Actual Area = 238,635 um^2

AHB-Master was expected to have an area of 348,450 um^2 but we actually got it to be 238,635 um^2 which is 31% less than our estimated area. It is because of we assumed our adders to be carry ripple adder which adds a lot of area but how the design is actually synthesized with some level of optimization will be different than what we thought of. In addition, how the multiplier is synthesized is probably different than our estimated way, which was the worst case scenario. AHB-Master takes a total of 2.6% which is not a significant chunk of our design. The multiplier takes about 157,878 um^2 of area which is about 69% of the AHB-helper itself. In total, the actual area was significantly less than our estimated area.

MCU:

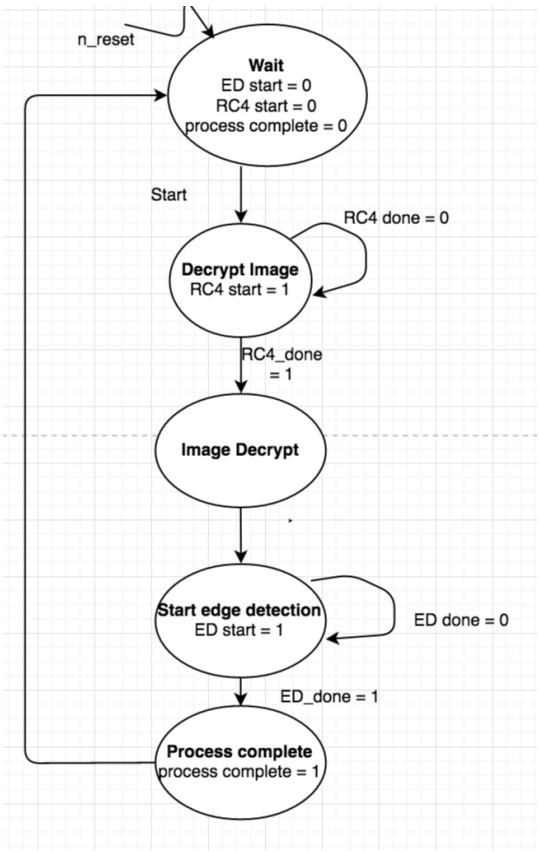
MCU is the central hub of our design. It controls which process the design needs to do currently. Once it gets a start signal, it first instructs the Rc4 module to complete the decryption of the supplied encrypted image and then instructs the Edge-Detection module to do the sobel edge-detection. The process complete signal goes through the MCU since it's the main control unit.

- **Block Diagram -**



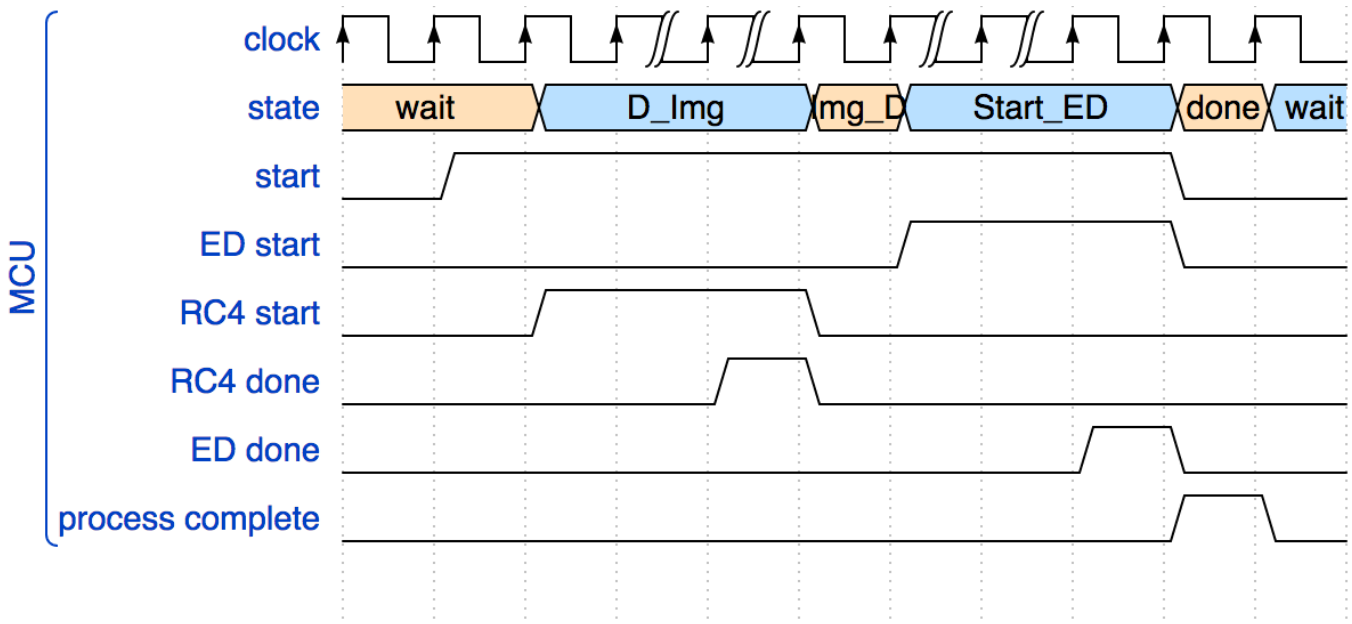
The MCU's block diagram is very simple. There are 3 inputs and 3 outputs. These consist of start, which begins operation within the MCU. RC4_start and ED_start which initiate the RC4 and Edge detection modules, RC4_done and RC4_done which informs the MCU that a certain task is complete, and finally process_complete which informs the AHB slave that all tasks have been completed.

- **State Transition Diagram -**



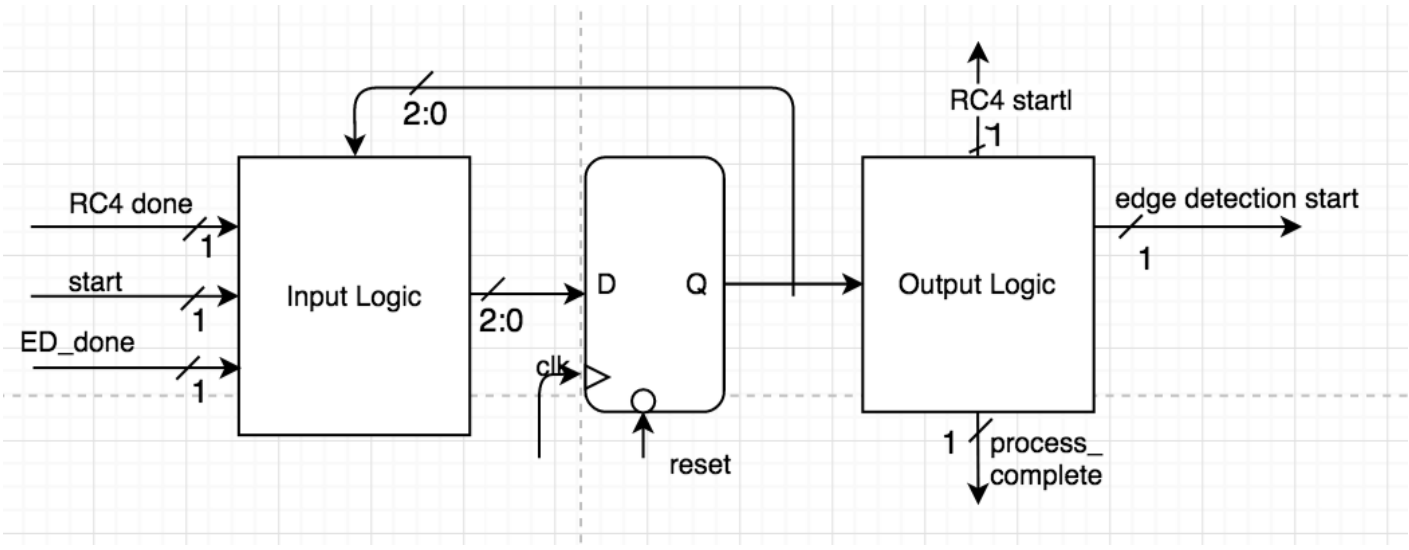
Start initiates the MCU, sending it to it's begin decryption state. In this state, RC4_start is asserted. When the RC4 module is done decrypting it sends back RC4_done. This sends the MCU to begin edge detection state, where it asserts ED_start and de-asserts RC4_start. Once Edge detection is complete, the Edge detection module sends back ED_done which informs the MCU that all tasks are complete. The MCU then moves to 'done' state, where it asserts process complete for one clock cycle.

- **Timing Waveform Diagram -**



Start initiates the MCU, sending it to it's begin decryption state. In this state, RC4_start is asserted. When the RC4 module is done decrypting it sends back RC4_done. This sends the MCU to begin edge detection state, where it asserts ED_start and de-asserts RC4_start. Once Edge detection is complete, the Edge detection module sends back ED_done which informs the MCU that all tasks are complete. The MCU then moves to 'done' state, where it asserts process complete for one clock cycle.

- **RTL Diagram -**



The RTL diagram for the MCU is very simple. It is a simple FSM, with three state registers.
There is no external combinational logic.

- **Area Estimation**

Core Area Calculations				
Name of Block	Category	Gate/FF Count	Area (um2)	Comments
MCU Input Logic	Combinational	15	11,250	
MCU State Registers w/ Reset	Reg. w/ Reset	3	7,200	
MCU Output Logic	Combinational	6	4,500	

The above table gives the area estimation of the MCU module. The area breakdown is as follows:

1) Input logic combinational area
2) Output logic combinational area
3) State register area

This makes the total area of the module come to **22,950 micrometer squared**. The area is calculated based on estimating the number of input and output bits of each block mentioned above. With finding all the possible combinations of the inputs and outputs (input bits - 1) X (output bits), we find the number of gates for each block. With the gates of the block the area is calculated based on their category of combinational block and registers and found using known values.

- **Actual Area Comparison with original Estimation -**

Estimated Area = 22,950 um^2

Actual Area = 10,836 um^2

The mcu takes 10,836 um^2 of area which is about 0.1% of the area of our design. The actual area is more than twice the estimated area primarily because of the difference in our estimated combinational area. We estimated it to be 15,750 um^2 of area but when we synthesized our design, we received it to be 6,084 um^2. We think it is because in the area difference in the input logic of the module. The non-combinational area was still within close proximity ot our estimation of 7,200 um^2 of area. In total, the actual area was about half of the estimated area due to the optimization during the synthesis of our design.

3.3 Design Timing Analysis

Per Path Timing Estimation Table

Starting Component	Propagation Delay	Combinational Logic	Propagation Delay	Ending Component	Setup Time or Propagation Delay	Total Path Delay	Target Clock Period
Edge-Detection state register	0.4 ns	Edge-Detection Output logic, Gradient Calculation, AHB-Helper (2 2:1 muxes), AHB-	33.57 ns	AHB-Master state register	0.2 ns	34.07 ns	35 ns

		Master Input Logic					
RC4 pseudo state register	0.4ns	RC4 Pseudo output logic, Pseudo Comb. block	19.99 ns	Counter J in RC4 Pseudo	0.2ns	20.59 ns	35 ns
AHB-Slave MM reg	0.4 ns	AHB-Master img size calc, Start Addr mux, 20-bit adder, valid addr mux	15.6 ns	SRAM	10 ns	26 ns	35 ns
RC4 pseudo state register	0.4 ns	RC4 Pseudo output logic, flex-counter, AHB-line control mux, AHBpix num mux, AHB-input logic	17.78 ns	AHB-Master state register	0.2 ns	18.38 ns	35 ns
AHB-Slave MM reg	0.4 ns	3:1 Mux, 1 20-bit adders, 2 pixNum 2:1 muxes, AHB master 2:1 Mux	15.6 ns	SRAM	10 ns	26 ns	35 ns

Description and Rationale

Path: Edge detection state reg -> AHB master state register

This is probably the most significant critical path in our design. This is due to the gradient calculations that need to be conducted in the edge detection module. For this path, the starting point was the output logic of the edge detection block. Once the output logic block sends an activation signal to the gradient calculation block, the gradient gets calculated. For this 24 20-bit adder are required. They will be organized as two sets of 1 bit adders in parallel. This calculates the gradient for two pixels. This is then sent out to the AHB helper, which again adds to mux-es worth of delay. This is then sent out to the AHB master where it is received as w_data. On calculating the delay for this, it was found to be about 34.07 ns. This is below our set clocking rate.

Path: RC4 pseudo register -> Counter j in RC4 Pseudo

One of the critical paths starts from RC4 pseudo-random state register. That is signal which is meant to signal to the combinational logic to accomplish: "j:= (j + S[i] + key[i mod keylength]) mod 256" operation. Since there are several ways to supply value to ncouterJ which are selected by an if statement we need 3-1 mux. We approximated the actual operation as 3-8-bit adders. After approximating the gates, we ended up with delay of 19.99ns.

Path: AHB-Slave MM reg for HADDR -> SRAM

This path is for the write instruction given by the Edge-detection module. For calculating the right value of the address that the Edge-Detection needs the master to write to, we need to get the image dimension directly from the AHB-Slave mm registers. These registers will have a propagation delay of 0.4 ns. However, both these dimensions go through a 12x12 multiplier (img size calc) which adds a lot to our delay since it would be 10 full adders in series(for the worst case we assumed full adders, we can use half adders reducing the delay). After going through this combination block, the address value goes through a 2:1 mux which we assumed to have a delay of 3 gates (a typical 2:1 mux has 5 gates

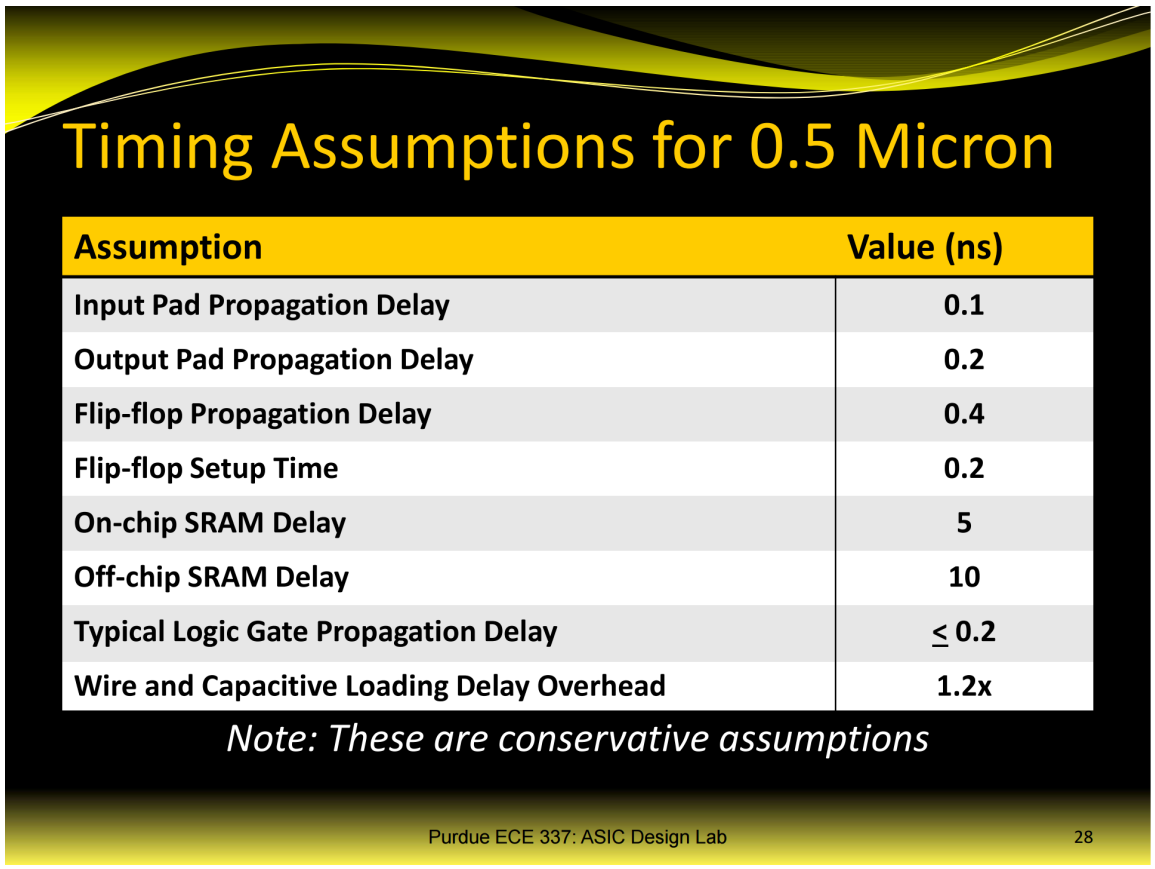
but the critical path involves only 3 gates). After putting the data onto the AHB-HADDR bus, it ends in the register value stored in the SRAM which has a delay of 10 ns. Summing all these delays, we get a total delay of 26ns well below the target clock period. Our calculations were based on the gate delays, propagation and setup delays for registers given in the lab.
.

Path: RC4 state register -> AHB master state register

This path is for the calculation of pixNum from the RC4 block. To calculate the pixel number, a signal is sent from the output logic of the RC4 pseudo block to 20-bit adder. This value is then sent to the AHB helper that adds 2 mux-es worth of delay. This is then sent into the AHB master. This delay was estimated to be about 18.38ns which is less than the clocking period.

Path: AHB Slave MM reg for pixNum -> SRAM

This path is for getting the address in the SRAM. This path goes through the Sample Image Storage module. Since the path starts from memory mapped registers, there is a propagation delay of 0.4 ns. While going through the Sample image storage module, it goes through 1 3:1 mux which we assumed to be a series of 2 2:1 mux. A 2:1 mux was assumed to have a delay of 3 gates (a typical 2:1 mux has 5 gates but the critical path involves only 3 gates), so a total of 6 gate delay would be present in 1 3:1 mux giving a delay of 6 * 0.2 = 1.2 ns. The data then goes into a 20-bit adder. 1-bit adder has a delay of 3 gates(based on the logic circuit). Considering 20-bit adder to be a series of 20 1-bit adders, the total delay would be of 20*3*0.2 = 12ns. This part of the path gives us the highest delay. Then, the data goes to another 3 2:1 muxes giving the delay of 1.8 ns. Summing everything, we get a total delay of 26ns. Our calculations were based on the gate delays, propagation and setup delays for registers given in the lab.



The table above shows all the possible path delays when finding the critical path. These were used in finding all the possible path delays in the timing estimation table.

Actual Critical Path

Path 1

```
Startpoint: DUT_ahbS/curr_DATA_reg[60]
            (rising edge-triggered flip-flop clocked by tb_clk)
Endpoint: DUT_EDM/state_reg[0]
          (rising edge-triggered flip-flop clocked by tb_clk)
Path Group: tb_clk
Path Type: max

Point                                                   Incr       Path
-----------------------------------------------------------------------------
clock tb_clk (rise edge)                                0.00       0.00
clock network delay (ideal)                             0.00       0.00
DUT_ahbS/curr_DATA_reg[60]/CLK (DFFSR)                  0.00 #     0.00 r
DUT_ahbS/curr_DATA_reg[60]/Q (DFFSR)                    0.49       0.49 f
DUT_ahbS/U160/Y (INVX2)                                 0.08       0.57 r
DUT_ahbS/U87/Y (INVX2)                                  0.13       0.71 f
DUT_ahbS/image_width[8] (AHB_slave)                     0.00       0.71 f
DUT_EDM/image_width[8] (edm)                            0.00       0.71 f
DUT_EDM/U222/Y (INVX4)                                  0.12       0.82 r



DUT_EDM/U1528/Y (AOI22X1)                               0.14      19.07 r
DUT_EDM/U1532/Y (NAND3X1)                               0.13      19.20 f
DUT_EDM/U1533/Y (XNOR2X1)                               0.19      19.39 f
DUT_EDM/U1540/Y (NAND2X1)                               0.11      19.50 r
DUT_EDM/U1578/Y (NOR2X1)                                0.17      19.67 f
DUT_EDM/U1579/Y (NAND3X1)                               0.15      19.82 r
DUT_EDM/U1586/Y (OAI21X1)                               0.08      19.89 f
DUT_EDM/state_reg[0]/D (DFFSR)                          0.00      19.89 f
data arrival time                                                 19.89

clock tb_clk (rise edge)                               20.00      20.00
clock network delay (ideal)                             0.00      20.00
DUT_EDM/state_reg[0]/CLK (DFFSR)                        0.00      20.00 r
library setup time                                     -0.10      19.90
data required time                                                19.90
-----------------------------------------------------------------------------
data required time                                                19.90
data arrival time                                                -19.89
-----------------------------------------------------------------------------
slack (MET)                                                       0.00
```

Path 2

```
Startpoint: DUT_ahbS/curr_DATA_reg[53]
            (rising edge-triggered flip-flop clocked by tb_clk)
Endpoint: DUT_EDM/state_reg[2]
          (rising edge-triggered flip-flop clocked by tb_clk)
Path Group: tb_clk
Path Type: max

Point                                                   Incr       Path
-----------------------------------------------------------------------------
clock tb_clk (rise edge)                                0.00       0.00
clock network delay (ideal)                             0.00       0.00
DUT_ahbS/curr_DATA_reg[53]/CLK (DFFSR)                  0.00 #     0.00 r
DUT_ahbS/curr_DATA_reg[53]/Q (DFFSR)                    0.41       0.41 r
DUT_ahbS/U75/Y (BUFX2)                                  0.23       0.64 r
DUT_ahbS/image_width[1] (AHB_slave)                     0.00       0.64 r
DUT_EDM/image_width[1] (edm)                            0.00       0.64 r
DUT_EDM/U359/Y (INVX4)                                  0.13       0.77 f
DUT_EDM/U774/Y (BUFX2)                                  0.19       0.96 f
                                                        [illegible]
DUT_EDM/U1419/Y (OAI21X1)                               0.16      18.88 f
DUT_EDM/U1588/Y (OAI21X1)                               0.17      19.05 r
DUT_EDM/U1589/Y (OAI21X1)                               0.08      19.13 f
DUT_EDM/state_reg[2]/D (DFFSR)                          0.00      19.13 f
data arrival time                                                 19.13

clock tb_clk (rise edge)                               20.00      20.00
clock network delay (ideal)                             0.00      20.00
DUT_EDM/state_reg[2]/CLK (DFFSR)                        0.00      20.00 r
library setup time                                     -0.11      19.89
data required time                                                19.89
-----------------------------------------------------------------------------
data required time                                                19.89
data arrival time                                                -19.13
-----------------------------------------------------------------------------
slack (MET)                                                       0.76
```

Path 3

```
Startpoint: DUT_ahbS/curr_DATA_reg[64]
            (rising edge-triggered flip-flop clocked by tb_clk)
Endpoint: DUT_RC4/DUT_core/DUT_STATE_MACH/cur_state_reg[1]
          (rising edge-triggered flip-flop clocked by tb_clk)
Path Group: tb_clk
Path Type: max

Point                                                       Incr       Path
-----------------------------------------------------------------------------
clock tb_clk (rise edge)                                    0.00       0.00
clock network delay (ideal)                                 0.00       0.00
DUT_ahbS/curr_DATA_reg[64]/CLK (DFFSR)                      0.00 #     0.00 r
DUT_ahbS/curr_DATA_reg[64]/Q (DFFSR)                        0.45       0.45 f
DUT_ahbS/U123/Y (BUFX2)                                     0.96       1.41 f
DUT_ahbS/image_height[0] (AHB_slave)                        0.00       1.41 f
DUT_RC4/img_hight_i[0] (RC4)                                0.00       1.41 f
DUT_RC4/DUT_core/img_hight_i[0] (RC4_CORE)                  0.00       1.41 f
DUT_RC4/DUT_core/DUT_STATE_MACH/img_hight_i[0] (RC4_CORE_STATE_MACH)
                                                            0.00       1.41 f
DUT_RC4/DUT_core/DUT_STATE_MACH/mult_200/a[0] (RC4_CORE_STATE_MACH_DW_mult_uns_0)
                                                            0.00       1.41 f
DUT_RC4/DUT_core/DUT_STATE_MACH/mult_200/U512/Y (INVX2)
                                                            0.76       2.17 r
DUT_RC4/DUT_core/DUT_STATE_MACH/mult_200/U602/Y (NOR2X1)
                                                            0.31       2.48 f


DUT_RC4/DUT_core/DUT_STATE_MACH/U61/Y (XOR2X1)               0.20      11.40 r
DUT_RC4/DUT_core/DUT_STATE_MACH/U59/Y (NOR2X1)               0.17      11.57 f
DUT_RC4/DUT_core/DUT_STATE_MACH/U54/Y (NAND3X1)              0.18      11.75 r
DUT_RC4/DUT_core/DUT_STATE_MACH/U45/Y (NOR2X1)               0.18      11.93 f
DUT_RC4/DUT_core/DUT_STATE_MACH/U44/Y (NAND3X1)              0.21      12.14 r
DUT_RC4/DUT_core/DUT_STATE_MACH/U43/Y (AOI22X1)              0.13      12.27 f
DUT_RC4/DUT_core/DUT_STATE_MACH/U38/Y (NAND3X1)              0.17      12.44 r
DUT_RC4/DUT_core/DUT_STATE_MACH/cur_state_reg[1]/D (DFFSR)
                                                             0.00      12.44 r
data arrival time                                                      12.44

clock tb_clk (rise edge)                                    20.00      20.00
clock network delay (ideal)                                  0.00      20.00
DUT_RC4/DUT_core/DUT_STATE_MACH/cur_state_reg[1]/CLK (DFFSR)
                                                             0.00      20.00 r
library setup time                                          -0.23      19.77
data required time                                                     19.77
-----------------------------------------------------------------------------
data required time                                                     19.77
data arrival time                                                     -12.44
-----------------------------------------------------------------------------
slack (MET)                                                            7.33
```

Path 4

```
Startpoint: DUT_ahbS/curr_DATA_reg[56]
            (rising edge-triggered flip-flop clocked by tb_clk)
Endpoint: DUT_RC4/DUT_core/DUT_STATE_MACH/cur_state_reg[1]
          (rising edge-triggered flip-flop clocked by tb_clk)
Path Group: tb_clk
Path Type: max

Point                                                          Incr       Path
-----------------------------------------------------------------------------
clock tb_clk (rise edge)                                       0.00       0.00
clock network delay (ideal)                                    0.00       0.00
DUT_ahbS/curr_DATA_reg[56]/CLK (DFFSR)                         0.00 #     0.00 r
DUT_ahbS/curr_DATA_reg[56]/Q (DFFSR)                           0.45       0.45 f
DUT_ahbS/U78/Y (BUFX2)                                         0.30       0.76 f
DUT_ahbS/image_width[4] (AHB_slave)                            0.00       0.76 f
U14/Y (BUFX2)                                                  0.54       1.30 f
DUT_RC4/img_width_i[4] (RC4)                                   0.00       1.30 f
DUT_RC4/DUT_core/img_width_i[4] (RC4_CORE)                     0.00       1.30 f
DUT_RC4/DUT_core/DUT_STATE_MACH/img_width_i[4] (RC4_CORE_STATE_MACH)
                                                               0.00       1.30 f
DUT_RC4/DUT_core/DUT_STATE_MACH/mult_200/b[4] (RC4_CORE_STATE_MACH_DW_mult_uns_0)
                                                               0.00       1.30 f
DUT_RC4/DUT_core/DUT_STATE_MACH/mult_200/U516/Y (INVX1)
                                                               0.89       2.19 r
DUT_RC4/DUT_core/DUT_STATE_MACH/mult_200/U604/Y (NOR2X1)
                                                               0.58       2.77 f
                                                                          .
                                                                          .
DUT_RC4/DUT_core/DUT_STATE_MACH/U45/Y (NOR2X1)                 0.18      11.88 f
DUT_RC4/DUT_core/DUT_STATE_MACH/U44/Y (NAND3X1)                0.21      12.09 r
DUT_RC4/DUT_core/DUT_STATE_MACH/U43/Y (AOI22X1)                0.13      12.21 f
DUT_RC4/DUT_core/DUT_STATE_MACH/U38/Y (NAND3X1)                0.17      12.39 r
DUT_RC4/DUT_core/DUT_STATE_MACH/cur_state_reg[1]/D (DFFSR)
                                                               0.00      12.39 r
data arrival time                                                        12.39

clock tb_clk (rise edge)                                      20.00      20.00
clock network delay (ideal)                                    0.00      20.00
DUT_RC4/DUT_core/DUT_STATE_MACH/cur_state_reg[1]/CLK (DFFSR)
                                                               0.00      20.00 r
library setup time                                            -0.23      19.77
data required time                                                       19.77
-----------------------------------------------------------------------------
data required time                                                       19.77
data arrival time                                                       -12.39
-----------------------------------------------------------------------------
slack (MET)                                                              7.39
```

Path 5

```
Startpoint: DUT_ahbS/curr_DATA_reg[56]
            (rising edge-triggered flip-flop clocked by tb_clk)
Endpoint: DUT_RC4/DUT_core/DUT_STATE_MACH/cur_state_reg[3]
          (rising edge-triggered flip-flop clocked by tb_clk)
Path Group: tb_clk
Path Type: max

Point                                                    Incr       Path
-----------------------------------------------------------------------------
clock tb_clk (rise edge)                                 0.00       0.00
clock network delay (ideal)                              0.00       0.00
DUT_ahbS/curr_DATA_reg[56]/CLK (DFFSR)                   0.00 #     0.00 r
DUT_ahbS/curr_DATA_reg[56]/Q (DFFSR)                     0.45       0.45 f
DUT_ahbS/U78/Y (BUFX2)                                   0.30       0.76 f
DUT_ahbS/image_width[4] (AHB_slave)                      0.00       0.76 f
U14/Y (BUFX2)                                            0.54       1.30 f
DUT_RC4/img_width_i[4] (RC4)                             0.00       1.30 f
DUT_RC4/DUT_core/img_width_i[4] (RC4_CORE)               0.00       1.30 f


DUT_RC4/DUT_core/DUT_STATE_MACH/U44/Y (NAND3X1)           0.21      12.09 r
DUT_RC4/DUT_core/DUT_STATE_MACH/U35/Y (OAI22X1)           0.18      12.26 f
DUT_RC4/DUT_core/DUT_STATE_MACH/U30/Y (NOR2X1)            0.12      12.38 r
DUT_RC4/DUT_core/DUT_STATE_MACH/U29/Y (NAND3X1)           0.05      12.43 f
DUT_RC4/DUT_core/DUT_STATE_MACH/cur_state_reg[3]/D (DFFSR)
                                                          0.00      12.43 f
data arrival time                                                   12.43

clock tb_clk (rise edge)                                 20.00      20.00
clock network delay (ideal)                               0.00      20.00
DUT_RC4/DUT_core/DUT_STATE_MACH/cur_state_reg[3]/CLK (DFFSR)
                                                          0.00      20.00 r
library setup time                                       -0.11      19.89
data required time                                                  19.89
-----------------------------------------------------------------------------
data required time                                                  19.89
data arrival time                                                  -12.43
-----------------------------------------------------------------------------
slack (MET)                                                         7.46
```

4. Success Criteria

4.1 Fixed Criteria:

1. (2 points) Must have test benches for all the top-level components and the entire design.
2. (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.
3. (2 points) IC layout produced from design must pass all the geometry and connectivity checks.
4. (2 points) The entire design complies with targets for area, pin count, throughput (if applicable), and clock rate.
5. (4 points) The entire design must synthesize completely, without any inferred latches, timing arcs, and sensitivity list warnings.

4.2 Fixed Success Criteria Summary and Status

1. 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 the functional requirements given in the design specific success criteria. (2 pts)
STATUS: Completed. All the top-level test benches exist, can be compiled and meet the functional requirement of the design.

2. 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. (2 pts)

STATUS: Completed. The source and mapped version have same behavior for the same test cases of the modules. The compilation is error free and all the test cases pass for both the source and mapped version.

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

STATUS: Completed. The layout of the chip is generated through the Innovus and virtuoso components. The layout meets the project specifications with the pins attached in the layout.

4. 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 if 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. (2 pts)

· Area: 4mm x 4mm

· Pin Count: 105 pins for each bit (power, ground, clock, reset, HSIZE_slave [1:0], HADDR [31:0], HWDATA [31:0], HRDATA_slave [31:0], HSIZE_slave [1:0], HWRITE_slave, HREADY, HRESP, HWRITE, HRESP_slave, RC4_done, process_complete, HADDR_slave [31:0], HWDATA_slave [31:0], HRDATA [31:0])

· Clock Period: 50 MHz

STATUS: Completed. The layout depicts the properties with the mapped log file.

5. Entire design synthesizes completely, without any inferred latches, timing arcs, and, sensitivity list warnings.

STATUS: Completed. Compilation occurs and meets specifications without any latches, timing arcs and sensitivity list warnings.

4.3 Supporting Proof for Fixed Success Criteria Status

Proof for Fixed Success Criteria 1:

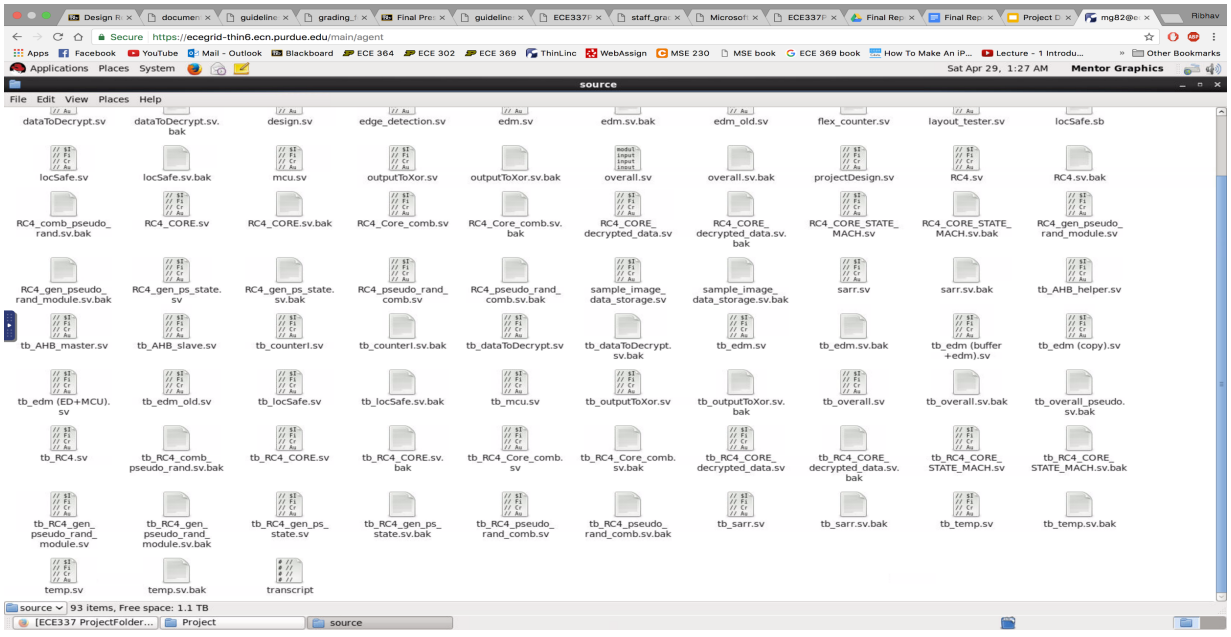


Fig: The above figure shows the presence of the testbenches for all the top level modules and even the sub modules.

All the testbenches reside in the source folder and are present to check the individual functionality of each module and provide the correct inferences to the expected outcome.

Proof for Fixed success criteria 3 and 4:

```
Library(s) Used:

    osu05_stdcells (File: /package/eda/cells/OSU/v2.7/synopsys/lib/ami05/osu05_stdcells.db)

Number of ports:                         6116
Number of nets:                         33325
Number of cells:                        27684
Number of combinational cells:          24576
Number of sequential cells:              2997
Number of macros/black boxes:               0
Number of buf/inv:                       4235
Number of references:                      10

Combinational area:            6573888.000000
Buf/Inv area:                   728208.000000
Noncombinational area:         2433744.000000
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:               9007632.000000
Total area:                 undefined
```

Fig: The above figure shows the area specification of the overall design generated.

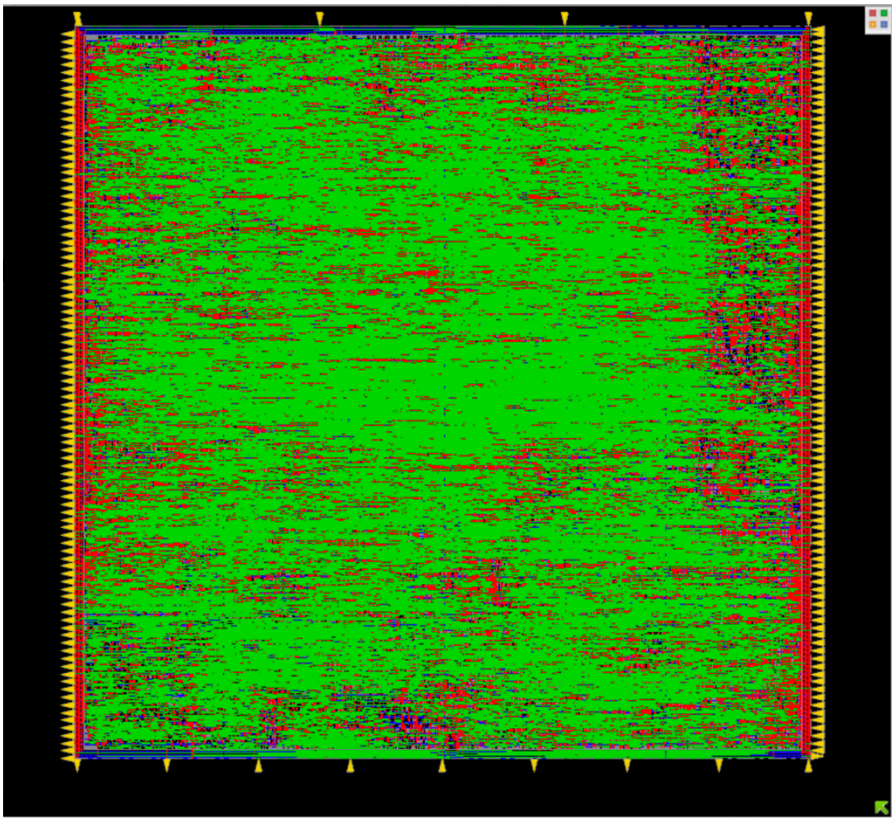


Fig: The above figure is the layout through innovus to demonstrate interconnectivity and layout mapping with pins

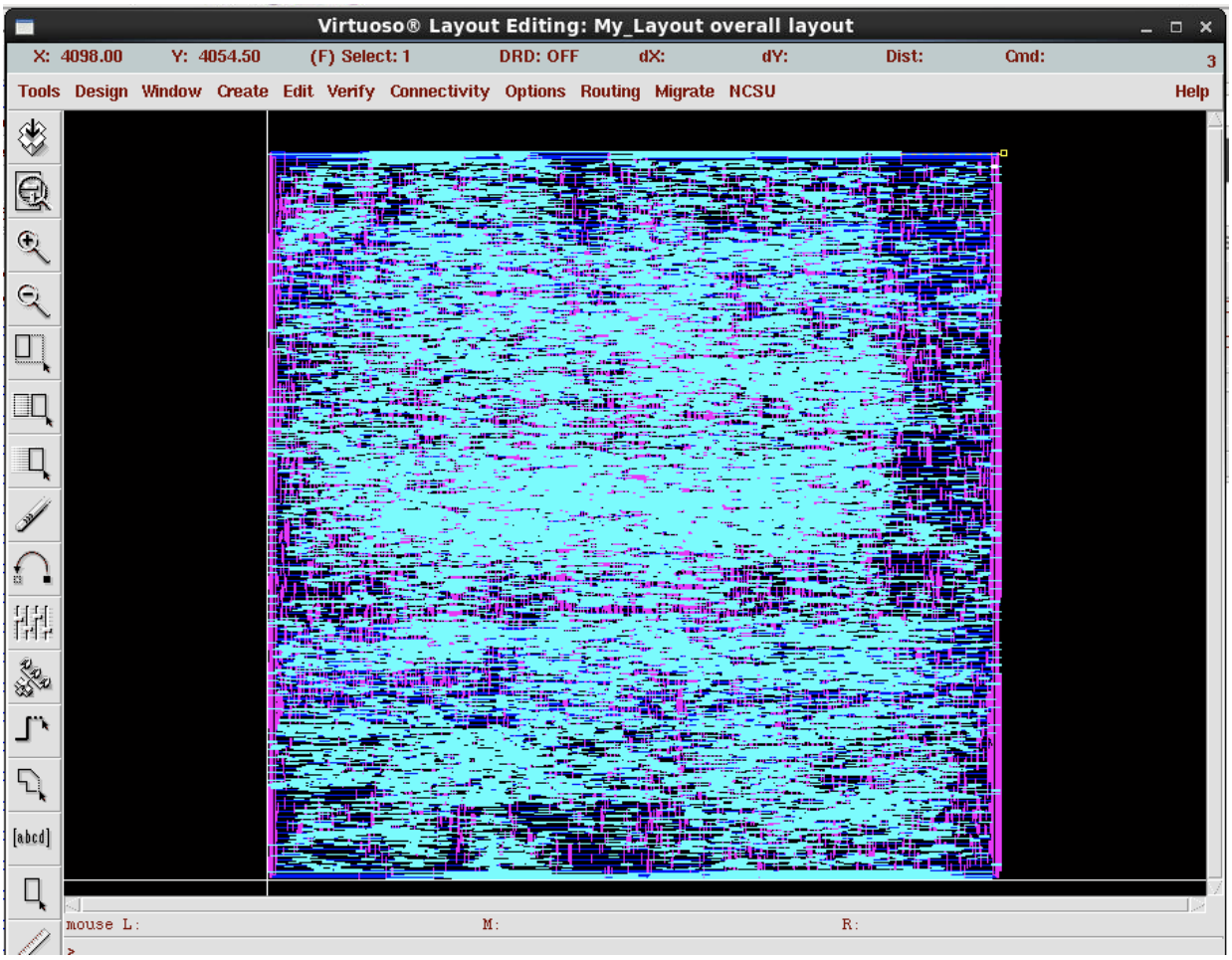


Fig: The above figure shows the layout through Virtuoso, also it depicts the chip area on the top left corner with pins.

The area of the chip designed was within the specifications of the project criteria and this can be depicted in the above figure which showed the mapped report containing the area estimation. Our clock was up to the standards of 50 MHz which was more than our expected clock rate. The IC layout shows the chip layout with the pin area and the figure above demonstrates that our chip with the pin out was close to 4mm x 4mm which was our expected area. The area can be justified due to the presence of the large state array required by the RC4 decryption implementation of the design.

Proof for Fixed Criteria 2 and 5:

```
Creating work library: source_work
Compiling 'source/overall.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:15 on Apr 29,2017
vlog -sv -work source_work source/overall.sv
-- Compiling module overall

Top level modules:
        overall
End time: 01:59:15 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/overall.sv' into work library 'source_work'
Compiling 'source/tb_overall.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:15 on Apr 29,2017
vlog -sv -work source_work source/tb_overall.sv
-- Compiling module tb_overall

Top level modules:
        tb_overall
End time: 01:59:15 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/tb_overall.sv' into work library 'source_work'
Compiling 'source/RC4_pseudo_rand_comb.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:15 on Apr 29,2017
vlog -sv -work source_work source/RC4_pseudo_rand_comb.sv
-- Compiling module RC4_pseudo_rand_comb

Top level modules:
        RC4_pseudo_rand_comb
End time: 01:59:15 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/RC4_pseudo_rand_comb.sv' into work library 'source_work'
Compiling 'source/RC4_gen_ps_state.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:15 on Apr 29,2017
vlog -sv -work source_work source/RC4_gen_ps_state.sv
-- Compiling module RC4_gen_ps_state

Top level modules:
        RC4_gen_ps_state
End time: 01:59:15 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/RC4_gen_ps_state.sv' into work library 'source_work'
Compiling 'source/sarr.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:15 on Apr 29,2017
vlog -sv -work source_work source/sarr.sv
-- Compiling module sarr

Top level modules:
        sarr
End time: 01:59:15 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/sarr.sv' into work library 'source_work'
Compiling 'source/counterI.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:15 on Apr 29,2017
vlog -sv -work source_work source/counterI.sv
-- Compiling module counterI

Top level modules:
        counterI
End time: 01:59:15 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/counterI.sv' into work library 'source_work'
Compiling 'source/locSafe.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:15 on Apr 29,2017
vlog -sv -work source_work source/locSafe.sv
-- Compiling module locSafe

Top level modules:
        locSafe
End time: 01:59:15 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/locSafe.sv' into work library 'source_work'
Compiling 'source/temp.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:15 on Apr 29,2017
vlog -sv -work source_work source/temp.sv
-- Compiling module temp

Top level modules:
        temp
End time: 01:59:15 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/temp.sv' into work library 'source_work'
Compiling 'source/outputToXor.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:15 on Apr 29,2017
vlog -sv -work source_work source/outputToXor.sv
-- Compiling module outputToXor

Top level modules:
        outputToXor
End time: 01:59:15 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/outputToXor.sv' into work library 'source_work'
Compiling 'source/RC4_CORE_STATE_MACH.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:15 on Apr 29,2017
vlog -sv -work source_work source/RC4_CORE_STATE_MACH.sv
-- Compiling module RC4_CORE_STATE_MACH

Top level modules:
        RC4_CORE_STATE_MACH
End time: 01:59:15 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/RC4_CORE_STATE_MACH.sv' into work library 'source_work'
Compiling 'source/dataToDecrypt.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:15 on Apr 29,2017
vlog -sv -work source_work source/dataToDecrypt.sv
-- Compiling module dataToDecrypt
```

```
Top level modules:
        RC4_CORE_decrypted_data
End time: 01:59:16 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/RC4_CORE_decrypted_data.sv' into work library 'source_work'
Compiling 'source/counterPixel.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:16 on Apr 29,2017
vlog -sv -work source_work source/counterPixel.sv
-- Compiling module counterPixel

Top level modules:
        counterPixel
End time: 01:59:16 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/counterPixel.sv' into work library 'source_work'
Compiling 'source/RC4.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:16 on Apr 29,2017
vlog -sv -work source_work source/RC4.sv
-- Compiling module RC4

Top level modules:
        RC4
End time: 01:59:16 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/RC4.sv' into work library 'source_work'
Compiling 'source/RC4_CORE.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:16 on Apr 29,2017
vlog -sv -work source_work source/RC4_CORE.sv
-- Compiling module RC4_CORE

Top level modules:
        RC4_CORE
End time: 01:59:16 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/RC4_CORE.sv' into work library 'source_work'
Compiling 'source/RC4_gen_pseudo_rand_module.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:16 on Apr 29,2017
vlog -sv -work source_work source/RC4_gen_pseudo_rand_module.sv
-- Compiling module RC4_gen_pseudo_rand_module

Top level modules:
        RC4_gen_pseudo_rand_module
End time: 01:59:16 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/RC4_gen_pseudo_rand_module.sv' into work library 'source_work'
Compiling 'source/edm.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:16 on Apr 29,2017
vlog -sv -work source_work source/edm.sv
-- Compiling module edm

Top level modules:
        edm
End time: 01:59:16 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/edm.sv' into work library 'source_work'
Compiling 'source/sample_image_data_storage.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:16 on Apr 29,2017
vlog -sv -work source_work source/sample_image_data_storage.sv
-- Compiling module sample_image_data_storage

Top level modules:
        sample_image_data_storage
End time: 01:59:16 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/sample_image_data_storage.sv' into work library 'source_work'
Compiling 'source/AHB_master.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:16 on Apr 29,2017
vlog -sv -work source_work source/AHB_master.sv
-- Compiling module AHB_master

Top level modules:
        AHB_master
End time: 01:59:16 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/AHB_master.sv' into work library 'source_work'
Compiling 'source/AHB_slave.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:16 on Apr 29,2017
vlog -sv -work source_work source/AHB_slave.sv
-- Compiling module AHB_slave

Top level modules:
        AHB_slave
End time: 01:59:16 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/AHB_slave.sv' into work library 'source_work'
Compiling 'source/AHB_helper.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:16 on Apr 29,2017
vlog -sv -work source_work source/AHB_helper.sv
-- Compiling module AHB_helper

Top level modules:
        AHB_helper
End time: 01:59:16 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/AHB_helper.sv' into work library 'source_work'
Compiling 'source/mcu.sv' into work library 'source_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 01:59:16 on Apr 29,2017
vlog -sv -work source_work source/mcu.sv
-- Compiling module mcu

Top level modules:
        mcu
End time: 01:59:16 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/mcu.sv' into work library 'source_work'
Simulating Source Design
Reading pref.tcl
Done simulating the source design
```

Fig: The figures above show that the code compiles without any errors for the source version of the design.

```
Creating work library: mapped_work
Compiling 'mapped/overall.v' into work library 'mapped_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 02:19:06 on Apr 29,2017
vlog -sv -work mapped_work mapped/overall.v
-- Compiling module edm_DW01_add_0
-- Compiling module edm_DW01_add_1
-- Compiling module edm_DW01_add_2
-- Compiling module edm_DW01_add_3
-- Compiling module edm_DW_cmp_4
-- Compiling module edm_DW01_sub_4
-- Compiling module edm_DW_mult_uns_0
-- Compiling module edm_DW01_sub_10
-- Compiling module edm_DW01_sub_9
-- Compiling module edm_DW01_sub_8
-- Compiling module edm_DW01_add_22
-- Compiling module edm_DW01_add_21
-- Compiling module edm_DW01_sub_15
-- Compiling module edm_DW01_sub_14
-- Compiling module edm_DW01_add_26
-- Compiling module edm_DW01_add_25
-- Compiling module edm_DW01_sub_17
-- Compiling module edm_DW01_sub_16
-- Compiling module edm_DW01_add_28
-- Compiling module edm_DW01_add_27
-- Compiling module edm_DW01_sub_13
-- Compiling module edm_DW01_sub_12
-- Compiling module edm_DW01_sub_11
-- Compiling module edm_DW01_add_24
-- Compiling module edm_DW01_add_23
-- Compiling module edm_DW01_add_73
-- Compiling module edm_DW01_add_71
-- Compiling module edm_DW01_add_69
-- Compiling module edm_DW01_add_67
-- Compiling module edm_DW01_add_60
-- Compiling module edm_DW01_add_45
-- Compiling module edm_DW01_add_43
-- Compiling module edm_DW01_add_41
-- Compiling module edm_DW01_add_39
-- Compiling module edm_DW01_add_33
-- Compiling module edm_DW01_add_32
-- Compiling module edm_DW01_sub_18
-- Compiling module edm_DW01_add_31
-- Compiling module edm_DW01_add_30
-- Compiling module edm_DW01_add_85
-- Compiling module edm_DW01_add_86
-- Compiling module edm_DW01_add_82
-- Compiling module edm_DW01_add_84
-- Compiling module edm_DW01_add_89
-- Compiling module edm_DW01_add_92
-- Compiling module edm_DW01_add_93
-- Compiling module edm_DW01_add_96
-- Compiling module edm_DW01_add_97
-- Compiling module edm_DW01_add_90
-- Compiling module edm_DW01_add_100
-- Compiling module edm_DW01_add_101
-- Compiling module edm_DW01_add_102
-- Compiling module edm_DW01_add_104
-- Compiling module edm_DW01_add_105
-- Compiling module edm_DW01_add_106
-- Compiling module edm_DW01_add_107
-- Compiling module edm_DW01_add_87
-- Compiling module edm_DW01_add_137
-- Compiling module edm_DW01_add_139
-- Compiling module edm_DW01_add_144
-- Compiling module edm_DW01_add_142
-- Compiling module edm_DW01_add_152
-- Compiling module edm_DW01_add_148
-- Compiling module edm_DW01_add_143
-- Compiling module edm_DW01_add_147
-- Compiling module edm_DW01_add_134
-- Compiling module edm_DW01_add_136
-- Compiling module edm_DW01_add_126
-- Compiling module edm_DW01_add_135
-- Compiling module edm_DW01_add_141
-- Compiling module edm_DW01_add_128
-- Compiling module edm_DW01_add_151
-- Compiling module edm_DW01_add_155
-- Compiling module edm_DW01_add_133
-- Compiling module edm
-- Compiling module sample_image_data_storage_DW01_add_0
-- Compiling module sample_image_data_storage_DW01_add_1
-- Compiling module sample_image_data_storage
-- Compiling module mcu
-- Compiling module AHB_helper
-- Compiling module AHB_master_DW_mult_uns_1
-- Compiling module AHB_master_DW01_add_1
-- Compiling module AHB_master_DW01_add_0
-- Compiling module AHB_master
-- Compiling module AHB_slave
-- Compiling module RC4_CORE_STATE_MACH_DW_mult_uns_0
-- Compiling module RC4_CORE_STATE_MACH
-- Compiling module dataToDecrypt
-- Compiling module RC4_Core_comb
-- Compiling module RC4_CORE_decrypted_data
-- Compiling module counterPixel_DW01_inc_0
-- Compiling module counterPixel_DW01_inc_1
-- Compiling module counterPixel
-- Compiling module RC4_CORE
-- Compiling module RC4_pseudo_rand_comb_DW01_add_0
-- Compiling module RC4_pseudo_rand_comb_DW01_add_1
-- Compiling module RC4_pseudo_rand_comb_DW01_add_2
-- Compiling module RC4_pseudo_rand_comb_DW01_add_3
-- Compiling module RC4_pseudo_rand_comb_DW01_add_4
-- Compiling module RC4_pseudo_rand_comb_DW01_add_5
-- Compiling module RC4_pseudo_rand_comb
-- Compiling module RC4_gen_ps_state
-- Compiling module sarr
-- Compiling module counterI_DW01_inc_0
-- Compiling module counterI_DW01_inc_1
-- Compiling module counterI
-- Compiling module locSafe
-- Compiling module temp
-- Compiling module outputToXor
-- Compiling module RC4_gen_pseudo_rand_module
-- Compiling module RC4
-- Compiling module overall

Top level modules:
        overall
End time: 02:19:07 on Apr 29,2017, Elapsed time: 0:00:01
Errors: 0, Warnings: 0
Done compiling 'mapped/overall.v' into work library 'mapped_work'
Compiling 'source/tb_overall.sv' into work library 'mapped_work'
QuestaSim-64 vlog 10.3b Compiler 2014.05 May 29 2014
Start time: 02:19:07 on Apr 29,2017
vlog -sv -work mapped_work source/tb_overall.sv
-- Compiling module tb_overall

Top level modules:
        tb_overall
End time: 02:19:07 on Apr 29,2017, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
Done compiling 'source/tb_overall.sv' into work library 'mapped_work'
Simulating Mapped Design
Reading pref.tcl
Done simulating the mapped design
```

Fig: The above figure shows the mapped version compilation of the entire design.

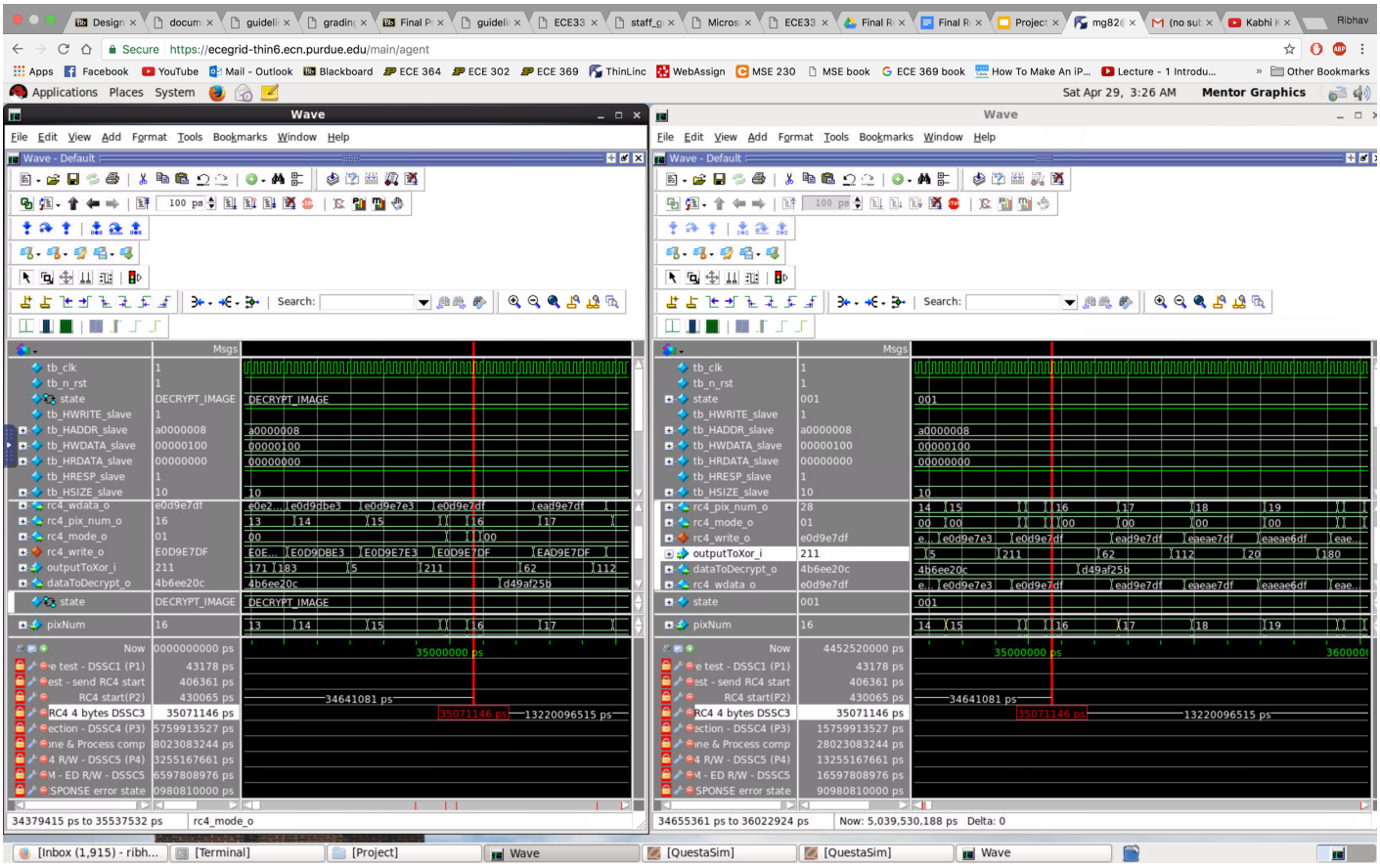


Fig: Similar behavior of Source and Mapped versions.

The figure above demonstrates similar behavior for the source and mapped compilation for the same time at 3.4 ms depicting the similar data values and assertions in signals at that time.

4.4 Design Specific Success Criteria

1. Demonstrate by simulation of a Verilog test bench that decryption and edge-detection do not begin until all user inputs have been received.
2. Demonstrate by simulation of a Verilog test bench that our module works correctly for variable image size.

3. Demonstrate by simulation of a Verilog test bench that image obtained after RC4 decryption matches the expected output.
4. Demonstrate by simulation of a Verilog test bench that Sobel Edge detection gives a visible and convincing image.
5. Demonstrate by simulation of a Verilog test bench that data transmitted serially on the AHB-LITE bus is reading, writing and waiting correctly.

4.5 Point Allocation for Design Specific Success Criteria

1. Demonstrate by simulation of a Verilog test bench that decryption and edge-detection do not begin until all user inputs have been received: **1 point**
2. Demonstrate by simulation of a Verilog test bench that our module works correctly for variable image size: **1 point**
3. Demonstrate by simulation of a Verilog test bench that image obtained after RC4 decryption matches the expected output: **2 point**
4. Demonstrate by simulation of a Verilog test bench that Sobel Edge detection gives a visible and convincing image: **2 point**
5. Demonstrate by simulation of a Verilog test bench that data transmitted serially on the AHB-LITE bus is reading, writing and waiting correctly: **2 point**

4.6 Design Specific Success Criteria Summary and Status

1. Demonstrate by simulation of a Verilog test bench that decryption and edge-detection do not begin until all user inputs have been received. This design specific criteria depicts that all the modules start functioning at the correct time and have the correct values to start functioning.

   STATUS: Completed. The generated waveform shows that all the data from the user inputs is received and then the edge detection and RC4 start.

2. Demonstrate by simulation of a Verilog test bench that our module works correctly for variable image sizes. The overall testbench works with images of two different sizes and the outputs for both the images a decrypted image and an edge detected image.

   STATUS: Completed. Compiled code for variable sized imaged and received expected correct output.

3. Demonstrate by simulation of a Verilog test bench that image obtained after RC4 decryption is like the expected output. The RC4 decryption works on the encrypted image to produce an image which is understandable out of gibberish.

   STATUS: Completed. The generated images show the decrypted RC4 images which are like the original image.

4. Demonstrate by simulation of a Verilog test bench that Sobel Edge detection gives a visible and convincing image. The Edge detected image is the gradient defined image of the original picture.

   STATUS: Completed. The generated images were convincingly edge detected.

5. Demonstrate by simulation of a Verilog test bench that data transmitted serially on the AHB-LITE bus is reading, writing and waiting correctly. The AHB signals are asserted at the right time to propagate correct data to the individual modules and relay their data to the SRAM.

STATUS : Completed. The images were generated correctly defining the correct functionality of the AHB bus.

5. Design Verification

The ASIC designed by our team consists of 7 major components. These are -

- The AHB-slave
- AHB-master
- AHB-helper
- Main Control Unit (MCU)
- Edge Detection Module (EDM)
- Sample Image Data Storage
- RC4 module

While designing our chip, we tested each of these modules individually, and incrementally added them one by one to the overall testbench. The exact method of testing will be documented in further detail in this section.

5.1 Design Verification Overview

What to Verify	Design Module(s) involved	Verification Procedure Summary	DSSC(s) Proved	Use in Final Demo	Comments
Chip holds processes until user inputs have been received	Top Level	Test bench to monitor AHB-slave MM-registers and MCU outputs	DSSC 1	Yes	Use temporary registers in test bench to grab user inputs and check MCU outputs
Correctness of Edge-detection on variable image sizes	Top Level	Compare results to an alternate implementation	DSSC 2	Yes	Test images are of different sizes. Should be able to reuse most of the test benches

Correctness of RC4 Decryption	Top Level	Compare decrypted results to original image	DSSC 3	Yes	Image is sent to python implementation of RC4 encryption whose output is used as input to our RC4 decryption. The output is compared to original image.
Correctness of Sobel Edge-Detection	Top Level	Compare edge detection results to an alternate implementation	DSSC 4	Yes	Sobel Edge-detected image is compared with the output of python implementation of Sobel edge-detection
AMBA AHB-Lite Protocol interfacing	Top Level	Test Bench provides samples of each type of bus transaction	DSSC 5	Yes	Should be able to reuse test bench code from AMBA bus interface controller

\

5.2 Test Scenario Breakouts

Our design was such that if the final design produced the required results, it almost guaranteed that each individual module was functioning correctly on its own. The following paragraphs will contain details on how each design specific success criteria were proven.

AHB COMMUNICATION (SLAVE + MASTER) TESTS

Testing of AHB slave-

DSSC 1: Demonstrate by simulation of a Verilog test bench that decryption and edge-detection do not begin until all user inputs have been received. This design specific criteria depicts that all the modules start functioning at the correct time and have the correct values to start functioning.

This test is essential because if the process begins without waiting for ALL the inputs to be received, the outputted results will be completely wrong. The inputs include the RC4 key, Image width, Image height and the image starting address in the SRAM. This criterion largely tests the correctness of the AHB slave, and the MCU.

To test that this is functioning correctly two things need to be checked -

1) All four inputs need to be received before the AHB slave sends the 'start' signal
2) The MCU holds the RC4 start and Edge Detection Start at low until the start signal is received.

The results for this test case can clearly be seen in the following waveform diagram.

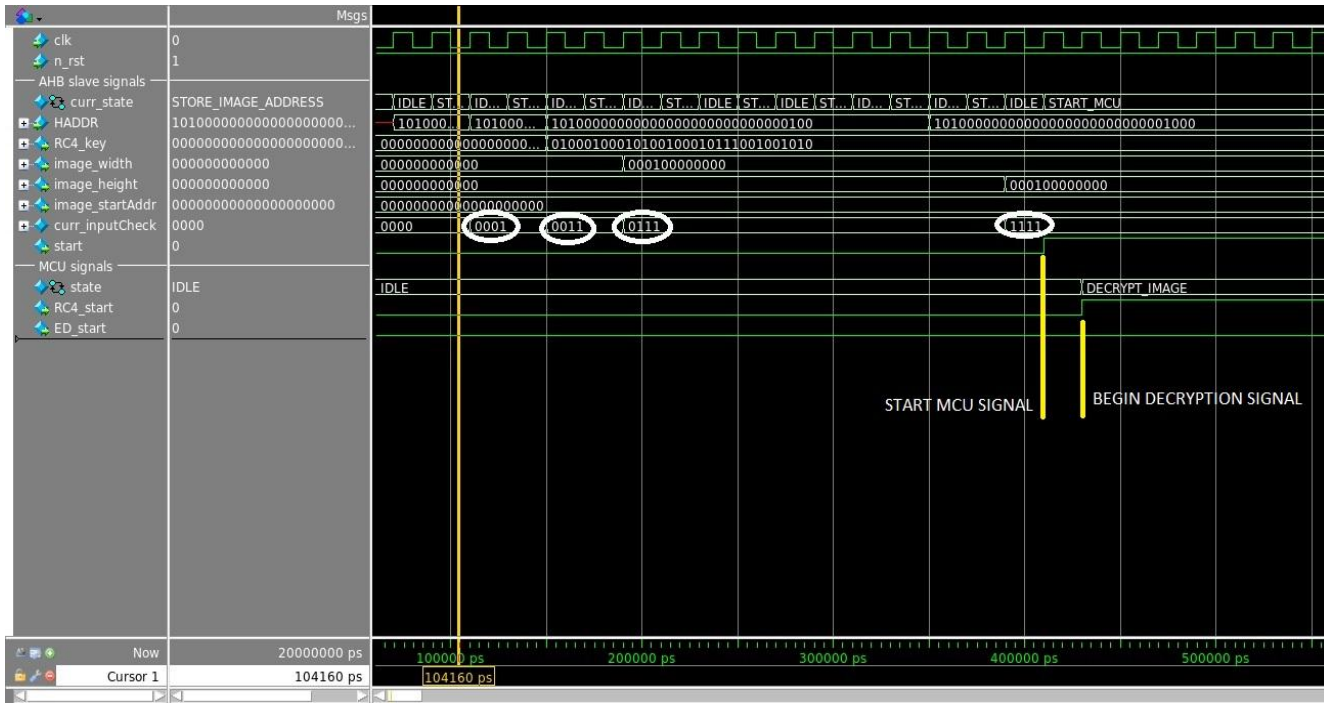


Figure 1: Receiving of inputs by AHB slave.

The AHB slave contains memory mapped registers which store and supply the inputted information to the surrounding modules. Without these inputs, all calculation will be incorrect. The HADDR line, while writing to the slave, chooses which register to write into with the use of its four least significant bits. The line curr_inputCheck contains a list of all the inputs that have been received. Each bit represents one of the four required inputs. Once all 4 inputs have been received, the AHB slave sends the start MCU signal to the MCU. One clock cycle later, the MCU sends the start decryption signal to the RC4. Up until then, both the RC4 start and the ED start signals are held low. This proves DSSC 1.

Testing of AHB master and AHB helper -

DSSC 5: Demonstrate by simulation of a Verilog test bench that data transmitted serially on the AHB-LITE bus is reading, writing and waiting correctly. The AHB signals are asserted at the right time to propagate correct data to the individual modules and relay their data to the SRAM.

The AHB master is responsible for the reading and writing of data from/to the SRAM. The AHB helper assists in this function by deciding between the Edge detection module's lines, and the RC4 module's lines. It does this with the help of the RC4 start and ED start lines coming out of the MCU. To test the functioning of these two modules together, the input lines of the AHB helper need to compare to the output lines of the AHB master, while considering the current state of ED start and RC4 start. The correct functioning of our module can be seen from the following waveforms -

Figure 2: RC4 read instruction to AHB master

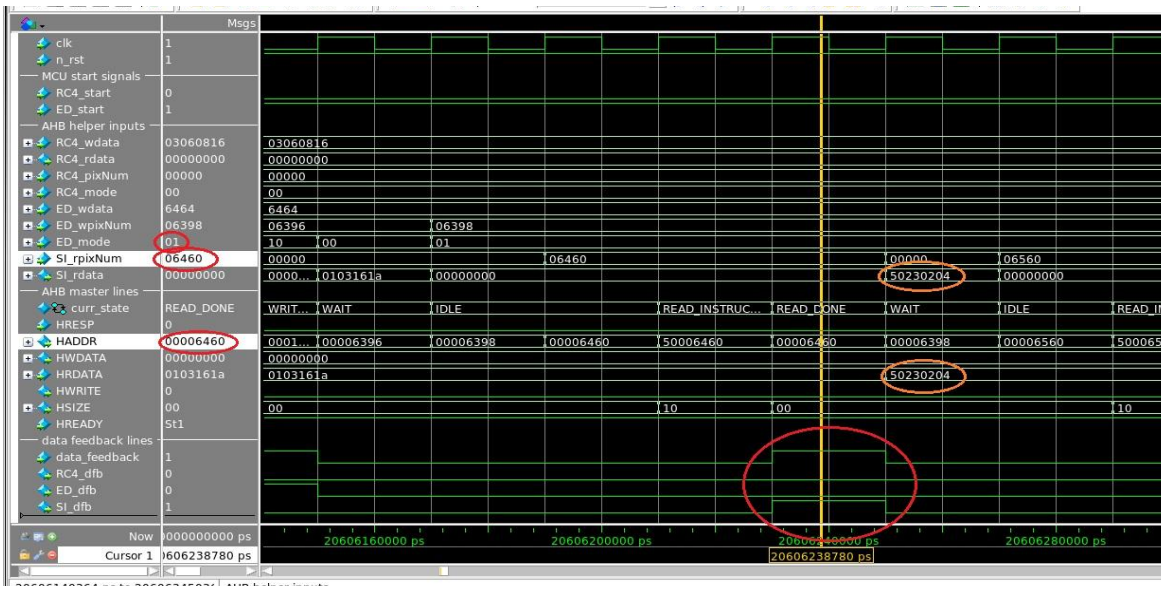


Figure 3: Edge Detection module read instruction to AHB master

It is evident from Figure 2 and Figure 3 that all read instructions carried out through the AHB bus are functioning as required. First the RC4/ED module sends the AHB bus a read request. It does this by setting mode to '01'. Along with this, the required pixel number is also set. This is basically the address in the SRAM from where the pixel data needs to be read. The AHB helper then relays the appropriate signals to the AHB master, and the AHB master reads the data from the SRAM. As soon as the data is available on the line, the AHB master asserts data feedback, followed by which the AHB helper relays it to the correct line (RC4/ED/Sample Image). **This can be seen in the red circle at the bottom of the cursor.** One clock cycle later, the data on the HRDATA line is seen on the correct RC4/ED read line. At this point, the AHB master is in its idle state, and is ready for a new read/write request.

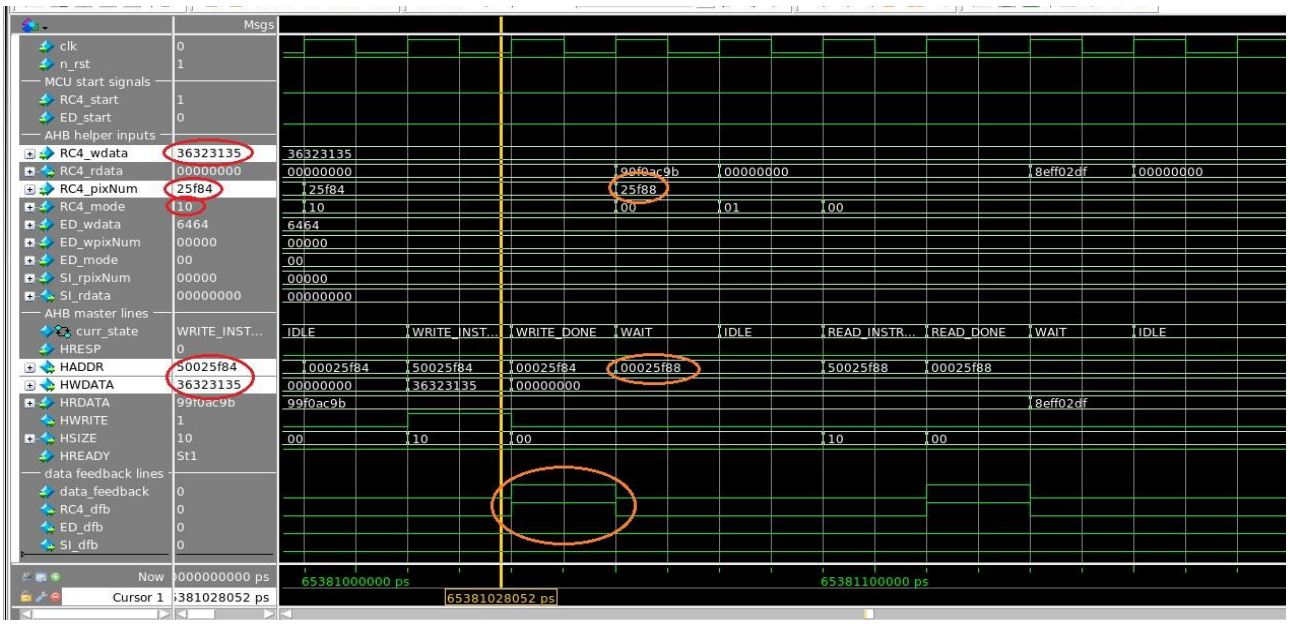


Figure 4: RC4 write instruction to AHB master

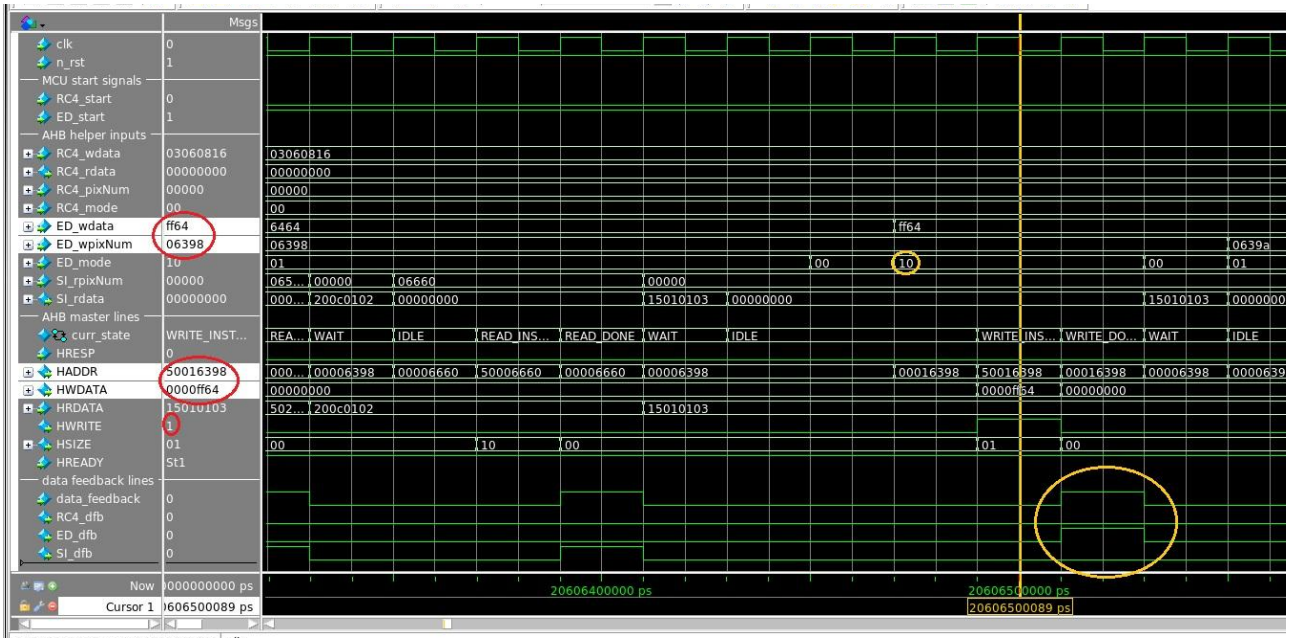


Figure 5: Edge detection Write Instruction to AHB master

It is evident from Figure 4 and Figure 5 that all write instructions carried out through the AHB bus are functioning as required. First the RC4/ED module sends the AHB bus a write request. It does this by setting mode to '10'. Along with

this, the required pixel number that is being set, and the data to be written is sent. The AHB helper then relays the appropriate signals to the AHB master, and the AHB master writes the data to the SRAM. This can be seen by comparing the wdata lines from the individual modules, and the HWDATA line going out of the AHB master. As soon as the data has been written onto the SRAM, the AHB master asserts data feedback, followed by which the AHB helper relays it to the correct line (RC4/ED/Sample Image). **This can be seen in the orange circle at the bottom of the screen,** towards the right of the cursor. The reason the data feedback is one clock cycle later is because the AHB master needs to first move into 'write instruction' state. After this it goes into 'write done' where data feedback gets asserted. The AHB master stays in this state for one clock cycle then goes to its idle state, and is ready for a new read/write request.

RC4 DECRYPTION TESTS

DSSC 3: Demonstrate by simulation of a Verilog test bench that decryption of RC4 through our design matches the original picture which was encrypted through python code. The waveforms below show the decryption of the second four bytes of data. This is enough to demonstrate that the whole image is decrypted correctly, because if there is any error generated it will start from the first four bytes and will be propagated to the rest of the image.

To test the RC4 code:

1) decrypted data through python code or original data should be supplied

```
for byte in encryptedList:
    i = (i + 1) % 256
    j = (j + S[i]) % 256
    temp = S[i]
    S[i] = S[j]
    S[j] = temp
    decr.append(byte ^ int(str(S[(S[i] + S[j]) % 256])))

print("DECRYPTED FIRST FOUR BYTES")
print(decr[:4])
print(decr[4:8])
print(decr[8:12])
print(decr[12:16])
misc.imshow(np.array(decr).reshape(org_shape))

# print("Encrypted")
# print(encryptedList[0])
# print(encryptedList[1])
# with open("marked.bmp", "wb") as fileOut:
#     fileOut.write(bytes(encryptedList))
```

```
ncrypt
(256, 256)
ENCRYPTED FIRST FOUR BYTES
[138, 131, 247, 72]
[68, 250, 37, 211]
[156, 253, 106, 25]
[75, 110, 226, 12]
DECRYPTED FIRST FOUR BYTES
[202, 209, 208, 211]
[208, 219, 217, 230]
[225, 226, 219, 227]
[224, 217, 231, 223]
```

2) screenshot of the waveform of the testbench

Fig.2 - Python code which outputs the first 16 encrypted bytes and the first 16 decrypted bytes.

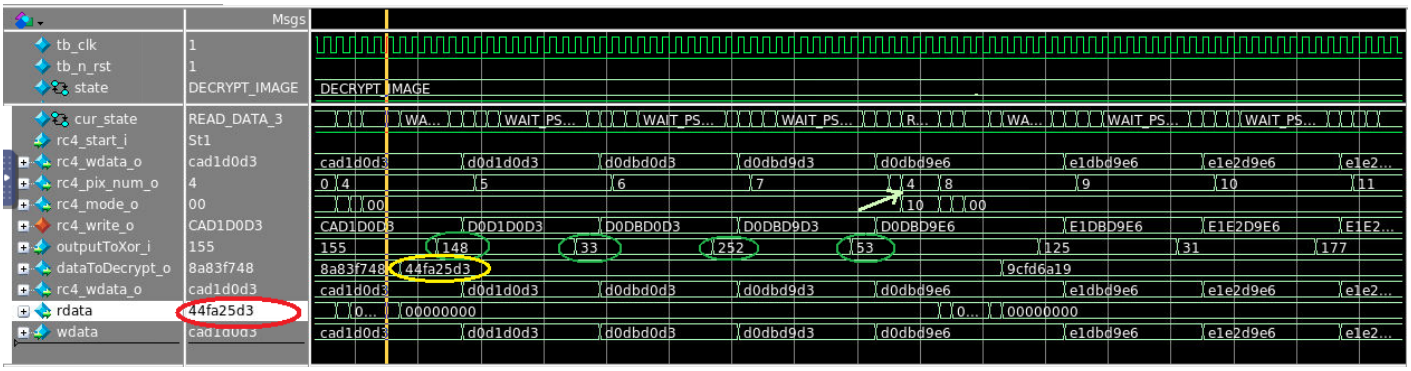


Fig.3 - Displays the decryption of the first 4 bytes.

First, RC4 module supplies AHB with instruction to read(rc4_mode). The data that is read from SRAM is only available for 1 clock cycle. When the module receives feedback from AHB master, it saves the data in internal registers. You can see on figure 3 that the data is available for one clock cycle and then it becomes zero (this is shown in the row of the red circle). The yellow circle shows that the data is saved internally(dataToDecrypt). After that, the module generated 4 bytes as shown with the green circle. When a byte is generated, it is xor with the corresponding byte of the input and that byte is then written in wdata registers. outputToXOr is shown in decimal. For example, the first data is 148, converted in hex it is 94. The first byte as shown in dataToDecrypt is 44. When those two values are xored the output is d0. After xor operation the value is stored in wdata and it could be seen right after the yellow circle that the value (first byte) changes correctly.

Now we need to compare values generated by python code and generated by verilog test bench. In picture 2 you can see the second four bytes of the encrypted image are 68, 250, 37, 211(since we decided to compare the second 4 bytes for verification). 68, 250, 37, 211 -> 44fa25d3. This exactly matches dataToDecrypt. Therefore, RC4 module communicates correctly with AHB. As shown with the code from python you can see that the original bytes (aka decrypted) are 208, 219, 217, 230 -> d0dbd9e6. This exactly matches the data in wdata register. Therefore, the rc4 module correctly decrypt. The white arrow shows the state where the module writes the data. Pix_num shows the current pixel of processing but since when we have read the second four pixels, at the time we need to write we are processing the 8th pixel. Therefore, we need to subtract 4 To supply the right location for writing, which is 4 as shown with the white arrow.

EDGE-DETECTION (EDGE DETECTION + SAMPLE IMAGE DATA STORAGE) TESTS

DSSC 4: Demonstrate by simulation of a Verilog test bench that Sobel Edge detection gives a visible and convincing image. This design specific criteria emphasise on the correct working of Edge-Detection and sees whether the image obtained is a convincing edge-detected image.

This test forms the core of our design since our final output is the edge-detected image. The input to the edge-detection module is the decrypted image obtained after RC4 decryption. This test largely depends on the correct working of Edge-detection module and the sample image data storage module. AHB-Helper and AHB-Master supplement the working of these two modules.

To test that this is functioning correctly two things need to be checked -

1) The correct filling of the buffer in the sample image data storage module
2) The correct calculation of gradient in the Edge-Detection module
3) See whether after performing edge-detection on the entire image, whether it looks good enough.

The results for this test case can clearly be seen in the following waveform diagram.

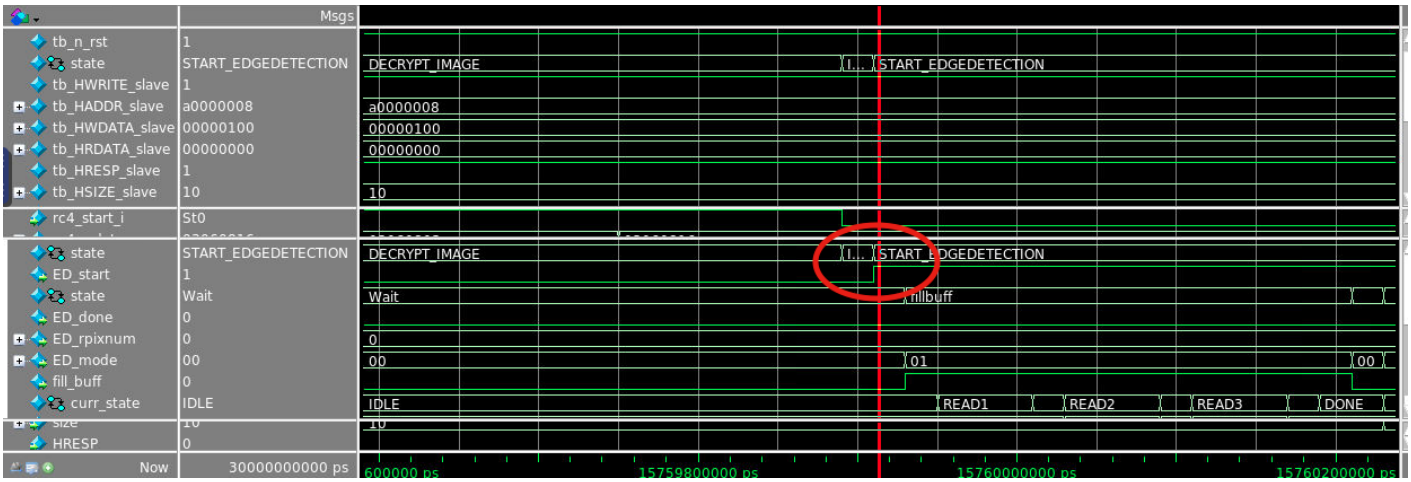


Fig.3 - Starting of Edge-Detection by MCU

Once the RC4 decryption is done, RC4 module sends a RC4_done signal to MCU which then decodes to move forward to the next part of our design i.e. Edge-detection. In above image, you can clearly see the change in mcu state from '*DECRYPT_IMAGE*' to '*START_EDGEDETECTION*'. At the same time, the ED_start signal is asserted sending a signal to edge-detection module to begin the edge-detection process. now the edge-detection resets the pixel number within the module to the start address of the image. And to perform the edge-detection, it sends a signal to sample image data storage in the form of '*fill_buff*'. From here on, we look at the filling of the buffer in the sample image data storage module.

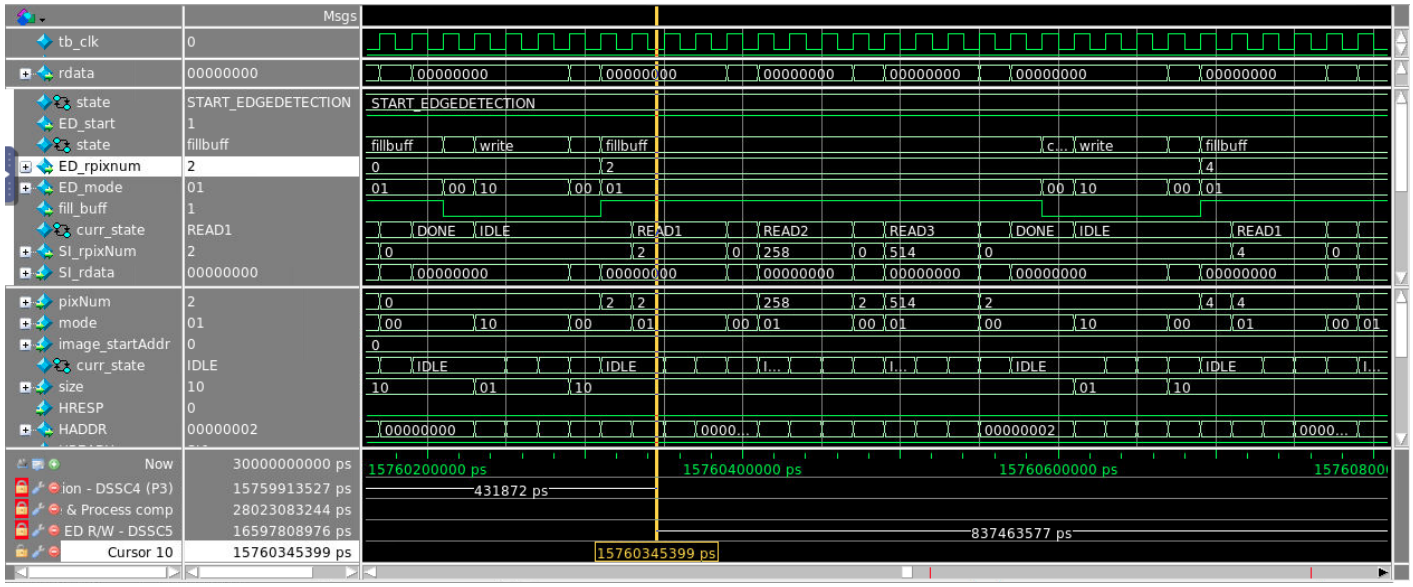


Fig.4 - Filling of buffer in sample image data storage module

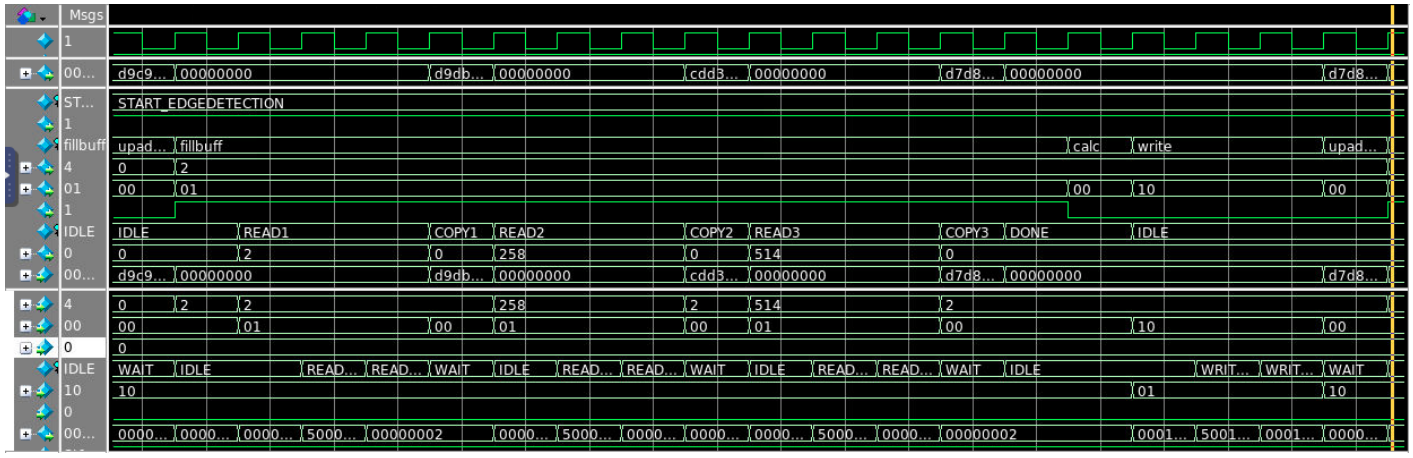


Fig.5 - Magnified image of various states required to fill the buffer once

From fig. 4 and fig. 5, we can see how the buffer is filled once in the sample image data storage module. From fig. 4, it is easy to note that a buffer fill is followed by calc and then writing of the calculated pixel into a new image space in SRAM. From fig. 5, you can see that edge-detection for two pixel of information requires reading a buffer thrice since at each time, a total of 4 bytes are read. reading thrice gives us the sliding window buffer and completely fills the sample image data storage. At this point, a buff_filled signal is sent to Edge-detection module sending the entire window buffer for calculation of gradient.

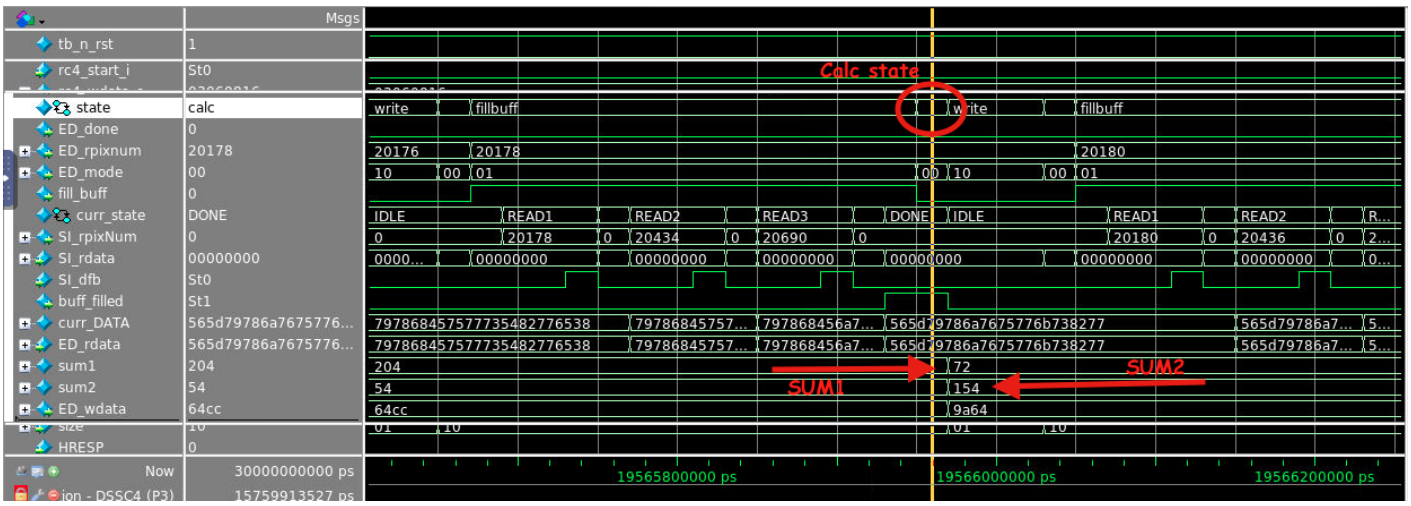


Fig.6 - Calculation of gradient in the Edge-Detection module

From the above image, looking carefully at the ED_rdata, we can form the window buffer ourselves although it requires some work since it is depicted in hexadecimal. The buffer at the point where cursor is made below.

Window Buffer	Y1	Y2	Y3	Y4
X1	107	115	130	119
X2	106	118	117	119
X3	86	93	121	120

Now, calculating the gradients for X and Y,
gradient X1 = (130 - 107) + 2 (117 - 106) + (121 - 86) = 80
gradient Y1 = (86 - 107) + 2(93 - 115) + (121 - 130) = -74
gradient X2 = (119-115) + 2(119-118) + (120 - 93) = 33
gradient Y2 = (93 - 115) + 2(121 - 130) + (120 - 119) = -39

sum2 = abs (grad X1) + abs(grad Y1) = 154
sum1 = abs (grad X2) + abs(grad Y2) = 72

The order for sum is reversed because it makes it easy for us to write, as sum2 basically means the more significant byte for the two pixels we need to write while the sum1 marks the least significant byte out of the two pixels we edge-detected. The values for sum are marked in the image above. Once the values are calculated, they are written in SRAM using AHB-master which is explained in DSSC 5 before.

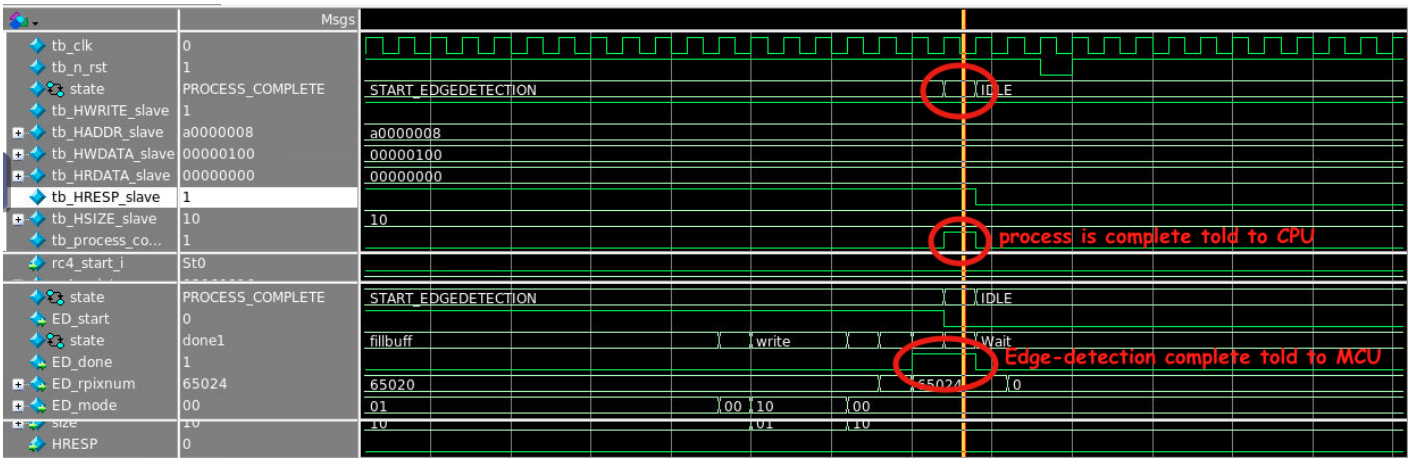
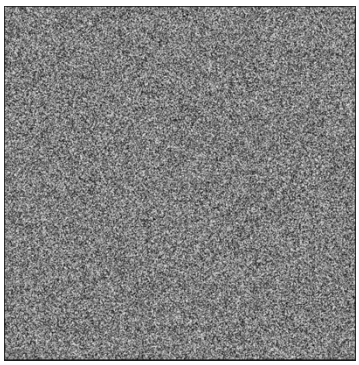


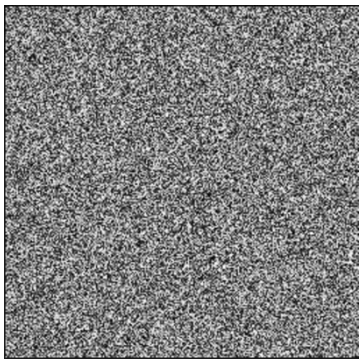
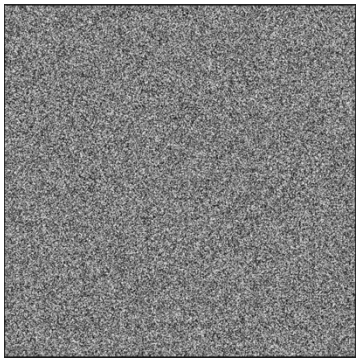
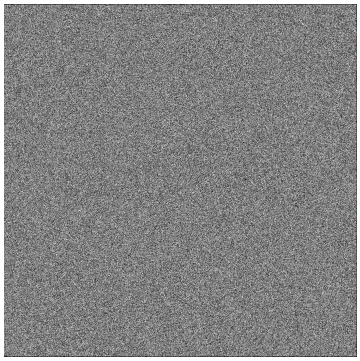
Fig.6 - Calculation of gradient in the Edge-Detection module

After completing the edge-detection process, the Edge-detection module sends a ED_done signal to MCU which then notifies the CPU via AHB-slave that the entire process is done and our design is waiting for further instructions.
Now we need to check whether the image we obtained is convincing enough.

Comparison of Output Images

DSSC 2: Demonstrate by simulation of a Verilog test bench that our module works correctly for variable image sizes.
Following are the results of our design compared with the python implementation:

Case	Encrypted Image	Decrypted Image	Python Implementation
girl (512 x 512)			

fruits (256 x 256)			
city (512 x 512)			
man (1024 x 1024)			

Case	Decrypted Image	Edge-detected Image	Python Implementation
girl (512 x 512)			
fruits (256 x 256)			
city (512 x 512)			

man (1024 x 1024)			

The above images are different sizes and our design does a pretty good job in decrypting and edge-detecting images with variable sizes. This successfully shows the completion of **DSSC 2** in our list.

6. Project Management

6.1 Per Member Responsibilities

Individual Responsibilities -

Ruchir Aggarwal -

1. Implemented AHB-helper and MCU
2. Wrote system level test bench
3. Wrote module level test bench for AHB-slave, RC4, AHB-helper.
4. Prepared RTL and block diagrams for various modules
5. Made the layout of the chip

Gautam Rangarajan -

1. Implemented AHB-master, AHB-slave, Edge-detection module.
2. Wrote system level test bench
3. Wrote module level test bench for AHB-master, MCU, Edge-Detection module.
4. Prepared RTL and block diagrams for various modules

Ribhav Agarwal -

1. Implemented Edge-detection module, sample image data storage
2. Prepared RTL and block diagrams for various modules
3. Wrote module level test bench for Edge-Detection module, sample image data storage.
4. Did most of the documentation
5. Made the layout of the chip

Dimcho Karakashev -

1. Implemented RC4
2. Wrote test benches for sub-modules in RC4
3. Prepared RTL and block diagrams for various modules

7.Appendix

Directory where all files are located: /home/ecegrid/a/mg84/ASIC_Design/Project

Type: File Location	Description
Verilog code: Project/source/AHB_helper.sv	Purely combinational logic chooses between RC4 or Edge detection instructions
Verilog code: Project/source/AHB_master.sv	AHB master communication with SRAM
Verilog code: Project/source/AHB_slave.sv	AHB slave communication with CPU
Verilog code: Project/source/counterI.sv	Will contain a value which is equivalent to the I value in the pseudo code for RC4.
Verilog code: Project/source/counterPixel.sv	CounterPixel is basically flex counter which keeps track of how many bytes have been decrypted.
Verilog code: Project/source/dataToDecrypt.sv	Will save internally the 4 bytes of data which are read through AHB
Verilog code: Project/source/edm.sv	Main edge detection module
Verilog code: Project/source/locSafe.sv	Will be used to save an intermediate value for RC4.
Verilog code: Project/source/mcu.sv	Main control unit
Verilog code: Project/source/outputToXor.sv	Once a pseudo-random value is generated by the generation of pseudo-rand value module it is saved in this file
Verilog code: Project/source/overall.sv	Overall wrapper file
Verilog code: Project/source/RC4_Core_comb.sv	All the combination logic needed for the RC4 Core module from the State machine
Verilog code: Project/source/RC4_CORE_decrypted_data.sv	Once the inputted data is xored and decrypted, it is saved in this module.
Verilog code: Project/source/RC4_CORE_STATE_MACH.sv	Contains the state machine of RC4 CORE
Verilog code: Project/source/RC4_CORE.sv	Wraps all needed sub-modules for RC4 core.

Verilog code: Project/source/RC4_gen_pseudo_rand_module.sv	Wraps all needed sub-modules for RC4 gen pseudo rand module
Verilog code: Project/source/RC4_gen_ps_state.sv	RC4 state machine for generation_of_pseudo_random module
Verilog code: Project/source/RC4_pseudo_rand_comb.sv	All combinational logic for RC4_gen_pseudo_rand_module
Verilog code: Project/source/RC4.sv	Wraps sub-modules RC_CORE and RC4_gen_pseudo_rand_module
Verilog code: Project/source/sample_image_data_storage.sv	Buffer storage support module for edm
Verilog code: Project/source/sarr.sv	Storing the state array for the RC4
Test Bench: Project/source/tb_AHB_helper.sv	Test bench for AHB helper
Test Bench: Project/source/tb_AHB_master.sv	Test bench for AHB master
Test Bench: Project/source/tb_AHB_slave.sv	Test bench for AHB slave
Test Bench: Project/source/tb_counterI.sv	Test bench for the flex counter I
Test Bench: Project/source/tb_dataToDecrypt.sv	Test bench for the data to decryption
Test Bench: Project/source/tb_edm.sv	Test bench for EDM
Test Bench: Project/source/tb_locSafe.sv	Test bench for locSafe module
Test Bench: Project/source/tb_mcu.sv	Test bench for MCU
Test Bench: Project/source/tb_outputToXor.sv	Test bench for outputToXor
Test Bench: Project/source/tb_overall.sv	Test bench for overall design
Test Bench: Project/source/tb_RC4_Core_comb.sv	Test bench for all combinational logic for RC4s
Test Bench: Project/source/tb_RC4_CORE_decrypted_data.sv	Test bench for RC4_CORE_decrypted_data
Test Bench: Project/source/tb_RC4_CORE_STATE_MACH.sv	Test bench for RC4_CORE_STATE_MACH
Verilog code: Project/source/tb_RC4_CORE.sv	Test bench for RC4_CORE (all submodules)

Test Bench: Project/source/tb_RC4_gen_pseudo_rand_module.sv	Test bench for all RC4_gen_pseudo_rand submodule
Test Bench: Project/source/tb_RC4_gen_ps_state.sv	Test bench for all RC4_gen_pseudo_rand_module state machine
Test Bench: Project/source/tb_RC4_pseudo_rand_comb.sv	Test bench for all RC4_gen_pseudo_rand_module combinational logic
Test Bench: Project/source/tb_RC4.sv	Test bench for the overall RC4
Test Bench: Project/source/tb_sample_image_data_storage.sv	Test bench for the sample image data storage
Test Bench: Project/source/tb_sarr.sv	Test bench for the state array for the RC4
Data Sheet:	AHB-Lite Manual Specification
Python Code:	Python implementation of the same algorithm for Edge-Detection
Python Code:	Python implementation of the same algorithm for RC4
Diagrams	Contains all necessary information pertaining to different diagrams used in the project design