AESelerator

Advanced Encryption Standard Cryptographic Accelerator

Project Documentation

Yipeng Huang (Integration Master)

Scott Rogowski (Verification Master)

Hsin-Tien Shen (Design Master)

I. Team Member Responsibilities

Yipeng Huang—*Integration Master*
Responsible for overall design from top-down perspective, implementation in Verilog of Inputter and Outputter modules, TCP/IP header processing logic, and documentation.

Scott Rogowski—*Verification Master*
Responsible for the verification suite including all files within the verif folder and all the requisite coding in the C programming language. Additional responsibilities include diagramming.

Hsin-Tien Shen—*Design Master*
Responsible for implementation of subunits from bottom-up perspective, design and implementation of RoundKeyGenerator and Encrypter modules.

II. Design Overview

Design purpose

This project will implement a hardware-based encryption unit that implements the Advanced Encryption Standard (AES). The AESelerator encryption unit will be able to communicate via a network interface to send and receive data in TCP/IP format.

Encryption methods allow sensitive data to be sent over a network connection securely. The sending party and the receiving share an encryption key that is communicated separately from the data itself. If encrypted data is intercepted during transmission, the intercepting party cannot decrypt and gain access to the data without access to the key.

The Advanced Encryption Standard (AES) is a symmetric-key encryption standard adopted by the US government for all governmental encryption requirement. The algorithm was selected for its simplicity, robustness, and speed. A detailed description of the algorithm can be found in the Federal Information Processing Standards Publication 197.

Description of the AES-128 standard

There exist three variants of AES, characterized by the number of bits in the cipher key used for encryption. The three standards are AES-128, AES-192, and AES-256, which respectively use 128, 192, and 256 bits in the cipher key. The length of the cipher key directly influences the running time of the algorithm and the size of various algorithm variables involved in creating the ciphertext. For a hardware implementation of AES, it would be infeasible to incorporate the implementations of all three standards in one design.

Of the three standards, AES-128 requires the smallest amount of storage elements and requires the fewest number of iterations of the cipher algorithm. Since our design is targeted to encrypt data at the same bandwidth as the network interface, the rate at which our design encrypts data is important. Therefore, we have selected AES-128 as the standard for this implementation.

Encryption Algorithm Overview

The algorithm for AES is specified in the Federal Information Processing Standards Publication 197. However, for the purposes of understanding this project documentation, it is useful to understand the broad stages in the algorithm and the functions that are called in each stage. The following overview will describe the structure of the algorithm without describing in detail the mathematical operations of the algorithm.

The AES algorithm involves two main loops: the key expansion loop and the cipher loop.

The key expansion loop uses the previous round key to generate a new round key, which is used by the cipher loop. The first round key is simply the given encryption key. From here, the key expansion loop relies on two functions, SubWord() and RotWord(), and a constant lookup table Rcon[ ] to generate its new key.

The cipher loop takes the current encrypted state of the 128 bit data as input, along with the round key from the key expansion loop, and executes a round function outputting a new encrypted state of the data. In the first round, the input is simply the plaintext data. Then, for AES-128, the round function is carried out ten times, and the output of the tenth round function is the ciphertext. The cipher loop relies the four functions SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey(). The SubBytes() function relies on a constant lookup table S-box.

Definition of Terms

Plaintext
Data sent from the remote host to the AESelerator to be encrypted. In AES, plaintext is encrypted 128 bits at a time.

Ciphertext
Data that has been encrypted by AESelerator and is ready to be sent back to the remote host. In AES, ciphertext is generated 128 bits at a time.

Flit
The unit of network data that is sent on the network in one clock cycle of the AESelerator. Each flit contains four bytes of data. Flits are sent by either the remote host (user) or the AESelerator.

Packet
The complete group of flits that share a same TCP/IP header. Each packet is composed of 13 flits, for a total of 52 bytes. A packet consists of five TCP/IP header flits (for 20 bytes), and two sets of four flit (128 bit) plaintext or ciphertext.

Datapath Timing Plan

Each packet processed by AESelerator goes through three stages: *buffer* from the network at the AESelerator, *encrypt*, and *write out* to the network.

The timing for each stage is outlined in the figure below:

<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th><th>32</th><th>33</th><th>34</th></tr>
<tr><td>Encryption 1</td><td colspan="5">Buffer TCP/IP header flits</td><td colspan="8">Decode TCP/IP header, buffer data flits</td><td colspan="13">Round key generation, Encrypt</td><td colspan="8">Write out</td></tr>
<tr><td>Encryption 2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="5">Buffer TCP/IP header flits</td><td colspan="8">Decode TCP/IP header, buffer data flits</td><td colspan="8">Round key generation, Encrypt</td></tr>
<tr><td>Encryption 3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="5">Buffer TCP/IP header flits</td><td colspan="3">Decode …</td></tr>
</table>

For each encryption operation, the AES accelerator will receive 52 bytes of input arriving as 13 flits, each four bytes in size. Flits may arrive at the accelerator as fast as one flit per clock cycle; therefore, the upper bound of the bandwidth of the accelerator is one complete encryption per 13 clock cycles. When the 52-byte input to the accelerator does not arrive continuously, the input buffer will simply stall until the full input arrives.

Once all 20 bytes of the header arrive, the bytes may be removed from the input buffer for processing. The buffer will then continue receiving the 32 data bytes.

Once all 32 bytes of the data arrive, the cipher logic will have 13 clock cycles to completely generate the ciphertext. Since the design is targeted to run at the same bandwidth as the network, the cipher operation must complete within 13 cycles and begin writing out the output to the network. If the cipher logic takes any more time than 13 cycles, incoming flits would start accumulating in the input buffer and eventually overflow. Therefore the design specification requires that all ciphertext logic complete in 13 clock cycles.

Error handling

AESelerator will have to anticipate incorrect input flits. Packets may be erroneous due to a number of reasons, including incorrect destination IP, incorrect checksum, or incorrect values for any of the other header bytes. Furthermore, the packets may contain too many or too few flits, resulting in all following flits going out of alignment.

When Inputter sends a packet off to the Encrypter for encryption, the error detection unit checks all possibilities that the packet contains errors. If errors are found, the RoundKeyGenerator and Encrypter do not interrupt their operations. Instead, the Inputter signals to the Outputter that the next incoming ciphertext (the packet that contained errors) should be converted to an error packet when it is serialized to the network output interface. The timing of events is illustrated below.

<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th><th>32</th><th>33</th><th>34</th></tr>
<tr><td>Encryption 1</td><td colspan="5">Buffer TCP/IP header flits</td><td colspan="8">Decode TCP/IP header, buffer data flits</td><td colspan="13">Packet error checking (error found), Round key generation, Encrypt</td><td colspan="8">Write out error packet</td></tr>
</table>

An error packet has the checksums set to zero, and the data flits are also zero.

III. Unit-Level Interfaces

Top-level interfaces

The following top-level interface specification will describe the functionality of each of the interfaces of the top-level module, AESelerator.v.

Interface type	**Interface name**	**Data width**	**Interface functionality**
Inputs	reset	1 bit	The reset signal to the system. Asserted once when AESelerator is started.
	clock	1 bit	The clock signal.
	config_i	8 bits	Configuration input from the CPU. Once AESelerator is reset, configuration input including AESelerator's IP address and the encryption key would arrive eight bits at a time. Configuration takes 20 clock cycles.
	flit_i	4x8 bits	The network input interface for flits. Each flit is 32 bits wide. The design prefers bits to be organized into byte-sized arrays because the encryption logic requires plaintext to be bundled into bytes.
	flit_valid_i	1 bit	A control bit that indicates that the network interface is passing flits to AESelerator. This control bit is asserted if and only if the flit_i input should be stored into the input buffer.
Outputs	ready_o	1 bit	A control bit that indicates that AESelerator is outputting valid data flits. This control bit is asserted true if and only if the network interface should pass the bits at flit_o to the network.
	flit_o	4x8 bits	The network output interface for flits.

TCP/IP packet format for network interface

The plaintext and ciphertext packets have to be prefaced with TCP/IP headers consisting of five flits, or 20 bytes.

The following table describes the permitted values and purpose of each byte.

<table>
<tr><th>Byte number</th><th>Permitted values</th><th>Purpose</th><th>Description</th></tr>
<tr><td>0</td><td>0100 0101</td><td>IP version and header length</td><td>The first four bits the IP version and has to be 4. The second four bits is the header length and has to be 5 to indicate 20 bytes.</td></tr>
<tr><td>1</td><td>0000 0000, 0010 0000, or 0100 0000</td><td>Type of IP service</td><td>Bits 0-2 are precedence. This type of packet should never be assigned priority higher than 2. Bit 3 is delay and should be 0. Bit 4 is throughput and should be 0. Bit 5 is reliability and should be 0. Bits 6-7 are unused.
<table>
<tr><td>Bit number</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>Conten s</td><td colspan="3">Precedence</td><td>Delay</td><td>Throughput</td><td>Reliability</td><td colspan="2">Unused</td></tr>
</table>
Figure 2. Contents of Byte 1</td></tr>
<tr><td>2-3</td><td>0000 0000 0011 0100</td><td>Total length</td><td>The total packet length of each transmission is 52 bytes.</td></tr>
<tr><td>4-5</td><td>0000 0000 0000 0000</td><td>Unused</td><td>This field should have the value 0</td></tr>
<tr><td>6-7</td><td>0000 0000 0000 0000</td><td>Fragmentation flags</td><td>This field should have the value 0</td></tr>
<tr><td>8</td><td>0000 0000 to 1111 1111</td><td>Time to live</td><td>A value that indicates how many more network interfaces the packet can visit before it is discarded. If this is positive the AESelerator will decrement it by 1. If it is 0, the packet should be dropped from the network and the AESelerator will return that this is an error in the input.</td></tr>
<tr><td>9</td><td>0000 0110</td><td>Link layer protocol</td><td>This value should be 6 to indicate Transmission Control Protocol.</td></tr>
<tr><td>10-11</td><td></td><td>Checksum</td><td>The first 13 bits should be the parities of the 13 flits. The last 3 bits should be 0.</td></tr>
<tr><td>12-15</td><td></td><td>Source IP</td><td>The IP address of the network interface that sent the packet. The AESelerator should return this as the destination IP.</td></tr>
<tr><td>16-19</td><td></td><td>Destination IP</td><td>The IP address of the destination of this packet. The AESelerator should check if this address matches the AESelerator’s IP address.</td></tr>
<tr><td>20-51</td><td></td><td>Data value</td><td>The data value for the AESelerator to encrypt.</td></tr>
</table>

III. Subunits and Test Harness

Block diagram of subunits

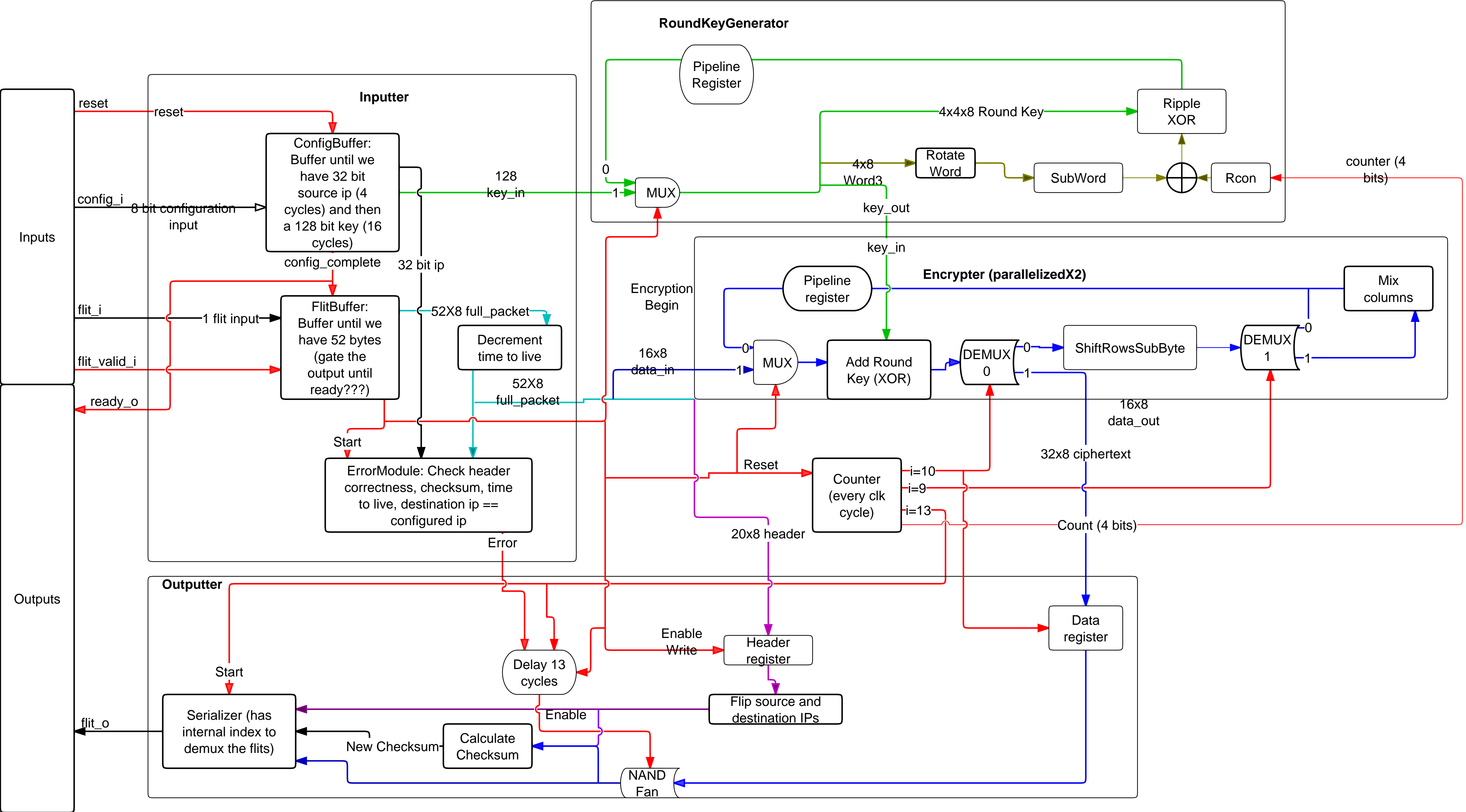
Our design will include one top-level module named AESelerator.v. The AESelerator.v module will provide all input and output of the design, and will mediate communication between the four major subunits: Inputter.v, Outputter.v, RoundKeyGenerator.v, and the Encrypter.v.

The block diagram on the following page lays out the relationship of the major subunits, along with the flow of data and control signals between the subunits.

The data lines between units are color-coded according the following scheme:

Red: control lines (1 bit)
Cyan: full data packet, including header (52 bytes)
Blue: data encrypting (32 or 16 bytes)
Purple: header (20 bytes)
Green: key (16 bytes)
Black: other

AESelerator



Subunit description and interfaces

The following unit level interface specification will describe each major unit's functionality and its interfaces.

RoundKeyGenerator.v

The RoundKeyGenerator takes in the 16 byte key from Inputter and stores the key to pipeline registers. The key then goes through the methods RotateWord(), Subbytes(), and is then bitwise-XOR'ed with the original key. The new key is then provided to the Encrypter for the next round of encryption.

Encrypter.v

The Encrypter takes in 16 bytes of data and carries out rounds of encryption. The first round begins with the AddRoundKey method, which bitwise adds the data with the key for that round. Then it goes to 10 rounds of SubBytes, ShiftRows, MixColumns and AddRoundKey. Then finally the data goes to the final round, which includes only SubBytes, ShiftRows and AddRoundKey. The completed ciphertext is provided to the Outputter stage for output to the network.

Inputter.v and Outputter.v

The two tables on the following pages describe the functionality of the Inputter and Outputter modules, along with a detailed breakdown of the functionality of each of the units' interfaces.

Inputter subunit description and interfaces

Module name	Module functionality	Component type	Interface name	Data width	Interface functionality
Inputter.v	Provide all parsing and buffering functionality for processing the inputs to the design. Inputs include the network interface and the CPU configuration interface.	Inputs	config_i	8	Configuration input from the CPU.
			flit_i	4x8	The network input interface for flits.
			flit_valid_i	1	A control bit that indicates that the network interface is passing flits to AESelerator.
		Outputs	ready_o	1	A control bit that indicates that AESelerator is outputting valid data flits.
			key	4x4x8	The 128-bit first key that will be used to generate the key schedule.
			full_packet	52x8	The entire packet that is fully loaded into buffer.
			start_encryption	1	A control bit that is asserted when the entire packet has been collected. Signals RoundKeyGenerator and Encrypter to start rounds of processing.
			error	1	A control bit that indicates the packet AESelerator just received failed error checking. Signals the error header generator to overwrite headers once the errornous packet is in write out stage.

Outputter subunit description and interfaces

Module name	Module functionality	Component type	Interface name	Data width	Interface functionality
Outputter.v	Handle all error checking and serialization of the outputs of the AESelerator.	Inputs	error	1	A control bit that indicates the packet AESelerator just received failed error checking. Signals the error header generator to overwrite headers once the errornous packet is in write out stage.
			header	20x8	The TCP/IP header from the Inputter unit.
			i10	1	A control signal that is asserted whenever the encrypter completes its tenth round of encryption, indicating the Ouputter unit should begin accepting ciphertext from Encrypter.
			i13	1	A control signal that is asserted whenever the Encrpyter completes its 13th round of encryption, indicating the Outputter unit should begin serializing the contents in its output buffer.
			ciphertext	2x16x8	The ciphertext from the two Encrypter pipelines.
		Output	flit_o	4x8	The network output interface for flits.

Test harness structure

The test harness is coded in the 'C' programming language and checks for correct device behavior and overall encryption accuracy. Every cycle, the test bench will decide whether or not to calculate and begin sending a new encryption, a new configuration, or a reset. The actual values of these instructions are dictated using the current state of the machine and user defined parameters. Upon generation, the instructions are placed into future input queues to be sent into the device sequentially each clock cycle. Similarly, upon calling a new instruction, the test bench will immediately calculate the future expected output values and place these values into output queues. Every cycle, the actual output values are compared to the expected output values for that cycle.

The behavior of the test harness is controlled using density and mask parameters. Density parameters are used to dictate how often to send a new packet, interrupt a packet being sent, or reset. In addition, how often any individual header component is purposefully wrong is specified through these density parameters. Mask parameters allow the encryption key, source ip, destination ip, time to live, and data inputs to be masked in order to allow for a small or a wide range of values.

The reset instruction will generally be followed by a configuration instruction. If no purposeful errors are dictated, this instruction will send a new IP address and encryption key over the course of 20 cycles. The test harness will then expect a ready signal from the device. The new packet instruction will generally be called upon receiving the ready signal. This instruction will send out a new header and new data in addition to asserting the 'data_valid_i' signal over the course of 13 cycles. After a lag of 13 cycles during which the encryption is presumably occurring within the device, the test harness will expect the encrypted data to come back over 13 more cycles. On every cycle, the returned data will be compared against the expected data queue. If the header is sent malformed, the test harness will expect an error packet to be returned immediately upon receiving the entire packet.

Internally, the test harness purposefully uses a method different from the implementation's method to do its encryption. While the implementation uses a slightly faster though more complicated AES method done by combining some instructions together, the test harness will use the simpler original AES method.

IV. Microarchitecture Design

The following tables will describe the interfaces and implementation plan of all modules and submodules of AESelerator.

Detailed Microarchitecture

AESelerator

Module / Submodule Name	Inputs	Outputs	Implementation Notes
Inputter	reset, clk, 8 bit config_i, 4x8 flit_i, flit_valid_i	ready_o, error, 4x4x8 bit key, 52x8 full_packet, encrypt_begin	Upon receipt of reset, the inputter will start taking in the configuration bytes including a source ip address and a 128 bit key which will take 20 cycles. After this is complete, the inputter will assert the key given and stand ready to accept packets and will indicate this by asserting ready_o. Upon receipt of flit_valid_i, the inputter will begin to receive the encryption flits and will continue accepting for 13 cycle. Following this, the time to live will be decremented and the packets will be checked for errors. If an error is found, (or the the buffer overflows) an error will be asserted. Otherwise, the inputter will assert the signal to start encryption for one cycle.
RoundKeyGenerator	reset, clk, 4x4x8 bit key, 4 bit counter, encrypt_begin	4x4x8 bit round_key	The RoundKeyGenerator takes in 16 bytes key from Inputter and stores in the pipeline registers and outputs this key. The last word of the key then gets rotated, SBoxed and XORed with an Rcon word. Then, this new word gets ripple XORed with original key. Finally we output this new key and put it back into the pipeline register, repeating the cycle. Upon the start of a new encryption, reset is called which will select the key from the inputter rather than the key from the pipeline register
Encrypter	reset, clk, 4x4x8 plaintext, 4x4x8 round_key, i=10, i=9	4x4x8 ciphertext	There will be two encrypters working in parallel. Encryptor takes in 16 bytes data and goes to initial round: AddRoundKey with the new key from RoundKeyGenerator. Then it goes to 10 rounds of SubBytes, ShiftRows, MixColumns and AddRoundKey. Then finally the data goes to the final round: SubBytes, ShiftRows and AddRoundKey and gives the final data out to the next stage: Outputter. The control line i10 and i9 choose between two multiplexers for different rounds.
Outputter	reset, clk, 20x8 header, 32x8 ciphertext, error, encrypt_begin, i=10, i=13	flit_o	The outputter will take the header at the start of encryption and the ciphertext at the completion of encryption. Upon the completion of encryption, the checksum will immediately be calculated and replace that value in the header. The serializer will begin to serialize its parallel input, all 52 bytes of it, over the next 13 cycles. If the outputter gets an error flag for whatever reason, it will immediately generate and begin outputting the error packet.
Counter	reset, clk	4 bit count, i=9, i=10, i=13	A 4 bit counter that will reset to 0 at the start of each encryption and count upwards. When i=9, the i=9 bit will be asserted. When i=10, the i=10 bit will be asserted. When i=13, the i=13 bit will be asserted. Ideally, this should be reset every 13 cycles but this will not always be the case so it will just stop at 16

Inputter

Module / Submodule Name	Inputs	Outputs	Implementation Notes
Config Buffer	reset, clk, 8 bit config_i	config_complete, 4x4x8 key, 4x8 IP	Upon assertion of reset, buffer until we have 32 bit source ip and 128 bit key. Upon receiving everything, stop accepting input and assert config_complete. Output the key and the ip directly from the flipflops continuously.
Flit buffer	reset, clk, config_complete, 4x8 flit_input, flit_valid_i	52x8 full_packet, encrypt_begin	Upon assertion of config_complete and flit_valid_i, buffer until we have the 52 bytes (13 flits) of the packet. Upon filling the buffer, assert start encryption. Output the full_packet directly from the flipflops continuously.
Decrement time to live	52x8 full_packet	52x8 full_packet	Continuously evaluate to decrement the time to live byte of the full packet
Error Module	start, 4x8 ip, 52x8 full_packet	error	Upon assertion of start, check the packet for errors. In parallel, check for header correctness, checksum, time to live, and that the destination ip == the configured ip. If there is an error, assert error.

Outputter

Module / Submodule Name	Inputs	Outputs	Implementation Notes
Header register	reset, clk, enable_write, 20x8 header_i	20x8 header_o	Upon assertion of enable_write, save the value of header_i into the registers. Store it there outputing continuously.
Data register	reset, clk, enable_write, 32x8 data_i	32x8 data_o	Upon assertion of enable_write, save the value of data_i into the registers. Store it there outputing continuously.
Calculate checksum	8 bit last_digit_of_every_flit	8 bit checksum	Take the last bit of every flit and continuously output the checksum
Delay 13 cycles	reset, clk, error_i	error_o	Store the value of error_i for 13 cycles and then output it on the 13th cycle
Error header generator	20x8 header_i, enable	20x8 header_o	If enable is asserted, change the given header into an error header
NAND fan	32x8 data_i, zeroer	32x8 data_o	If zeroer is asserted, zero all the data using a NAND fan
Serializer	reset, clk, start, 20x8 header, 32x8 data	4x8 flit_o	Upon assertion of start, store the values of header and data in internal registers and begin outputting them every clock cycle one flit at a time

RoundKeyGenerator

Module / Submodule Name	Inputs	Outputs	Implementation Notes
16X8 DEMUX	Two 4X4X8 bit keys, selector	4X4X8 bit key	Basic DEMUX
Pipeline Register	4X4X8 key or data, clk, reset	4X4X8 key or data	4x8 array of delay flipflops, store the state for one cycle while calculations are finishing.
Rot	4X8 word	4X8 word	Rotate the entire word forward and carry around to replace w0 with w3.
Sbox8	dataIn[7:0]	dataOut[7:0]	

Sbox32	dataIn[31:0]	dataOut[31:0]	Perform a constant substitution of one word for another word. This is an array of size 256 which will replace any possible byte value with the byte value given by Rijndael's S-box
Rcon	Counter (4 bits)	4x8 word	RCON[] is a lookup table that is an array of size 8, each element is 4 bits. Based on which round we are on (implemented by the counter) return a value. Values for AES-256 are in "Verilog Design of 256-bit AES Crypto" p 24 ,74. Need to make sure that AES-128 uses a same table.
XOR	Two 4x8 words	4X8 word	Simply xor all bits together
Ripple XOR	4x8 word, 4x4x8 key	4x4x8 key	Contains an XOR module for each of the 4 words. The input word is XORed with the first word of the input key (w0) and this becomes the first word of the new key (w4). Then, w5 is found by xoring w4 with w1. Next, w6 is found by xoring w5 with w2. Finally, w7 is found by xoring w6 with w3.

Encrypter

Module / Submodule Name	Inputs	Outputs	Implementation Notes
Pipeline Register	4X4X8 key or data, clk, reset	4X4X8 key or data	4x8 array of delay flipflops, store the state for one cycle while calculations are finishing.
Sboxes	4x4x8 data	4x4x8 data	Implement an array of 16 Sboxes for each byte of input
SBox	1 byte	1 byte	Described in the lookup table section
ShiftSubByte	clk reset enable dataIn [127:0]	dataOut [127:0]	ShiftSubByte is a combination of SubByte and ShiftRows transformations
MixColumn			
1. xtime	dataIn [7:0]	dataOut [7:0]	xtime means a byte is multiplied by 2, by simply shifting one bit to the right.
2. mixColumn8	enable dataIn_a[7:0] dataIn_b[7:0] dataIn_c[7:0] dataIn_d[7:0]	dataOut[7:0]	
3.mixColumn32	enable dataIn[31:0] dataOut[31:0]	dataOut[31:0]	4 mixColumn8 components are used as for mixColumn32
4.mixColumn128	clk reset enable dataIn[127:0]	dataOut[127:0]	4 mixColumn32 instances are used for complete 128 bits mixColumn128 transformation

AddRoundKey	Sel_addKey[1:0] addkeyIn[126:0] roundKey[126:0]	addKeyOut[126:0]	AddRoundKey is just a 128-bit XOR logical operation between the round key and the state array for each round of the transformation.
Shift Rows	4x4x8 data	4x4x8 data	Simply shift the rows by differing but constant amounts. The first row is not rotated. The second row is rotated to the left by one byte. The third row is rotated to the left by two bytes. The forth row is shifted to the left by three bytes. Because this requires no gates, this should be a very fast operation.
Mix Columns	4x4x8 data	4x4x8 data	

Lookup Tables

Module / Submodule Name	Inputs	Outputs	Implementation Notes
SBox (Subword)			Perform a constant substitution of one word for another word. This is an array of size 256 which will replace any possible byte value with the byte value given by Rijndael's S-box
RCON			RCON[] is a lookup table that is an array of size 8, each element is 4 bits. Based on which round we are on (implemented by the counter) return a value. Values for AES-256 are in "Verilog Design of 256-bit AES Crypto" p 24 ,74. Need to make sure that AES-128 uses a same table.

V. Verification Strategy

Verification and validation plan

To verify that our accelerator design fully implements AES, we will have to demonstrate that ciphertext generated by our design matches ciphertext that is generated by a separate program that implements AES. The algorithm for generating AES ciphertext is well documented, and there exist many implementations of AES in various source languages. However, our design is unique in that it also involves generating a TCP/IP header to encapsulate the data. For that reason, we will have to write our own C source code that generates ciphertext and also the TCP/IP header for each packet. This output will be used in our testing harness to verify that the output from the C model and output from Verilog design match.

Bring up strategy

We will implement AESelerator and the accompanying testbench in stages to allow incremental testing.

Implementation and verification will start with the RoundKeyGenerator module and its submodules. The first major milestone will verify that the RoundKeyGenerator key schedule matches that generated by the testbench. The testbench will stress test the RoundKeyGenerator, subjecting it to initially a small range of possible initial key values, and later a full range of initial key values, to ensure that all logic in the RoundKeyGenerator is covered in the test.

The next major milestone will be implementation of the Encrypter. Again, the testbench will be designed to stress test the Encrypter to guarantee good coverage of all possible cases.

Through this stage, inputs to and outputs from the design under test will not involve TCP/IP packet formats; instead, it will first focus on correct implementation of the AES algorithm.

Implementation of the Inputter module and submodules is next. The first verification milestone will be successful configuration of the AESelerator IP address and initial key. Then, we will implement flit buffering capabilities. The testbench will be modified to pass all configuration data and flits via the TCP/IP interface, while still accepting output as unmodified ciphertext. Once these features are correctly implemented, the testbench will introduce errors in the inputs, and will check that the Inputter is capable of detecting those errors.

Implementation of the Ouputter module and submodules will be the final stage of bring up. The ciphertext will be packaged into flits and outputted in order over the network. The testbench will need to buffer AESelerator's output flits and verify they are correct.

The table on the following pages describes, for each bring up milestone, the AESelerator and testbench required and params.cfg values to create that test.

Bring Up Strategy

Step Number	Functionality	Purpose	AESelerator Inputs involved	AESelerator Outputs involved	Required completed AESelerator components	Required completed testbence components	Testbench params.cfg values	Status
1	Clock and Reset	Ensure that the top level module and the testbench compile together. Ensures that modules implemented after this can have their interfaces routed to the testbench.	clock, reset		AESelerator.v	Interfaces in testbench files.		
2	Counter	Ensure that the counter for keeping track of encryption rounds operates correctly	clock, reset	i=10, i=9, count	Counter.v			
3.1	SBox	Code inspection to ensure that we have configured the static lookup table SBox correctly.			Sbox.v			
3.2	RotateWord()	Code inspection to ensure that the RotateWord() function module is implemented correctly.			Rot.v			
3.3	SubWord()	Code inspection to ensure that the SubWord() function module is implemented correctly.			SBox32.v			
3.4	Rcon []	Inspect that the Rcon[] table is copied over correctly.			Rcon.v			
3	RoundKeyGenerator	Stress test to ensure that our key schedule logic is correct. Testbench should stress test this for 10,000 cycles to guarantee that key generation is accurate.	clock, reset, 128_key, encrypt_begin, Counter (4 bits)	Roundkey, which changes with each clock cycle	All RoundKeyGenerator.v components	Key schedule generation logic. The interfaces of the testbench have to be modified to accommodate for this test.	max_cycles 10000 key_mask from 0000000000000000000000000000000f to ffffffffffffffffffffffffffffffff	
4.1	SubByte()	Ensure that the SubByte() function is implemented correctly and conducts the SBox table lookup correctly.			ShiftSubByte.v			
4.2	ShiftRows()	Ensure that the ShiftRows() function is implemented correctly.			ShiftSubByte.v			
4.3	MixColumns()	Ensure that the MixColumns() function is implmented correctly.			MixColumns.v			
4	Encrypter	Ensure that our encryption logic is accurate. Testbench should stress test this for 10,000 cycles to guarantee that encryption is accurate.	clock, reset, 128_key, encrypt_begin, 16x8_data	16x8_ciphertext	All RoundKeyGenerator.v, Encrypter.v components	AES encryption logic. The interfaces of the testbench have to be modified to accommodate for this test.	max_cycles 10000 key_mask from 0000000000000000000000000000000f to ffffffffffffffffffffffffffffffff data_mask from 000000000000000000000000000000000000000000000000000000000000000f to ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff	

5	Configuration	Ensure that configuring the IP address and first round key works correctly. This test only involves a portion of the design: we pull outputs from the design under test after the ConfigBuffer.	clock, reset, config_i	32_bit_ip, 128_key	ConfigBuffer.v	Logic for deciding on a key and IP address. The testbench has to be able to deliver the configuration settings in 32 bit words per cycle.	max_cycles 10000 density_newPacket 0 density_interruptPacket 0 ipconfig_mask from 0000000f to ffffffff key_mask ffffffffffffffffffffffffffffffff
6	Input buffering	Ensure that the Flitbuffer.v and Inputter.v buffer data flits correctly. This essentially tests that the Inputer, RoundKeyGenerator, and Encrypter operate correctly under the special condition that there are no errors in the input. Furthermore, this test is the first time both encrypter threads will operate at the same time.	clock, reset, config_i, flit_i, flit_valid_i	ready_o, 32x8_ciphertext	ConfigBuffer.v, FlitBuffer.v, all RoundKeyGenerator.v and Encrypter.v components	Logic for deciding two data packets. Logic for creating valid 52 byte packets from the data packet, IP, and checksum.	max_cycles 10000 density_newPacket from 0 to 1 density_interruptPacket from 0 to 1 ipconfig_mask ffffffff key_mask ffffffffffffffffffffffffffffffff destinationip_mask ffffffff timetolive_mask f data_mask fffffffffffffffffffffffffffffffffffffffffffffffffffff fffffffffff
7	Input error checking	Ensure that the error detection logic in the Inputter.v works correctly. This includes checking for IP header format, checksum, TTL, IP addresses.	clock, reset, config_i, flit_i, flit_valid_i	ready_o, error, 32x8_ciphertext	ConfigBuffer.v, FlitBuffer.v, ErrorModule.v, all RoundKeyGenerator.v and Encrypter.v components	Logic for including errors in the IP packet.	max_cycles 10000 density_reset 1 density_newPacket 0 density_interruptPacket 0 density_wrongVersion 0 density_wrongHeaderLength 0 density_wrongTypeOfService 0 density_wrongLength 0 density_wrong45 0 density_wrongProtocol 0 density_wrongChecksum 0 density_wrongDestinationIP 0 ipconfig_mask ffffffff key_mask ffffffffffffffffffffffffffffffff destinationip_mask ffffffff timetolive_mask f data_mask fffffffffffffffffffffffffffffffffffffffffffffffffffff fffffffffff
8	Outputter test with no errors	Test that AESelerator can output its encrypted data in a IP packet format, one flit at a time.	clock, reset, config_i, flit_i, flit_valid_i	ready_o, flit_o	All Inputter.v, RoundKeyGenerator.v, Encrypter.v components. In the Outputter.v, the DataRegister.v, Checksum, HeaderRegister.v, and Serializer.v have to be implemented.	All testbench components.	The error probabilities should be set to zero.
9	Outputter test with all errors	Test that AESelerator generates correct error packets when there are errors in the input.	clock, reset, config_i, flit_i, flit_valid_i	ready_o, flit_o	All components.	All testbench components.	The error probabilities should be set to non-zero values to introduce error in the input.

Verification Parameters

Standard Densities

Parameter name	Type/Length	Description
density_reset	float	How often the reset is asserted
density_newPacket	float	How often a new packet gets sent. This is important for checking how our design behaves under fragmentation.
density_interruptPacket	float	How often we interrupt the sending of the current packet for a cycle
density_interruptConfig	float	How often we interrupt the configuration line.

Header Error Densities

The header error densities are used to validate our error checker. Note that there are byte values that aren't checked as 'wrong' densities. The time to live (byte 8) and the source IP (bytes 12-15) are checked later through a mask. The timetolive must be decremented before it can be checked to be non-zero and the source IP can't ever result in an error from the error_checker.

Parameter name	Type/Length	Description
density_wrongVersion	float	How often the version (first half of byte 0) is not 4
density_wrongHeaderLength	float	How often the header length (second half of byte 0) is not five
density_wrongTypeOfService	float	How often the type of service (byte 1) is not 00000000, 01000000, or 10000000. The first three bytes are precendence and cannot be above 2 and the rest should always be zero
density_wrongLength	float	density_newPacket
density_wrong45	float	How often bytes 4-5 are not zero
density_wrongFragmentation	float	How often the fragmentation (bytes 6-7) is not zero
density_wrongProtocol 0	float	How often the protocal (byte 9) is not 6
density_wrongChecksum 0	float	How often the checksum (bytes 10-11) does not match expected
density_wrongDestinationIP 0	float	How often the destination ip (bytes 16-19) does not match our configured IP

Masks

Parameter name	Type/Length	Description
ipconfig_mask	4 bytes	Mask the IP portion of the configuration. This value shouldn't really matter. Just important to check corners.
key_mask	16 bytes	Mask the key portion of the configuration. Again, just check corners.
sourceip_mask	4 bytes	Mask the source ip part of the header file. Check corners.
timetolive_mask	4 bits	Mask the time to live part of the header file. Check corners and especially when the time to live is 1 or 0.
data_mask	32 bytes	Mask the actual data. 3000 things could go wrong in the encryption so it is very important that we check this thoroughly.

Cycles

Parameter name	Type/Length	Description
max_cycles	int	The cycles we run through before declaring a test passed

VI. Performance Estimates

Pipeline design detailed description

The pipeline design of the AESelerator is determined by the speed at which the network sends packets. This is due to the lack of a backpressure mechanism and the fact that the same circuitry is used ten times in repetition to conduct key generation and encryption.

The AESelerator design does not include a mechanism to provide feedback to the remote host whether or not the AESelerator is ready for the next flit; rather, the sender should assume that the AESelerator can accept and encrypt packets at the speed of one flit per clock cycle, with no gaps in the data transmission.

Furthermore, each round utilizes the same circuitry to conduct key generation and encryption: the data or key is loaded from the pipeline register, manipulated in combinatorial logic, and stored back to the same pipeline register. This design minimizes the design area of the AESelerator by ensuring that the modules in the AESelerator stay occupied on most clock cycles. If the pipeline were any deeper than that (i.e., if each round of key generation and encryption were split into two or more stages), some modules would necessarily be idle in some clock cycles, thereby decreasing power efficiency and increasing area.

Because of these two design concerns, the AESelerator must complete 10 rounds of key generation and encryption within 13 clock cycles. If it took any longer than that, the AESelerator would not be able to keep its input buffer empty give that packets arrive from the network at the maximum specified bandwidth of one flit per clock cycle.

Encryption stage speed estimate

We predict that the longest path in our design will from the encrypter pipeline register, through the encrypter combinatorial logic, and back to the encrypter pipeline register. The time it takes for this path to execute will determine how long one clock cycle has to be.

The key generator module has a similar long pathway from the pipeline register back to the pipeline register; however, we predict that the path in the encrypter will have to pass through more logic gates because the mix columns operation requires scalar multiplication on each element of the state array.

This pathway traverses roughly double the number of gates than the Execute stage pathway in the MIPS pipeline, which took 5 ns to complete on 90 nm technology. Therefore, we estimate that it will take 10 ns to complete this path, amounting to a **100 MHz clock speed**.

Using these estimates, the throughput of the AESelerator would be (10x8 cycles/second) / (13 cycles/packet) * (2 codewords/packet) * (128 bits/codeword) = **1.97 gigabits / second**.

VII. Area Estimates

Overview

The storage elements for the pipeline registers and the lookup tables will occupy the most area in the AESelerator design. We estimate that the combinatorial logic will be no larger than the sequential logic, so we will estimate the amount of sequential logic needed, and double that number for a conservative estimate of how much area the final AESelerator design will require.

Sequential logic size estimate

Area estimates in bits of memory elements for each of the most space consuming storage elements is listed below:

Storage element name	**Storage element size**	**Number of replicas in design**	**Total size (in bits of memory)**
SBox	16*16*8 bits	16+16+4 = 36	73728 bits
Counter	8 bits	1	8 bits
Encrypter pipeline register	16 * 8 bits	2	256 bits
Key Generator pipeline register	16 * 8 bits	1	128 bits
ConfigBuffer	20 * 8 bits	1	169 bits
FlitBuffer	13 * 4 * 8 bits	1	416 bits
Delay 13 cycles	1 bit	1	1 bits
Data register	32 * 8 bits	1	256 bits
Header register	20 * 8 bits	1	160 bits
Serializer	13 * 4 * 8 bits + 8 bits counter	1	424 bits
Total			**75,072 bits**

Overall logic size estimate

As described in the overview, we double the sequential logic size estimate to obtain the overall logic size estimate. Using the figure above, we arrive at the estimate that AESelerator will occupy **150,000 units of area**, where each unit is the area of one bit of memory.

VIII. Implementation Final Report

Implementation status

Successes

The round key generator and the Encrypter have been compiled and stress-tested to great success. Under regular stress testing using completely random keys and plaintext, after a few thousand cycles in a few dozen run, the ciphertext outputs of the implementation matched the expected ciphertext outputs of the test bench 100% of the time. In addition, we have masked the inputs to corner cases (all zeros, all ones, all zeros in the key and all ones in the plaintext, et cetera) and still achieved 100% matching outputs. Therefore, we have achieved 100% code coverage in the Encrypter and the RoundKeyGenerator modules.

The full AESelerator is compiled and running. Inputs are given sequentially and outputs are received sequentially. The timing matches perfectly over the full 39 cycle data input to ciphertext output.

We adhered to coding conventions that guaranteed the design would be synthesizable. The source code for the chip passes Leda static analysis. The RTL design was successfully synthesized in DC compiler and ICC. The synthesis results are reported below, and the log files for circuit synthesis are attached.

Almost successes

The partially implemented portions of the AESelerator have to do with error handling and testing the AESelerator with arbitrary pauses in the data transmission.

The logic for creating and handling error conditions exists in the AESelerator hardware implementation and in the testbench, but has not been tested. Because of this, we have not achieved full code coverage in the ErrorModule.

The logic for creating arbitrary pauses and any unusual timing conditions is not fully implemented in the testbench. The machinery for handling unusual timing conditions exists in the hardware implementation, but it is assumed that much timing adjustments are necessary for it to be fully functional. As of now, the test bench sends a reset, followed by the configuration, followed by repeated packets with no pauses or overlaps in the inputs.

Bugs and resolution

Bug description	Observed output	Potential cause	Resolution
Key generation exits too soon	Incorrect final round key is generated. However, this key is one of the keys in the key schedule.	Counter stopping round key generation too soon.	Resolved. Modified counter timing.
No encryption occurs	Encryption output is indeterminate.	Incorrect wiring of encrypter submodules.	Resolved. Two buses from from MixColumns and ShiftSubBytes were wired directly to PipelineRegister, where they should have been connected with a gate.
MixColumns output incorrect	Encryption data fails passing through the MixColumns stage.	Incorrect XORing of data in MixColumns.	Resolved.
Flits sent from outputter in reverse order	Flits are returned starting from the ciphertext, and are in exactly reverse order.	Incorrect indexing of Outputter's Serializer component.	Resolved. The meaning of the selector input to the Mux was misinterpreted.
Encryption results returned too soon	Flits are returned one clock cycle sooner than the verification test bench anticipates them.	Incorrect FinalBuffer output delay timing.	Resolved. The FianlBuffer now uses a timing signal one clock cycle later than original.
Ciphertext output completely incorrect once data is made random	Ciphertext becomes incorrect once we remove the constraint on using known data.	The output of the two encrypter pipelines, data0 and data1, are reversed.	Resolved. Two outputs were reversed, and all ciphertext output passes testing. This was not caught earlier because before this point the two data packets were identical.
Test runs with time to live errors introduced cause output to mismatch	Encrypter outputs data packets of zero due to perceived error condition, however, testbench reports no error occurred.	Misinterpretation of how error mechanism should be timed.	Resolved. The ErrorModule was decrementing the TTL value, then determining the checksum. The design requires the checksum to be determined first instead.

Corner cases

We ran the AESelerator for 25,000 cycles on the following special conditions to verify that the design performs well on corner cases:

	Passed	**Waveform screencap**
IPs are set to zero	Yes, the last two flits of the header (IPs) are returned as all zeros.	
Initial keys are set to zero	Yes, the key generator produces correct round keys.	
Data are set to zero	Yes, the encrypter correctly encrypts data that is all zero.	

Code coverage

We ran the testbench for 50,000 cycles with the keys, data, and IP settings all set to full random. The testbench is instructed to exit and fail if any output is not what is expected. The design did pass the test, which means we have excellent confidence that the key generation and encryption code in both the hardware design and in the C model are correct.

```
--------------------Cycle 50000--------------------
Expecting from ready_o:0 Actually got from ready_o:0
Expecting from flit_o:e074f49e Actually got from flit_o:e074f49e
PASSED: Test ended after 50000 cycles(all checks passed.)
```

Go up
SCORE LINE
90.74 90.74 bench.dut
SCORE LINE
100.00 100.00 counter
97.29 97.29 encrypter0
subtree...

SCORE LINE
97.29 97.29 encrypter1
subtree...

SCORE LINE
98.79 98.79 inputter
SCORE LINE
98.39 98.39 config_buffer
subtree...

SCORE LINE
100.00 100.00 decrement_ttl
98.95 98.95 flit_buffer
subtree...

Code coverage screen cap

Physical compilation notes

The design compiled in DC compiler using four different compilation settings to determine a range of timing and area estimates. Errors that cropped up during DC compilation were resolved, and the remaining warnings are all benign—most are warnings against connecting input ports to output ports, or input ports, such as reset and enable, that are hardwired to logical 1 or 0.

The design compiled in ICC compiler as well.

Timing report

Bandwidth and latency

	Timing value
Maximum input bandwidth	AESelerator can receive data as fast as one full packet every 13 clock cycles. If the first packet is transmitted to AESelerator at clock cycle 22 (the earliest at which data input is allowed, see "AESelerator configuration time" below), the first packet can finish transmission on clock cycle 34 at the earliest. Then, there can be as little as no delay between the end of the first packet and the beginning of second packet.
Maximum output bandwidth	AESelerator returns ciphertext at the rate of one full packet every 13 clock cycles. If the first packet begins returning at clock cycle 49, that packet will finish returning at clock cycle 61.
Minimum input bandwidth	In our design, AESelerator can handle arbitrary pauses in the input stream, both within the flits of the same packets, and between packets. However, this functionality has not been tested.
Minimum output bandwidth	In our design AESelerator continuously outputs flits of the same packets once it output transmission begins. When there were delays in the input, the output remains idle until the next packet is ready for output.

	However, this functionality has not been tested.
AESelerator configuration time	The CPU should assert reset for one cycle, and transmit configuration data for the next 20 cycles. Configuration completes at clock cycle 21. The ready_o signal is picked up at the testbench on clock cycle 22.
Encryption latency	AESelerator returns the first packet of the ciphertext at 14 cycles after the input completed. For example, if the last flit of the first packet was transmitted at clock cycle 34, the first header flit of the return packet is transmitted at clock cycle 49.
Error packet latency	The error packet has the same latency as a successful encryption's packet. If the last flit of the first packet was transmitted at clock cycle 34, the first header flit of the error packet is transmitted at clock cycle 49.

Clock speed

The design was compiled once with max standard cells with flattening turned off, and three times with flattening turned on using the max, typ, and min standard cells. Timing for each compilation follows.

	Critical path time	**Clock speed**	**Bandwidth**
No flattening, max cells	6.93 ns	144.3 MHz	2.84 gigabits / second
Flattening, max cells	3.90 ns	256.4 MHz	5.05 gigabits / second
Flattening, typ cells	3.73 ns	268.1 MHz	5.28 gigabits / second
Flattening min cells	3.68 ns	271.7 MHz	5.35 gigabits / second

This clock speed exceeds the estimated clock speed before implementing our design. Our original estimate was 100 MHz, which was made anticipating that there would be a long critical path in the encryption stage going from the pipeline register, through many gates, and back to the pipeline register again. The Encrypter circuitry turned out to be simpler than expected.

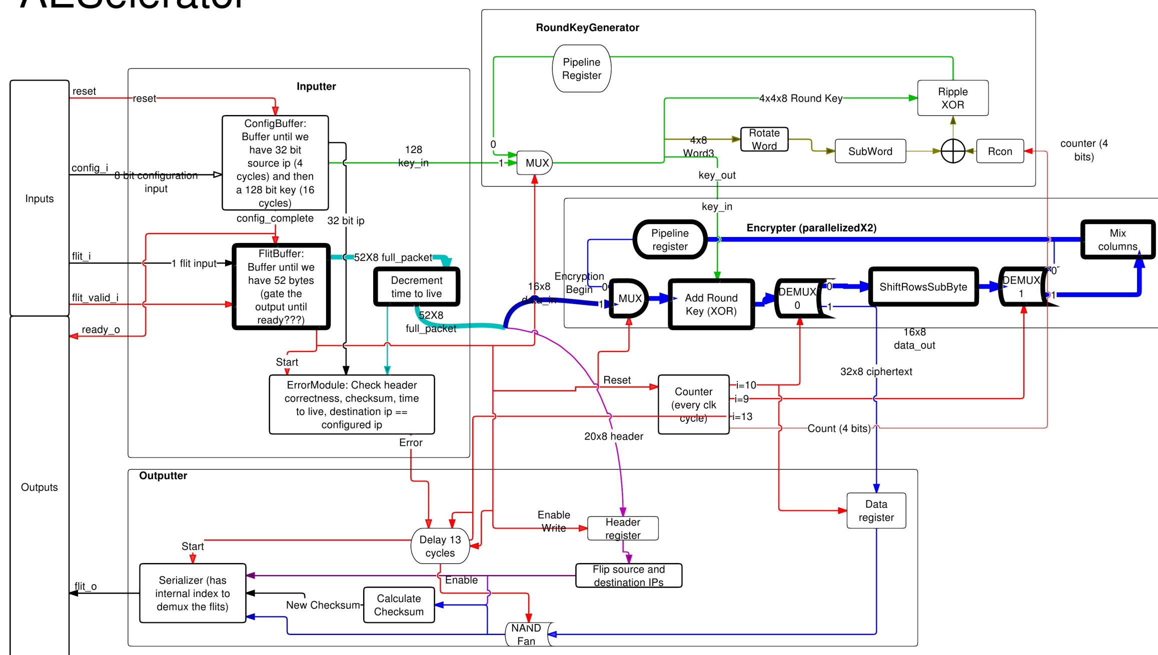
A sample calculation follows. At 256.4 MHz, the throughput of the AESelerator is (2.564x8 cycles/second) / (13 cycles/packet) * (2 codewords/packet) * (128 bits/codeword) = 5.05 gigabits / second.

Critical path

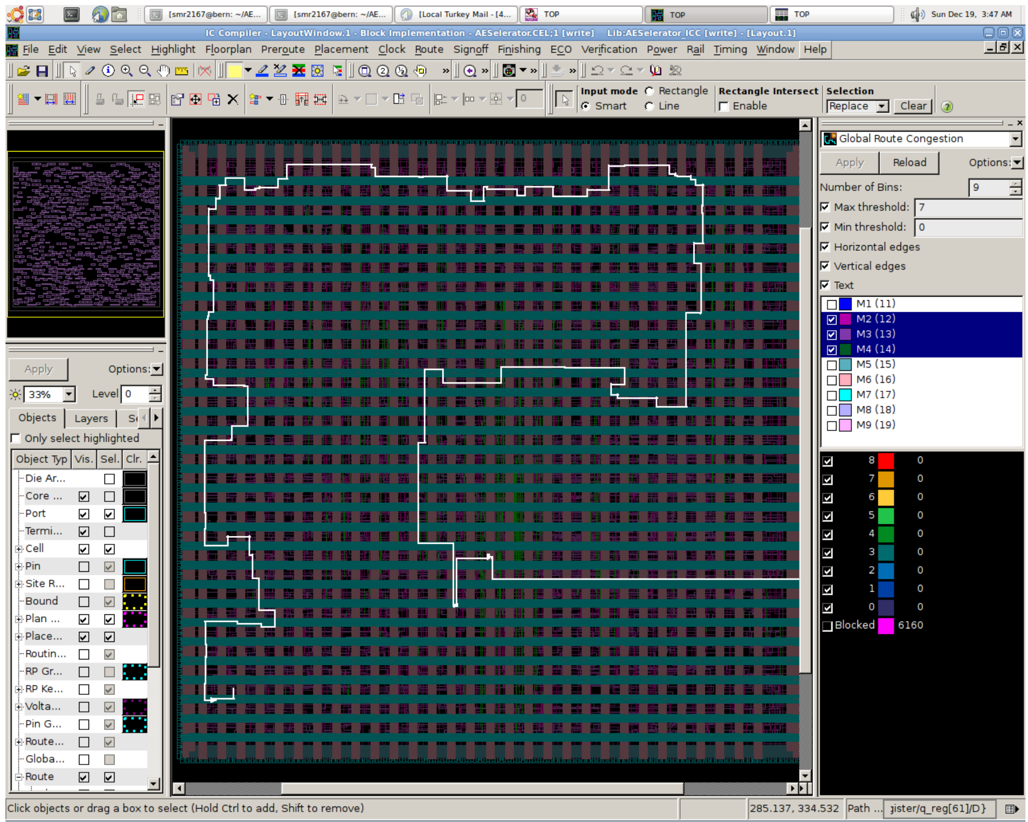
The critical path in our design originates from the flit buffer in the Inputter module, into the Encrypter module, and loops once through all the encryption logic before terminating the PipelineRegister.

The following figure traces this critical path on our block diagram.

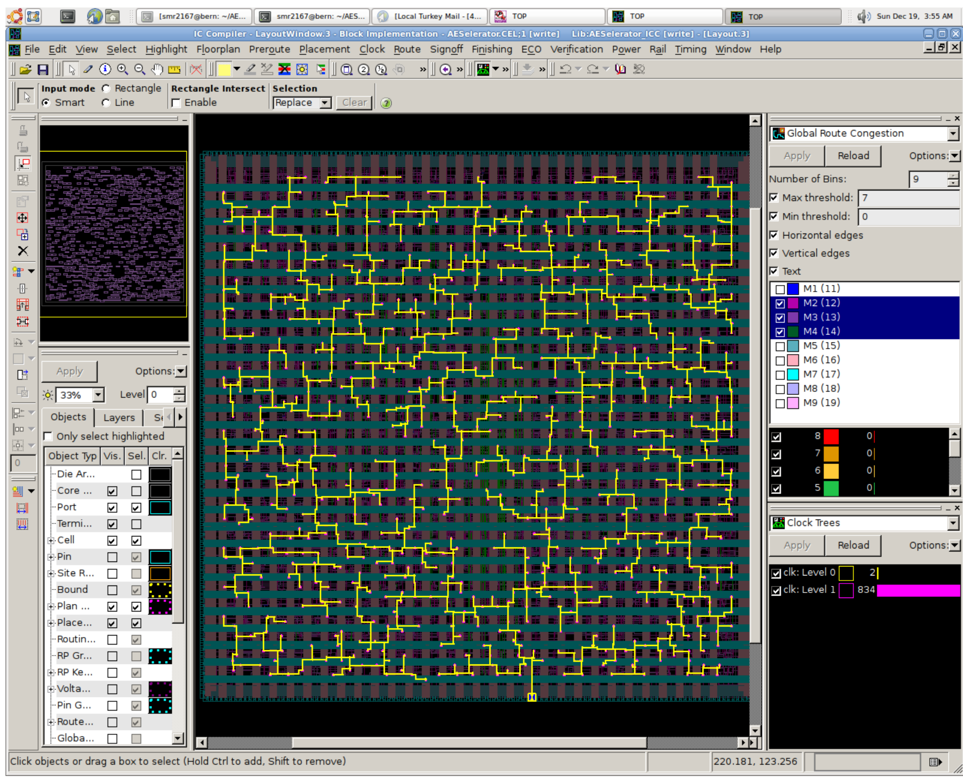
AESelerator



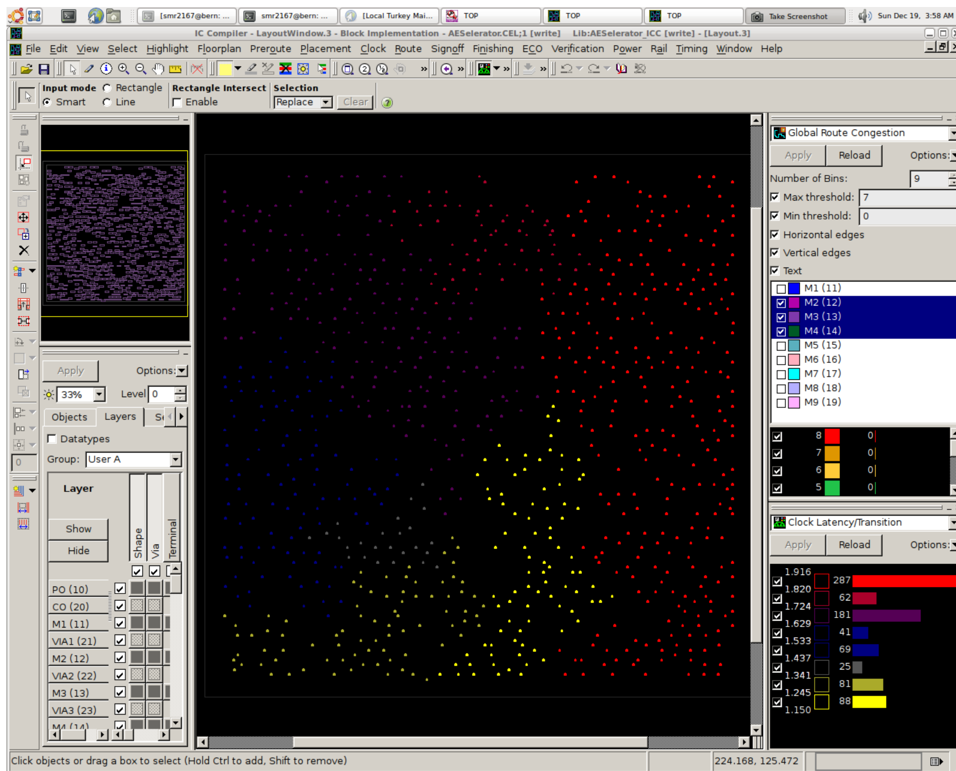
The following diagrams taken from ICC compiler show the clock tree and longest path. Please note that due to recompilations of the source code the diagrams may not be the images of the latest version of the implemented design.



Longest path



Clock tree



Clock latency

Potential optimizations

In order to improve the timing performance of AESelerator, we would remove the long critical path from the Inputter through the Encrypter. In order to do this, we would change our timing scheme by feeding the data directly into the PipelineRegister, instead of having it loop once through the encryption logic. This would cut the critical path length in half. The data latency would be one cycle longer, but in turn gain a higher bandwidth.

Area report

	Combinatorial area	Noncombinatorial area	Net interconnect area	Total cell area	Total area
No flattening, max cells	291164	45179	21962	336344	358306
Flattening, max cells	280727	44407	18776	325134	343911
Flattening, typ cells	286580	44732	19149	331313	350462
Flattening min cells	280194	44732	18982	324926	343909

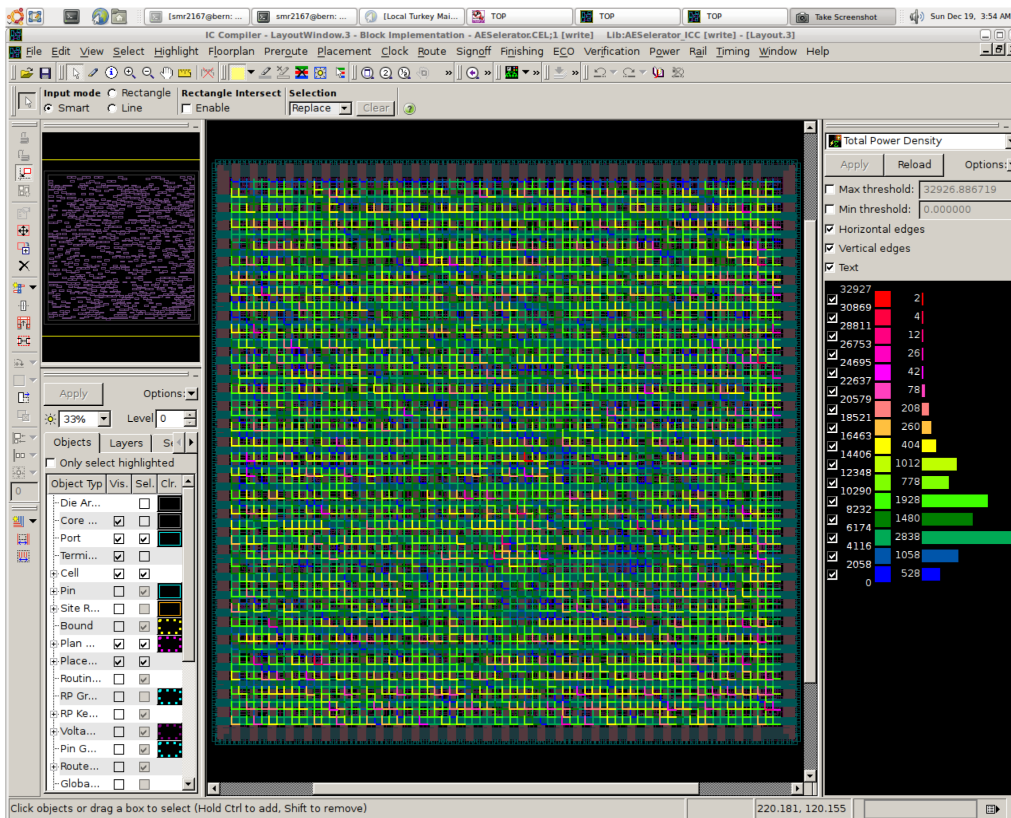
The final reported area is more than double the estimated area, assuming that each unit of area reported by DC compiler is equivalent to one bit of memory. As expected, combinatorial components occupy the majority of space on the chip.

Power report

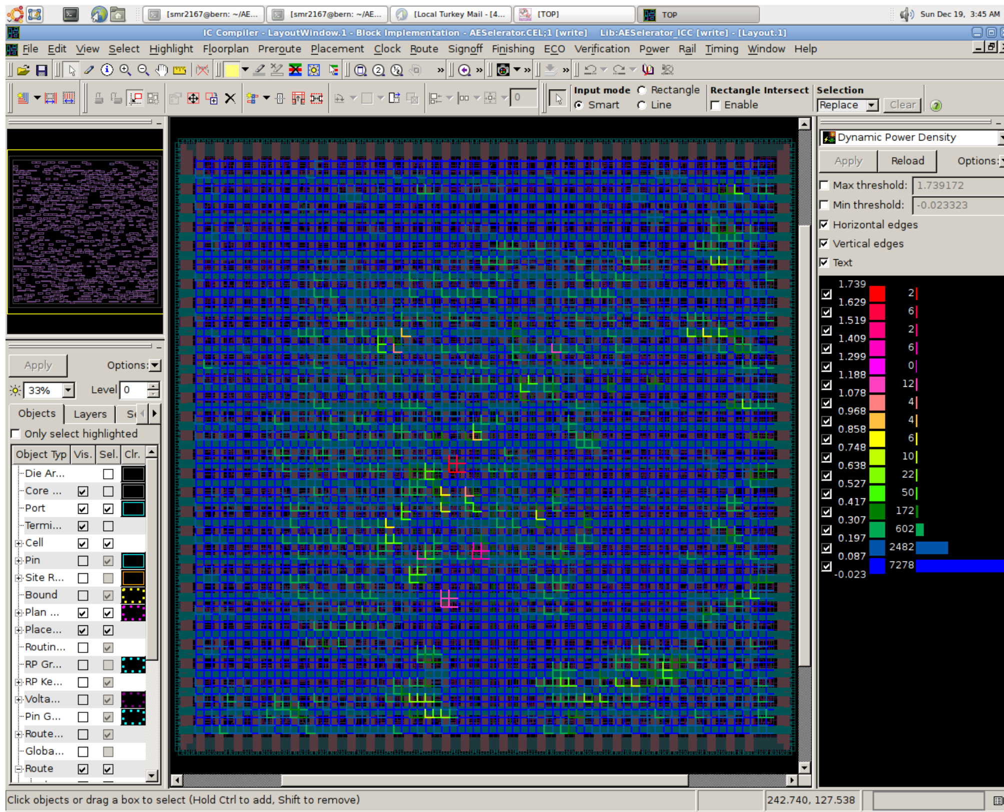
Diagrams concerning the power consumption of AESelerator can be found at the end of this section. The majority of power consumption on the AESelerator comes from leakage power. The dynamic power consumption is small, and is concentrated around small groups of logic.

Potential optimizations

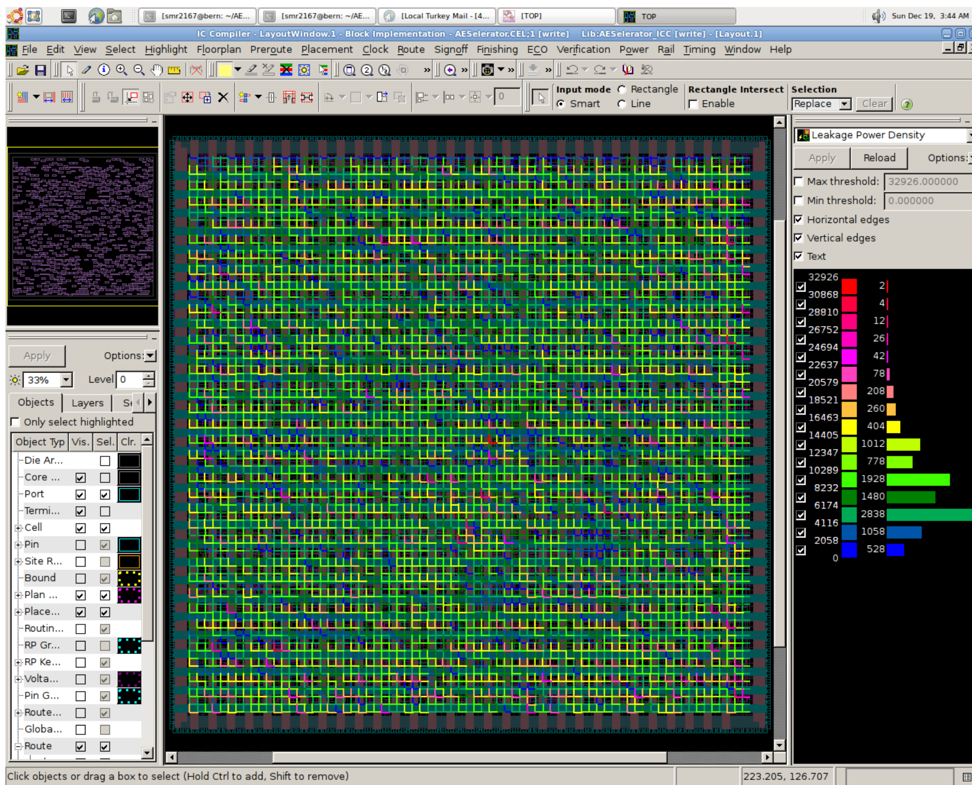
There is room for improvements to the design to minimize the total power consumption. In the current design, the Encrpyter and RoundKeyGenerator units continue functioning even if the Inputter detects an error condition. As an improvement to decrease power consumption, we may choose to disable these units completely by freezing data in the pipeline registers, and the clock signal to these modules may be disabled when they are not needed.



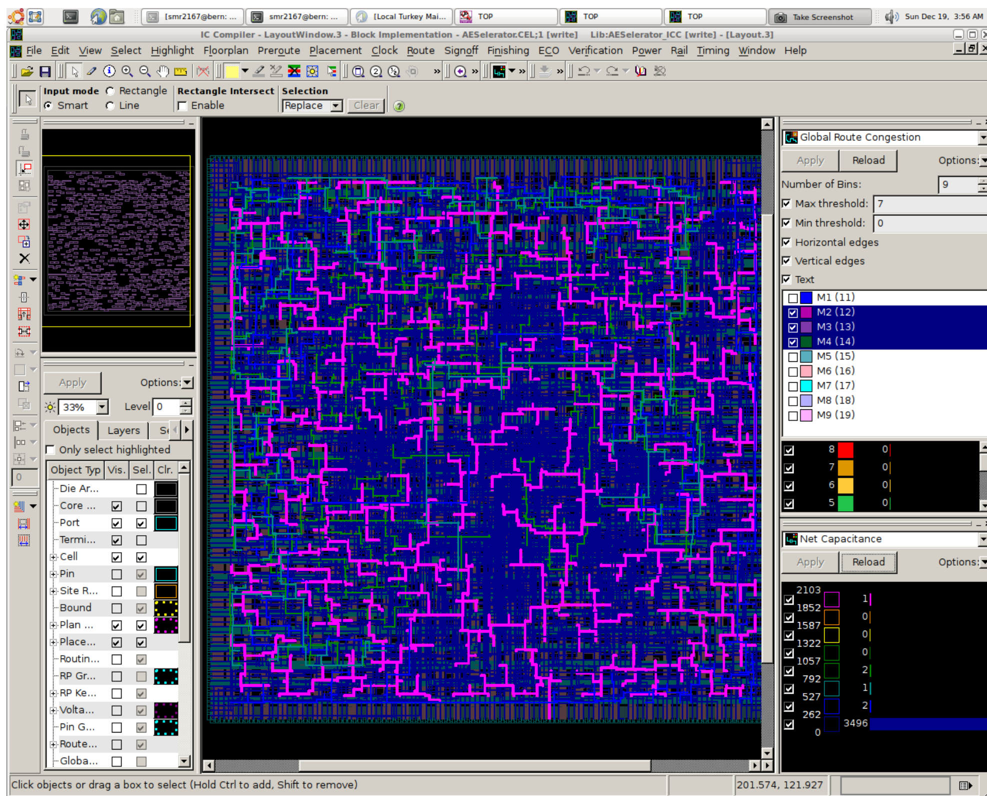
Total power distribution



Dynamic power distribution



Leakage power distribution



Net capacitance

IX. Document Revision History

November 6, 2010

Document established

November 7, 2010

Completed Design Overview

November 14, 2010

Completed Team Member Responsibilities, Unit Level Interfaces, Test Harness Structure

November 28, 2010

Added Detailed Microarchitecture, Bring-Up Strategy, Performance Estimates, and Area Estimates. Revised microarchitecture diagram.

November 30, 2010

Completed design documentation for phase 3 review. All chapters except bug resolution completed.

December 20, 2010

Added final implementation documentation.

December 21, 2010

Amended final implementation documentation