Hardware Encryption for Embedded Systems

Yiheng Yang

Semester's Thesis SA-2012-16

November 2012 until February 2013

Advisor: Ariane Keller, ariane.keller@tik.ee.ethz.ch
Co-Advisor: Dr. Stephan Neuhaus, stephan.neuhaus@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

Abstract

Future communication networks are facing much challenges in that high performance, low cost, and low power consumption for network programming are desired. Most importantly, an increasing demand for flexibility has asked for a dynamic communication node architecture. Though software implementation offers great flexibility, it suffers from low throughput rate. While a state-of-art ASIC implementation guarantees an optimized performance, cost and power consumption, it provides little flexibility since all the functions are fixed once the chip is fabricated.

A reconfigurable hardware/software platform in which the modules of the node's network stack are dynamically distributed at runtime, is promising to tackle the challenge of future Internet applications. This semester project is in the context of such a networking node architecture, called ANA (Autonomic Network Architecture)[1]. As encryption is an important means to achieve confidentiality in network communication, it is therefore, of our interest to realize and test an encryption engine in this introduced architecture. In this semester thesis, we select AES (Advanced Encryption Standard)[2] and implement it as a hardware thread in the embedded system to test and show the benefit of the system architecture.

Acknowledgements

Thanks to my project advisors Ariane Keller and Dr. Stephan Neuhaus who greatly helped me during the entire semester. Thanks to the Communication Systems Group at Computer Engineering and Networks Laboratory for providing me the chance to do the semester project.

Yiheng Yang, Feb 25, 2013

Contents

1 Introduction 9
1.1 Motivation . . . 9
1.2 Outline . . . 9

2 Related Work 11
2.1 ANA . . . 11
2.2 ReconOS . . . 11

3 A Brief Introduction to AES 13
3.1 Key Expansion . . . 13
3.2 Data Encryption . . . 14
3.2.1 sbox substitution . . . 14
3.2.2 shiftRows transformation . . . 14
3.2.3 mixColumn transformation . . . 14
3.2.4 addRoundKeys . . . 15
3.3 Data Decryption . . . 15

4 Architecture Design and Implementation 17
4.1 Hardware System Architecture . . . 17
4.1.1 Packet Structure . . . 18
4.1.2 Packet Decoder and Packet Encoder . . . 18
4.2 AES Encryption Engine . . . 19
4.2.1 AES Hardware Thread Top Module . . . 19
4.2.2 AES Core . . . 20
4.3 AES Decryption Engine . . . 24
4.4 Software Control Interface . . . 24
4.5 Implementation . . . 24

5 Validation and Evaluation 27
5.1 Simulation . . . 27
5.1.1 Test of AES-128 . . . 27
5.1.2 Test of AES-192 . . . 29
5.1.3 Test of AES-256 . . . 30
5.2 Validation . . . 30
5.3 Evaluation . . . 31
5.3.1 Synthesis Report . . . 32
5.3.2 Speed . . . 33

6 Conclusion and Future Works 37

A How to Use 39
A.1 System Configuration 39
A.1.1 Install XILINX 13.3 tool chain 39
A.1.2 Install Microblaze GNU Tools 39
A.2 Hardware System Synthesis 40
A.2.1 Package 40
A.2.2 Top Module Connections 40
A.2.3 Compilation 40
A.3 How to Configure hwt_aes 40
A.4 Compile and Download Linux Kernel 41
A.5 Operation in minicom 41

B AES Engine Architecture 43

C Task Description 45

D Declaration of Originality 49

E Time Line 51

References 53

List of Figures

2.1 ReconOS system architecture 12

4.1 Hardware Architecture 17
4.2 Packet Structure 18
4.3 Hardware Thread Top Module 19
4.4 Top FSM 20
4.5 AES Core 21
4.6 Encryption Stage 23

5.1 AES-128 round key 28
5.2 AES-128 encryption result 28
5.3 AES-128 decryption result 28
5.4 AES-192 round key 29
5.5 AES-192 encryption result 29
5.6 AES-192 decryption result 30
5.7 AES-256 round key 30
5.8 AES-256 encryption result 31
5.9 AES-256 decryption result 31
5.10 Validation Platform 32
5.11 Validation Result 34
5.12 Evaluation Architecture 35
5.13 Clock Cycles for one Packet 35
5.14 Maximum Packet Sending Rate 36

Chapter 1

Introduction

1.1 Motivation

This semester thesis is in the context of the EPiCS project. The goal of the EPiCS project is to lay the foundation for engineering the novel class of proprioceptive computing systems. Proprioceptive computing systems collect and maintain information about their state and progress, which enables self-awareness by reasoning about their behaviour, and self-expression by autonomously adapting their behaviour to changing conditions[3].

In this thesis we focus on the networking aspect of EPiCS which we call EmbedNet. EmbedNet uses the concepts of the network architecture developed in the ANA project as a basis. The ANA network is a novel architecture that enables flexible, dynamic, and fully autonomous formation of network nodes. The need for changing the protocol stack can arise if networking functionality needs to be patched, if the used encryption method is not considered safe anymore, or when privacy concerns change[4][5]. As encryption is an important means to achieve confidentiality in network communication, it is therefore, of our interest to realize an encryption engine in this introduced architecture and see how much it could benefit from hardware accelerators[6]. The objective of this semester thesis is to implement an encryption protocol that can be executed in hardware. It is required that the encryption module can be dynamically inserted in a given protocol stack and the key can be set from software at any time.

1.2 Outline

The scope of the semester thesis includes the familiarization of the general ideas and concepts of ANA[1], EmbedNet and ReconOS (Reconfigurable Operating System)[7]. We focus on the design and implementation of the encryption engine by using ReconOS, under the structure of EmbedNet. Validation and evaluation are carried out using ReconOS as well. The remaining chapters are organized as follows. Chapter 2 reviews related works including Autonomic Network Architecture and Reconfigurable Operating System, on which we built our architecture. A brief introduction of Advanced Encryption Standard (AES)[2] is presented in Chapter 3. The architecture design and implementation are detailed in Chapter 4, and Chapter 5 describes the validation platform and evaluation process. We conclude and discuss future works in Chapter 6. Also, a "How to Use" is detailed in Appendix A.

Chapter 2

Related Work

This semester thesis is in the context of EmbedNet project. EmbedNet uses the concepts of the network architecture developed in the ANA project as a basis. The ANA network architecture is a novel architecture which enables flexible, dynamic, and fully autonomous formation of network nodes[3].

2.1 ANA

In general, the ANA network architecture is organized in network compartments. In each compartment, a distributed set of protocol entities collaborate in order to provide communication services to other compartments[1]. Each compartment operates as a black-box supporting a generic communication API, which describes all communication between different networking functions and is based on a publish/resolve architecture.

In order to provide high flexibility, ANA divides networking functionality into functional blocks (FB)[1]. The dynamic combination of FB relies on the concept of indirection and is realized with so-called information dispatch points (IDPs). For a certain operation, data would be sent to an IDP that is bound to a FB. This layer of indirection allows the dynamic adaption of network stack by changing the bindings of IDPs. Sending a packet from one FB to another is aided by a central place called minmex. Ana can be extended so that FBs can be implemented either in hardware or software, or both. This architecture provides us with the flexibility to decide at run time which FB should be executed in hardware and which in software, based on the packets being processed.

2.2 ReconOS

Our architecture not only depends on ANA for the mechanisms to assemble functional blocks into flexible protocol graphs, but also relies on the ReconOS operating system to realize transparent and efficient communication between FBs, be it implemented in hardware or software.

ReconOS is a programming model, an execution environment, an operating system extension, and a hardware architecture. It is a way to bring some of the convenience of a software-like programming model to the detail-ridden world of dynamically reconfigurable hardware design[8]. With ReconOS, one can model a concurrent application for reconfigurable systems-on-chip (rSoC) using both software and hardware threads. The interactions between all threads are handled through common POSIX-like abstractions such as mailboxes, semaphores, or shared memory, hiding the complexities of bus access protocols, memory spaces, register files

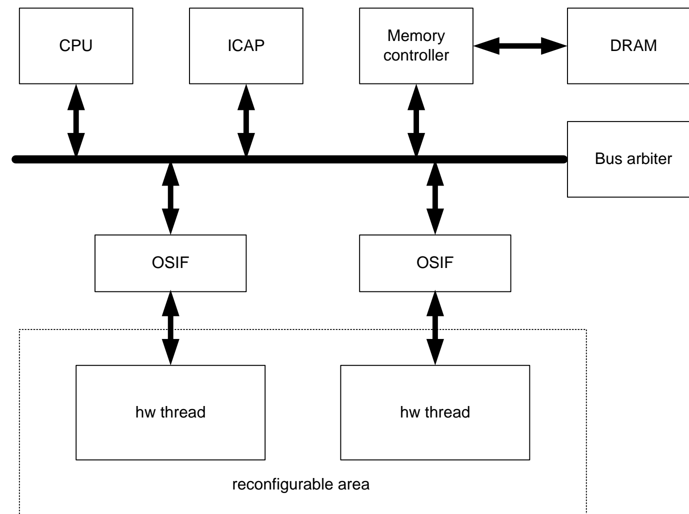


Figure 2.1: ReconOS system architecture

and interrupt handling. Figure 2.1 shows the architecture of a simple ReconOS system with two slots. The "OSIF" in the picture denotes the operating system interface. Besides the main CPU and the operating system interfaces, the main bus also connects a DRAM controller and the FPGAs internal configuration access port (ICAP), which supports dynamic partial reconfiguration[9].

Chapter 3

A Brief Introduction to AES

In this chapter, Advanced Encryption Standard (AES) is briefly introduced. AES is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001[2]. This standard specifies the Rijndael algorithm[10], a symmetric block cipher that can process data blocks of 16 bytes, using cipher keys with lengths of 128, 192, and 256 bits.

The data encryption process includes two major steps in general: (1) generate the round keys; (2) encrypt data blocks. To encrypt a plain text, we need round keys first. Round keys are generated from the given original cipher key. The process of this generation is called “Key Expansion”. The following data encryption process includes several rounds, of which each contains several steps. The plain text data block is treated as a 4×4 bytes of matrix. Each step is a function on all 16 bytes of the matrix.

3.1 Key Expansion

As indicated above, the AES algorithm may be used with three different key lengths, which may be referred to as AES-128, AES-192, and AES-256. The AES-128 needs 10 rounds to encrypt the input data block, each of these 10 rounds are separated into 4 steps (round 10 needs only 3 steps). One of the 4 steps is to XOR the processing data block by a 128 bit round key. So that we need to generate 10 round keys before we start to encrypt data.

The round keys are obtained from the original cipher key using a certain algorithm. The generation of round keys for different key lengths are similar. The Key Expansion generates a total of $Nb(Nr + 1)$ words: Nb is the length of the original cipher key, each of the Nr rounds requires Nb words of key data[2]. The resulting key schedule consists of a linear array of 4-byte words, denoted $w[i]$, with i in the range $0 \leq i < Nb(Nr + 1)$. So for the expansion of a 128-bit cipher key, we need 10 rounds to generate all 44 ($Nb = 4, Nr = 10$) words of the round keys. Following is the pseudo code for key expansion[2]:

```
word tmp
    i = 0;
    while (i != Nk)
    w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3]);
    ++i;
    end while

    i=Nk;
```

```
while(i!=Nb*(Nr+1))
tmp=w[i-1];
if (i mod Nk=0)
        tmp = sbox (shiftWord(tmp)) xor rcon[i/Nk];
elsif (Nk >6 and i mod Nk=4)
        tmp =sbox (tmp);
endif;
w[i] = w[i-Nk] xor tmp;
++i;
end while
```

sbox() is an operation to replace the word by another in a look up table. shiftWord() takes a word $[a_0, a_1, a_2, a_3]$ as input and returns the word $[a_1, a_2, a_3, a_0]$. For $1 \leq i \leq Nr$, the round constant word array[2], $rcon[i]$, contains the values $[2^{i-1}, \{00\}, \{00\}, \{00\}]$, where multiplication and exponentiation are done in the finite field with 2^8 elements, called the *Galois field with* 2^8 *elements*, or $GF(2^8)$.

3.2 Data Encryption

The data encryption process is more direct, compared to the key expansion. Total rounds required to cipher a plain text entirely depends on the key size itself. For AES-128, it takes 10 rounds to encrypt the data block. Similarly, we need 12 or 14 rounds to encrypt the same data block if using AES-192 or AES-256. Round 0 is only a XOR operation between the plain text block and the cipher key itself. The rest rounds contain 4 steps each: (1) sbox (replace each byte of the data block by another byte denoted in a look up table, or to calculate the value during run time); (2) shiftRows (shift different rows with different amounts); (3) mixColumn (multiply the data matrix by a certain matrix in GF (2^8) field); (4) addRoundKeys (a XOR operation on the data block using a corresponding round key). The final round contains only step (1), (2), and (4).

3.2.1 sbox substitution

The first step in one complete round is a sbox substitution, which provides the non-linearity in the cipher. Every byte in the data matrix would be substituted by a corresponding value in a look up table, the Rijndael S-box[10]. As an alternative, this value could be calculated during run time. Nevertheless it would severely effect the critical path, unless the calculation itself is pipelined.

3.2.2 shiftRows transformation

The second step is shiftRows transformation. The first row of the 4×4 byte matrix is not shifted. The second row is cyclically shifted left by 1 byte. The third row is cyclically shifted left by 2 bytes. Similarly, the last row of the matrix is left shifted by 3 bytes.

3.2.3 mixColumn transformation

The third step involves a matrix multiplication in GF(2^8) field. All bytes in the AES algorithm can be presented as the concatenation of its individual bit values in the order

$$\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}.$$

These bytes are interpreted as finite field elements using a polynomial representation:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0.$$

For instance, a four-term polynomials can be defined with coefficients that are finite field elements as:

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0.$$

Let

$$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0.$$

The multiplication is then achieved in two steps. First, the polynomial product

$$c(x) = a(x) \bullet b(x)$$

is algebraically expanded, and like powers are collected to give

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0.$$

The result, c(x), does not represent a four-byte word. So the second step of the multiplication is to reduce c(x) modulo a polynomial of degree 4. For the AES algorithm, this is accomplished with the polynomial

$$x^4 + 1,$$

so that

$$x^i mod(x^4 + 1) = x^{i \bmod 4}.$$

For AES algorithm, each column of the matrix is multiplied with a given matrix and is replaced by the answer obtained. The matrix multiplication is as follows:

$$\begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}$$

Together with shiftRows, mixColumns transform provides diffusion in the cipher[2].

3.2.4 addRoundKeys

The last step in a round is addRoundKeys. This is a XOR operation between each byte of the data matrix and a corresponding byte in the key matrix.

3.3 Data Decryption

The decryption process is similar to the encryption process. We do all the transformation in a reversed order. For example, for an AES-256 decryption, the first round still contains an addRoundKeys operation. Nevertheless the round key used for is the last round key produced in key expansion process. Next we enter 13 major rounds, each of them contains (in order): invShiftRows, invSbox, addRoundKey, and invMixColumns. Round 14 does not involve an invMixColumns operation. The only difference between invShiftRows with shiftRows performance is the direction,

we now cyclically shift the bits towards right. For invSbox, we replace the sbox matrix by an inverse one. Though possible to calculate the substitute byte during run time, it's more convenient to use a look up table as we introduced before. Then we use round keys in a reversed order for each round, e.g. for the final round, we use roundKey[0], the cipher key itself as the round key. As for the invMixColumns transformation, we employ an inverse matrix:

$$\begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix}$$

It guarantees that the key expansion process remains the same. Yet a slight modification on the encryption pipeline is necessary (as will be introduced in the next chapter). Also there's another option for the decryption process, see [2]. Then we need to slightly change the key expansion process as well. If an extremely high throughput AES encryption circuit with relatively mild area consumption is desired, unroll the sbox substitution and deepen the pipeline, as is detailed in [11].

Chapter 4

Architecture Design and Implementation

This chapter illustrates the architecture design and the implementation of the AES encryption engine aided by the features in ReconOS[9][12]. The AES encryption engine includes a top module, which consists of a packet decoder, a packet encoder, and the AES Core. The AES Core is then divided into several blocks: (1) a control block; (2) a keyExpansion block; (3) a readStage block; (4) a sendStage block; (5) four encryption blocks which work as a pipeline. Further, there is a keyBakup block which enables the data encryption and key expansion process in parallel. But before diving into the detailed discussion of encryption engine's architecture, it is necessary to briefly introduce the entire system architecture at the beginning. The hardware system, built by Richard Huber, was detailed in his Master thesis[13]. The maximum frequency of the system is 125 MHz, while using a 9-bit data bus for the traffic.

4.1 Hardware System Architecture

The entire hardware system resides on a single chip. As a prototype, we have two switches for the NoC (Network on Chip), each of them supports two functional blocks. See Figure 4.1.

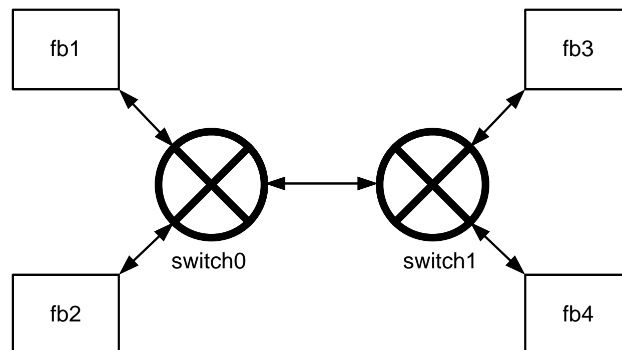


Figure 4.1: Hardware Architecture

The AES encryption engine is one of the four functional blocks. For the functional validation of this semester thesis, the rest being used are a Software to Hardware (s2h) functional block, a Hardware to Software (h2s) functional block, and an Eth-

ernet (eth) functional block. The AES functional block receives packets from the s2h (Software to Hardware) block, encrypts the data, and sends the ciphered text either to ethernet block or h2s block (depends on the dynamic routing). The interface of the functional blocks are exactly the same. It has a memory interface (MEMIF) which can fetch and restore data from and to the memory, an operating system interface (OSIF) which accepts instruction directly from software and can be used to send feed back messages, and an interface to the belonged switch. The feature of OSIF and MEMIF is provided by ReconOS, as is introduced in Chapter 2. From the software side, any new instructions can be sent directly from the OSIF and the very interface could be used by the hardware thread to send feed back messages. For AES encryption engine, the OSIF is used to configure the hardware thread, while the MEMIF is not used. For s2h FB, the memory interface is used to fetch data stored in the 512MB RAM. Each of the functional blocks may contain a packet decoder or a packet encoder which can be used for handling the packet header.

4.1.1 Packet Structure

The packet structure[13] designed by Richard Huber is depicted in Figure 4.2. The data width of the NoC is one byte. A two byte header belongs to each packet and is transmitted prior to the payload. All information needed by the switch to do the routing decision is located in the first header byte. Most importantly, the Global Address and Local Address tells the switch towards which functional block should the packet be sent. The Global Address denotes the address of the switch to which the destination hardware block is connected, while the Local Address distinguishes the destination hardware block within the set of hardware functional blocks connected to the switch[14]. Hence, before the hardware system starts to processing packets, a configuration information which contains the Global and Local Address must be pre-stored to all the related functional blocks.

	0	1	2~5	6~9	
0	Global Addr	Direction	Source IDP	Destination IDP	Payload
1		Latency Critical			
2					
3		Rsvd			
4	Local Addr				
5					
6	Priority				
7					

Figure 4.2: NoC Packet Format

4.1.2 Packet Decoder and Packet Encoder

The packet decoder and packet encoder (detailed in Richard Huber's Master thesis[13]) are the interface between the functional block and its belonging switch. The data width between the function block and the switch is 9 bits, of which the highest bit is used to denote whether the transmitted byte is the beginning of the packet or not. If that bit is set, the byte being transmitted is the first byte of a packet. For the packet decoder, if the switch_data_rdy signal is being set, the data on the switch_data bus is valid. For the packet encoder, if the thread_data_rdy signal being

set, the data on the thread_data bus which is provided by this functional block is valid.

4.2 AES Encryption Engine

The AES encryption engine, as mentioned above, is treated as a complete functional block. In this chapter, it may be also referred as a hardware thread (compared to an implementation in software). The architecture of the encryption engine is depicted in the following Figure 4.3.

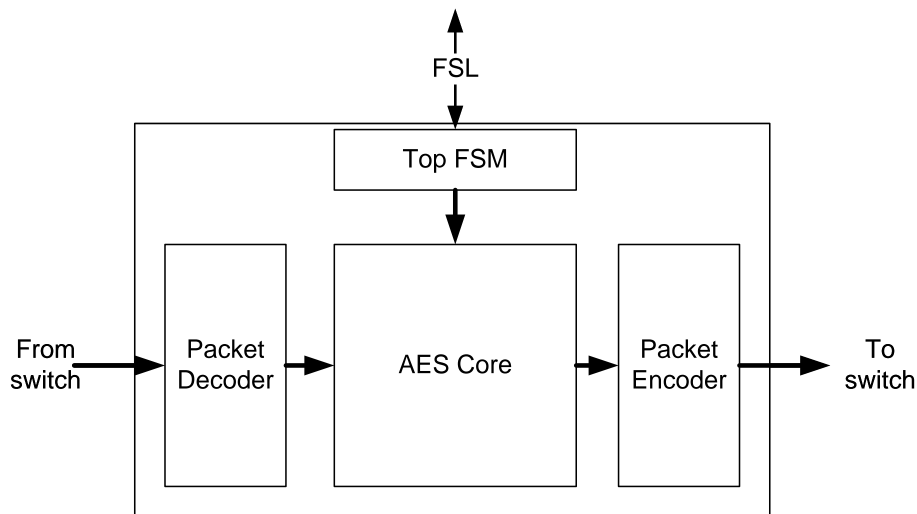


Figure 4.3: Top Module of the Encryption Engine

The encryption engine contains an interface to software, a packet decoder and a packet encoder which are connected to the switch, and an AES core. The AES core consists of a control module, a key expansion module, a key back up module, a read stage and a send stage, and four encryption blocks. Since the task of processing packet headers is left for packet decoder and packet encoder, the AES core is focused on payload encryption only.

4.2.1 AES Hardware Thread Top Module

The major task of the top module is to receive configuration messages from the software. The top module includes an AES Core, a packet decoder, and a packet encoder. It employs the operating system interface (OSIF, also referred to as FSL) provided by ReconOS to achieve that goal. After reset, the AES hardware thread awaits for the configuration information. The operating system first sends the hardware thread with the Global Address and Local Address which specifies the downstream handler of the packets. Two bits which identify the key size being used are also included in this message. Notice that both "10" and "11" are valid for denoting a 256 bit cipher key. Then the software starts to send the cipher key. Because the bus width of the FSL is limited to 32 bits, all 8 words cannot be transmitted at one time. After receiving all 8 words, the hardware threads sends an Ack message. Finally after the completion of the key expansion, AES engine is able to process packets.

When the AES core is encrypting data, the top module can receive new instructions at the same time. Then the new configuration data (cipher key and new

address, if necessary) is submitted to the core. Though the processing packet is not interfered, accepting new packets must be banned. After the key expansion is completed, the encryption pipeline would become active again and starts to process new packets. The state transition of the top module is as depicted in Figure 4.4.

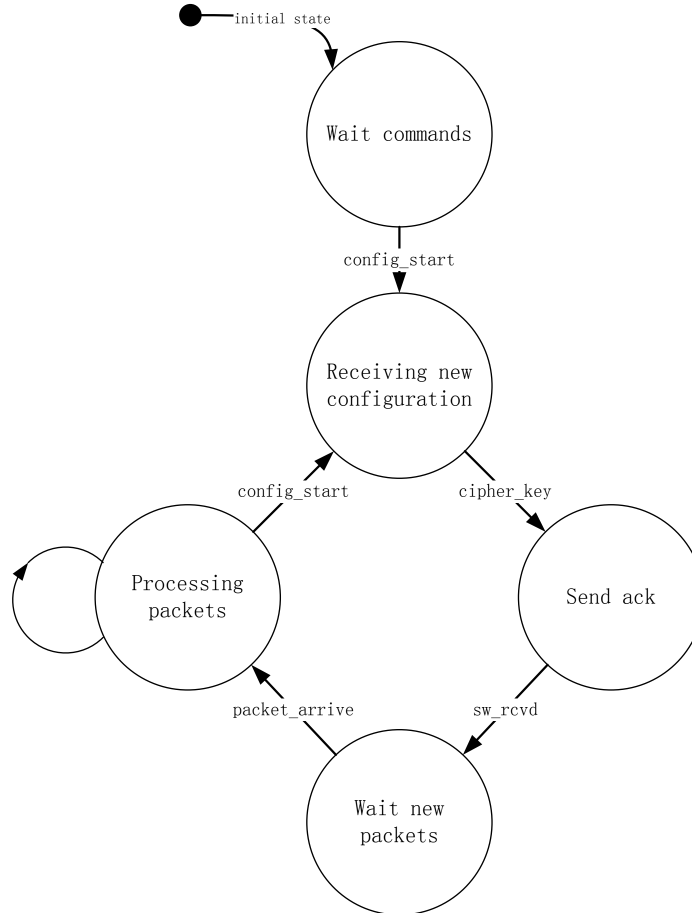


Figure 4.4: State Transition of the Top Module

4.2.2 AES Core

The AES Core, in general contains a key expansion block and an encryption pipeline. The core accepts new configuration data from the top module, expands the cipher key, receives new bytes, concatenates bytes to a 4×4 byte matrix, encrypts the data block (byte matrix), and finally sends the data towards packet encoder. All the tasks are processed in parallel, and a control block monitors the entire process while accepting new configuration. The architecture of the AES core is depicted in Figure 4.5.

Since the system works at 125 MHz with a 9-bit bus (8 used for data transmitting), the highest throughput is fixed to $125 \times 8 = 1Gb/s$. Hence, we aim to design a matching AES Core which works at least 125 MHz. Provided that the bus width is 8 bit, the engine should wait 16 clocks to start block ciphering. Therefore we need it to be pipelined unless we can complete all the ciphering rounds within 16 cycles (128 ns). One solution is to design a super fast architecture and feed the engine with a much higher frequency. Many architectures exist, see [15], [16], [17], [18], and [19].

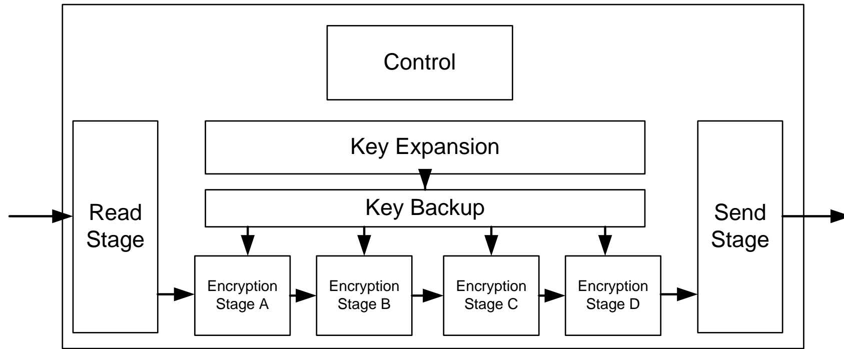


Figure 4.5: Architecture of the AES Core

By deep-pipelining the sbox substitution, some achieved a very high throughput on XILINX FPGA. Still, to avoid becoming the bottleneck in the system, our engine need to be pipelined. Our pipeline is separated into six stages, four for the data encryption and two for receiving and sending data. Because AES-256 requires 14 rounds of transformation, and because the read stage only consumes 16 clock cycles to receive a complete data block, the number of clock cycles that can be employed by a single encryption stage is confined to 16 — if the AES engine is expected to work at line speed. Hence, we split the encryption pipeline into 4 stages so that each stage would consume $4 \times 4 = 16$ clock cycles only. We further merged the shiftRows operation with the sbox substitution so as to save a clock cycle in one round. Because the shiftRows operation involves a routing of transformed bytes from the previous sbox substitution step only, this merging should not introduce additional combinational delays. Thus not interfering the critical path.

Control Module

The control module is connected to all the blocks in the AES Core. It bridges the AES Core to the top FSM in the hardware thread (upper layer). If a new key is received, it commands the key expansion block to generate the round keys. Because the existence of keyBakup module, the key expansion operation does not interfere with the encryption stages. Meanwhile, the control module forbids the read stage to receive new packets. This blocking would continue until all the data in the processing packet has been encrypted and transmitted. The control module detects this idle state in the pipeline and injects the round keys to the keyBakup module. Also, all four stages in the encryption pipeline would be informed of the using key size since it is necessary to determine how many rounds should a data block go through. The send stage would be informed of the global and local address of a new routing (if the software offers a new configuration). Then the receiving of new packets is unblocked at the same time.

Key Expansion

As is illustrated in Chapter 3, the first Nk words (Nk=4, 6, or 8, for AES-128, AES-192, or AES-256 in respect) of the expanded key are filled with the cipher key. Every following word, $w[i]$, is equal to the XOR of $w[i-1]$ and $w[i-Nk]$. For words in positions that are a multiple of Nk, $w[i-1]$ is first cyclic shifted by one byte and an application of a table lookup to all four bytes is followed[2]. Next it's XORed by a round constant, $rcon[i]$. finally the transformed $w[i-1]$ is XORed by the $w[i-Nk]$:

$$w[i] = rcon[i] \oplus sbox(shift(w[i-1])) \oplus w[i-Nk].$$

For example, let the first Nk=4 words of the cipher key for AES-128 be:

$$\begin{pmatrix} 2b & 28 & ab & 09 \\ 7e & ae & f7 & cf \\ 15 & d2 & 15 & 4f \\ 16 & a6 & 88 & 3c \end{pmatrix}$$

Notice that the first word is $w[0] = X$"$16157e2b$", 2b is the lowest byte. To get $w[4]$, we do the following transformation:
(1) shift the word $w[3]$:

$$\begin{pmatrix} 09 \\ cf \\ 4f \\ 3c \end{pmatrix} => \begin{pmatrix} cf \\ 4f \\ 3c \\ 09 \end{pmatrix}$$

(2)sbox substitution of the shifted word:

$$\begin{pmatrix} cf \\ 4f \\ 3c \\ 09 \end{pmatrix} => \begin{pmatrix} 8a \\ 84 \\ eb \\ 01 \end{pmatrix}$$

(3)XOR with $rcon[1]$:

$$\begin{pmatrix} 01 \\ 00 \\ 00 \\ 00 \end{pmatrix} \oplus \begin{pmatrix} 8a \\ 84 \\ eb \\ 01 \end{pmatrix} = \begin{pmatrix} 8b \\ 84 \\ eb \\ 01 \end{pmatrix}$$

(4)XOR with $w[0]$:

$$\begin{pmatrix} 8b \\ 84 \\ eb \\ 01 \end{pmatrix} \oplus \begin{pmatrix} 2b \\ 7e \\ 15 \\ 16 \end{pmatrix} = \begin{pmatrix} a0 \\ fa \\ fe \\ 17 \end{pmatrix}$$

So that we obtain $w[4] = X$"$17fefaa0$". $w[5]$ is obtained by XOR $w[4]$ with $w[1]$. Similarly we get $w[6]$ and $w[7]$. Hence, after the first round of key expansion, we obtain the the following round keys[2]:

$$\begin{pmatrix} 2b & 28 & ab & 09 & a0 & 88 & 23 & 2a \\ 7e & ae & f7 & cf & fa & 54 & a3 & 6c \\ 15 & d2 & 15 & 4f & fe & 2c & 39 & 76 \\ 16 & a6 & 88 & 3c & 17 & b1 & 39 & 05 \end{pmatrix}$$

Do the same thing in the following rounds we obtain the complete round keys. It's worth to mention that the key expansion routine for AES-256 is slightly different than for AES-128 and AES-192. If $Nk = 8$ and $i - 4$ is the multiple of Nk, then a sbox substitution is applied to $w[i-1]$ prior to the XOR.

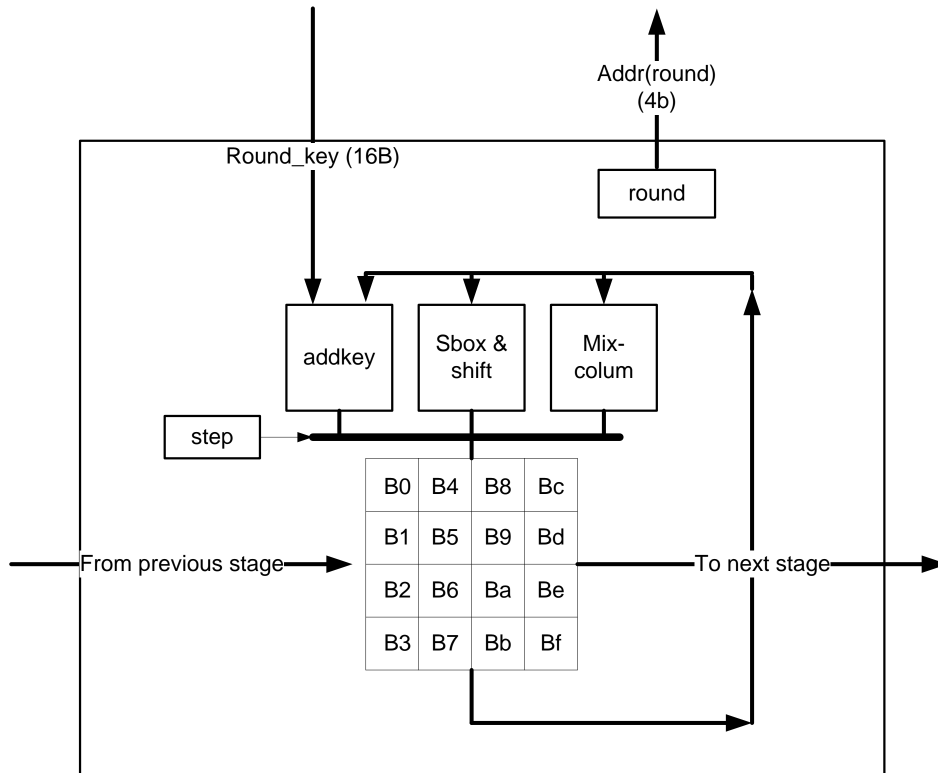


Figure 4.6: Encryption Block

Encryption Stage

The structure of one encryption stage is depicted in Figure 4.6. Every encryption stage is in charge of performing transform on the data matrix for approximately 4 rounds. The architecture of the 4 encryption engines are designed to be same.

Recall in Chapter 3, the mixColumn operation is achieved by:

$$\begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{pmatrix}$$

Since the given matrix only contains X"01", X"02", and X"03". We only have to consider how to achieve the multiplication of one byte with X"02" or X"03" in GF(2^8). A multiplication by X"02" is achieved by left shift the byte by 1 bit. Further, if b_7 of the original byte is 1, XOR the obtained byte by X"1b" so as to reduce the degree of polynomial to 8[2]. X"1b" represents polynomial $x^8 + x^4 + x^3 + x + 1$. The similar operation applies to the multiplication of X"03": data is left shifted by 1 bit and XORed by the initial unshifted value. Then a conditional X"1b" is XORed to reduce the degree of obtained polynomial, if the obtained value is larger than X"FF".

Read Stage and Send Stage

The read stage is the interface between the packet decoder and the encryption pipeline. It receives 8 bits of data every clock cycle. Once the read stage has col-

lected all 128 bits of data as a basic block, it pulls up the data_rdy signal and thus informs the first encryption stage that a 16 bytes of data block is ready. Further, if the byte number of the packet is not a multiple of 16, the read stage is in charge of appending additional X"00"s to the end. The following encryption stages will encrypt the appendix along with the original data. This has caused another problem: the receiving side needs to know how many bytes are there in the transmitting packet so as to discard the appended bytes generated by those additional X"00"s. So another task of the read stage is to count this number and send the information along with the data block through the entire pipeline. Because 1500 is the maximum of bytes in a normal ethernet packet. Two bytes are enough to contain this information. For the decrypting side, it needs only to check the last 4 bits of these two bytes to determine the situation of the final data block. If a complete packet has been passed, the read stage needs to check whether it is permitted to handle the next packet. This step is necessary since the software may want to change the cipher key at any time. If the new key is being sent and the key expansion is not ready, read stage must wait until this process is finished. Then the read_stage_rdy signal is set high again so as to allow incoming packets. The timing of the send stage is similar to the read stage. It's worth to mention that the send stage would transfer the packet length along with the final data block.

4.3 AES Decryption Engine

The AES Core for decryption (named as aesCoreDe in the vhdl package) preserves the most part in the aesCore. The control module, the key expansion module, and the read stage are not modified. Simply change the component name of aesCore to aesCoreDe if the hwt_aes is desired to work as a decryption engine. We still have 4 stages in the pipeline, only the order of transformation in each stage is altered. In order to tackle the corner situation that the final block does not contain 16 bytes of data, it is necessary to modify the pipeline further. To minimize the spoiling of the architecture, we inserted a check block between the read stage and the first encryption stage. This would introduce an increasing of circuit size (as can be seen in Chapter 5). Moreover, the state transition of the send stage has to be modified as well since it will not send the packet length, and the appendix in the last block shall be discarded.

4.4 Software Control Interface

Before sending packets, software has to configure the entire hardware system. First, the routing of the packets must be specified. This is achieved by configuring the Global and Local Address of every related functional blocks on the route. Next, the software needs to send the cipher key to the AES hardware thread while specifying the key size. After receiving the acknowledgement from the hardware thread, it's then safe to sending packets. All the communications are achieved by employing the operating system interface (OSIF) feature provided in ReconOS API. If a packet arrives before the key expansion is finished, it would be blocked. Further, the snd_rdy signal from the send stage remains low. In other words, the traffic hits a solid wall.

4.5 Implementation

The AES encryption engine is written in VHDL and implemented on XILINX VirtexVI ML605 FPGA board. A linux operating system resides on the same board.

The software control program is written in C. The tool flow for generating the hardware components of a ReconOS system starts from a base design prepared for the individual target platform. The base design is given as a standard XILINX EDK project. The completed ReconOS project is processed by XILINX' platgen tool, which generates a top module VHDL description together with VHDL wrappers and a synthesis script for all subcomponents of the system[12]. Finally generating the bit stream for the XILINX ML605 FPGA board. As for the software tool flow, we need to load and boot the Linux kernel before logging into the system. The compilation of the kernel and of the ReconOS application are separated into two independent steps. A detailed "how-to-use" is illustrated in Appendix A.

Chapter 5

Validation and Evaluation

In this chapter, the simulation process and validation platform is introduced. The evaluation results of the encryption engine will be displayed. The simulation is performed in Modelsim. The tests of expanding a 128 bit cipher key, a 192 bit cipher key, and a 256 bit cipher key have been performed, and the results were compared with a set of official test pattern[2]. Next, the generated round keys were used to encrypt a certain data block, and the result is also compared with the official test pattern[2].

After obtaining the right simulation result, we validated the encryption engine in the hardware system on the XILINX ML605 board. We have tested the encryption results by using the 128 bit key, the 192 bit key, and the 256 bit key in respect. Further, the changing of cipher key operation has been performed. We tested different packet lengths to examine corner situations (e.g. packet length is not a multiple of 16).

Finally, an evaluation towards the AES encryption engine has been performed. In order to test the maximum packet sending rate of the encryption engine, we replaced the original s2h (software to hardware) functional block by a packet generator block of which the data_rdy signal is always set. Further, the read_rdy signal of a packet sink which intended to receive the packets is set high forever. The evaluation platform thus, is able to test the maximum packet sending rate while considering the effect of the bus interface.

5.1 Simulation

The first step of functional verification is performed in Modelsim. During the simulation, only the AES Core (aesCore and aesCoreDe) has been tested. We generated the official test vectors[2] and compared the results.

5.1.1 Test of AES-128

The official cipher key is

$$\{000102030405060708090a0b0c0d0e0f\}$$

X"00" is the lowest byte of the lowest word. The plain text is

$$\{00112233445566778899aabbccddeeff\}$$

The generated round keys have 44 words. To be concise, we list the 4 words of the last round key, round key[10] only. See Figure 5.1:

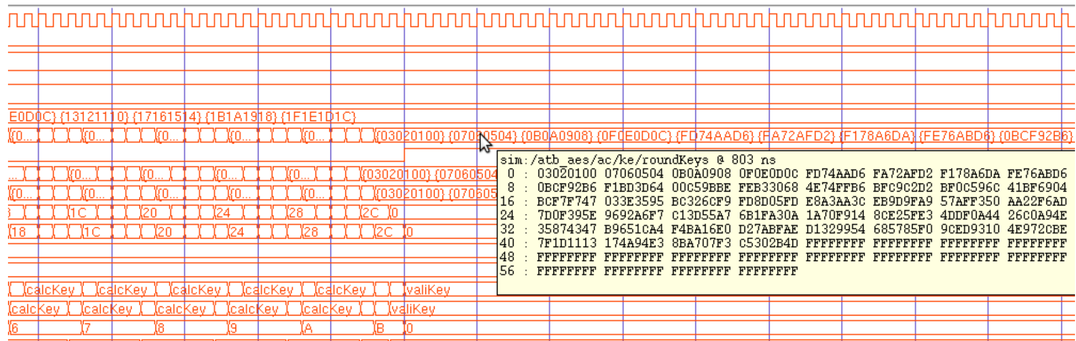


Figure 5.1: AES-128 round key

This is exactly the same round key provided in the official test pattern:

$$\{13111d7fe3944a17f307a78b4d2b30c5\}.$$

The ciphered text should be

$$\{69c4e0d86a7b0430d8cdb78070b4c55a\},$$

compared to our result, which is depicted in Figure 5.2:

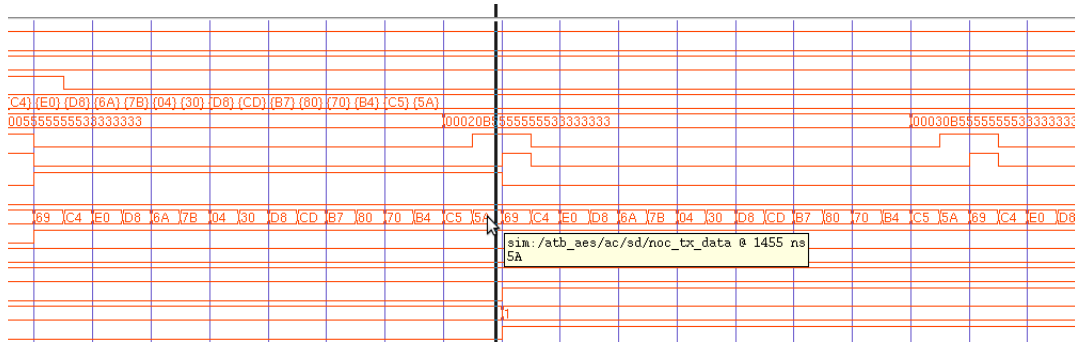


Figure 5.2: AES-128 encryption result

Figure 5.3 depicts the result from the decryption core it is fed with the ciphered text and the same cipher key:

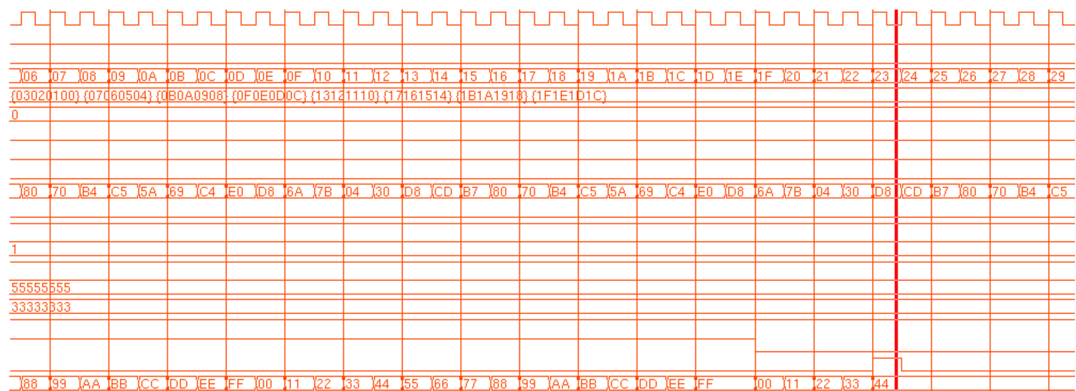


Figure 5.3: AES-128 decryption result

Notice that we sent the decryption engine with X"95" as the last byte denoting there were actually only 5 bytes in the last block. So the final data from the send stage should contain 5 bytes only, from X"00" to X"44". The same applies to the simulation of AES-192 and AES-256.

5.1.2 Test of AES-192

The official plain text remains the same, while there are additional two words appended for the cipher key:

$$\{000102030405060708090a0b0c0d0e0f1011121314151617\}.$$

The last 4 words for the round keys should be:

$$\{a4970a331a78dc09c418c271e3a41d5d\},$$

compared to our result in Figure 5.4:

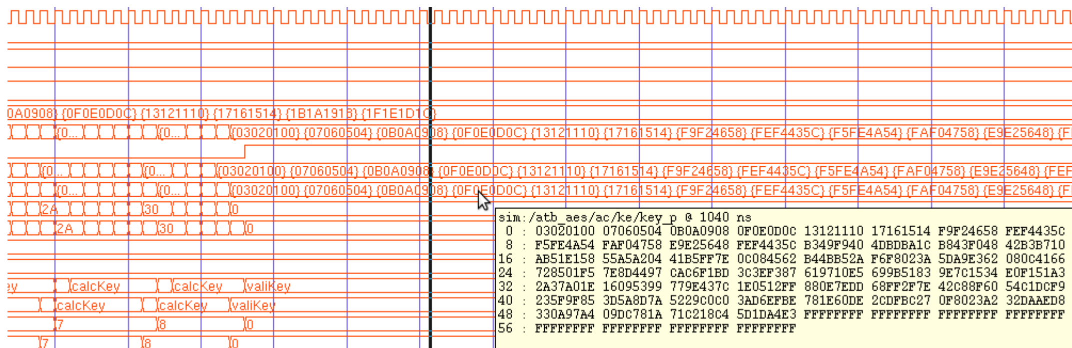


Figure 5.4: AES-192 round key

The ciphered text should be:

$$\{dda97ca4864cdfe06eaf70a0ec0d7191\},$$

compared to the simulation result in Figure 5.5:

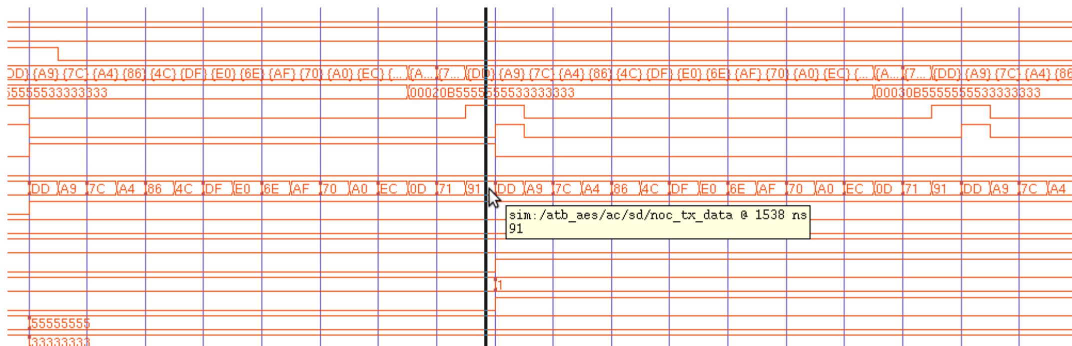


Figure 5.5: AES-192 encryption result

Figure 5.6 depicts the result from the decryption core if it is fed with the ciphered text and the same cipher key:

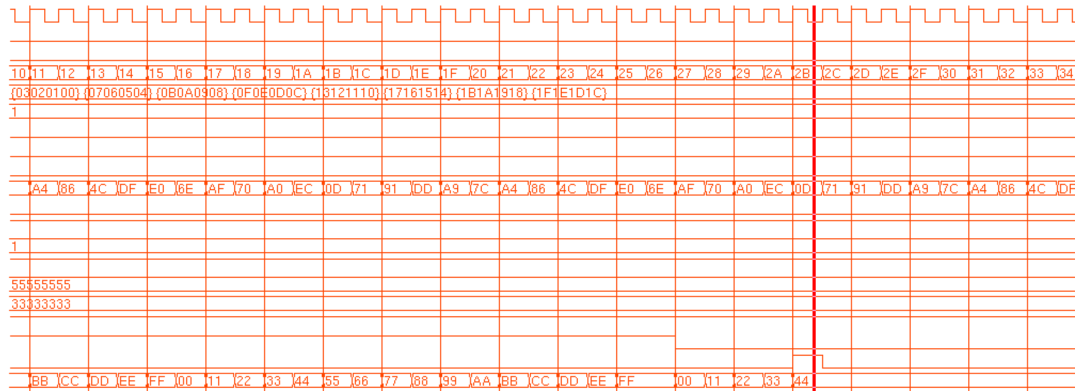


Figure 5.6: AES-192 decryption result

5.1.3 Test of AES-256

The official plain text remains the same, while there are another two words appended for the cipher key:

$$\{000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f\}.$$

The last 4 words for the round keys should be:

$$\{24fc79ccbf0979e9371ac23c6d68de36\},$$

compared to our result depicted in Figure 5.7:

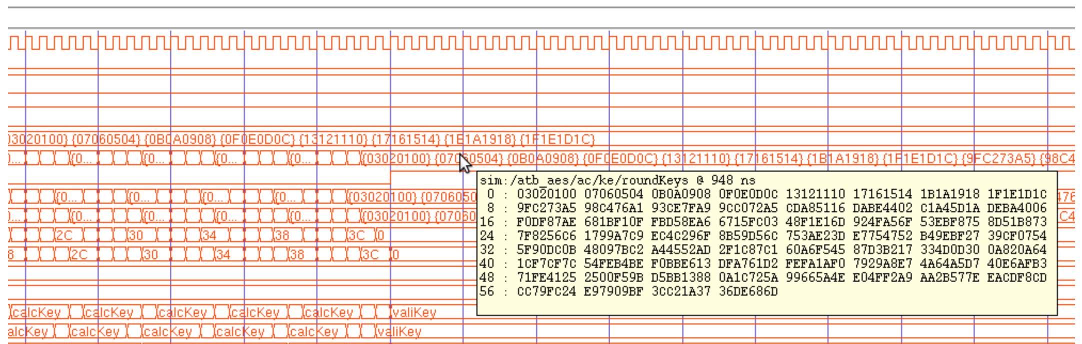


Figure 5.7: AES-256 round key

The ciphered text should be:

$$\{8ea2b7ca516745bfeafc49904b496089\},$$

compared to the simulation result depicted in Figure 5.8:

Figure 5.9 depicts the result from the decryption core if it is fed with the ciphered text and the same cipher key:

5.2 Validation

The validation platform involves a PC and the XILINX ML605 FPGA board. On the PC side, we use ”minicom” to monitor the Linux operating system which resides

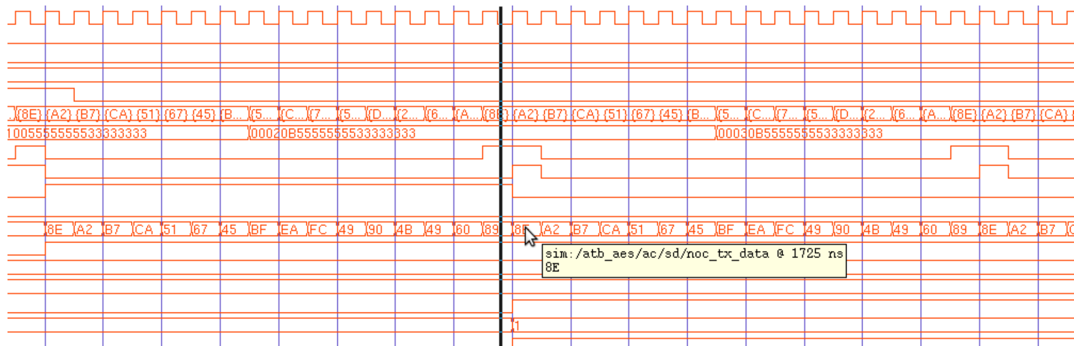


Figure 5.8: AES-256 encryption result

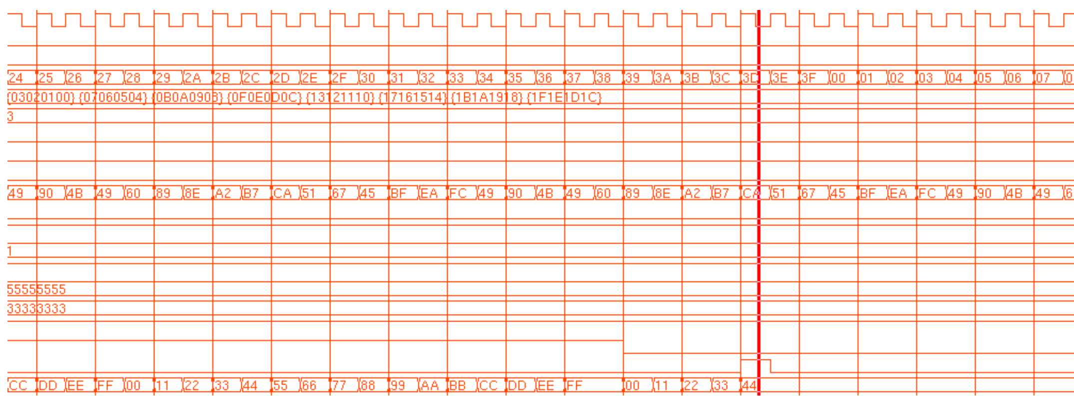


Figure 5.9: AES-256 decryption result

on a 2 GB flash disk on the FPGA board. The operating system tells the s2h (software to hardware functional block) to send packets to the AES encryption engine. The actual data is stored in a 512 MB RAM which locates next to the FPGA circuit. After encryption, the AES engine sends the packet to the ethernet block. Finally, back to the PC side, wireshark is being used to examine the result from the ethernet interface. The validation platform is depicted in Figure 5.10:

Like in the simulation process, encryptions using different key sizes have been tested. Corner situations have been considered. The changing of the key is also tested. We verified the switching of key sizes and fed the AES encryption engine with different packet lengths. The results were as expected. Figure 5.11 shows the validation result of sending a 1000 bytes packet (in plain text) to the ethernet block (left), followed by a packet of which the same payload were encrypted by AES-256 (middle). The final packet travelled through the encryption engine and decryption engine, so the result was expected to be the plain text. We verified the output of the encryption engine, the data were the same as the official test pattern (except for the padding, which was not provided by [2]). The output from the decryption engine was the same as the input of the encryption engine, the plain text itself.

5.3 Evaluation

The evaluation platform involves the software, a packet generator, a packet sink, and the AES encryption engine itself. We replaced the original s2h (software to hardware block) by the packet generator so as to ignore the negative effect of its slow packet sending rate. The packet generator is always ready to send data, while the packet

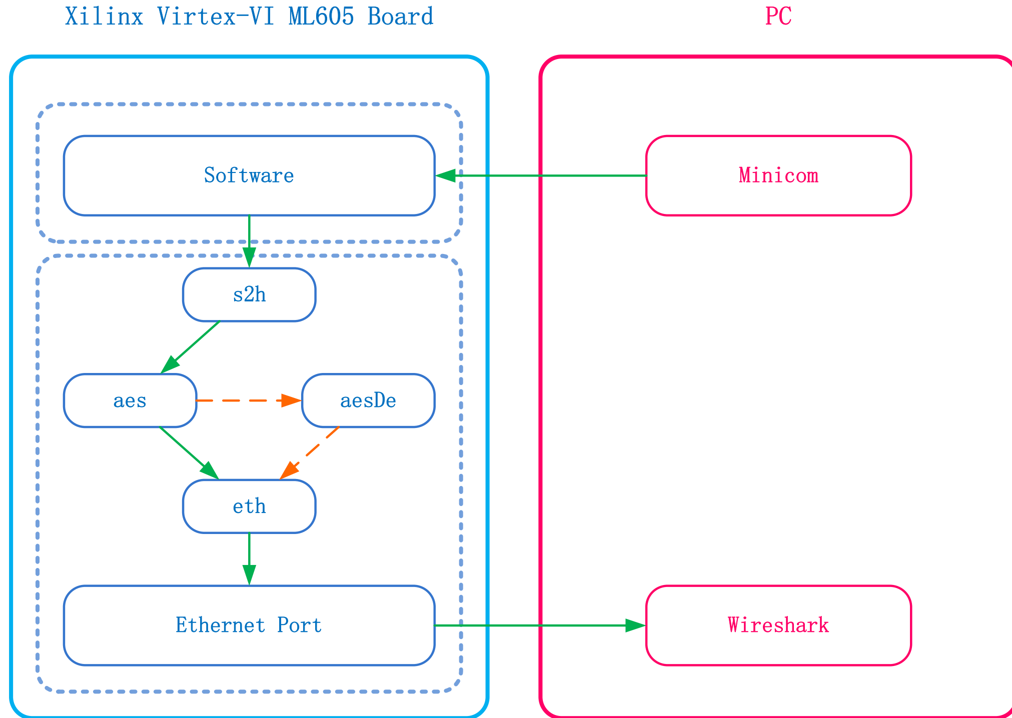


Figure 5.10: Validation Platform

sink can receive data at any time. As the packet sink has received enough packets, it sends a feedback to the software which contains the number of total clock cycles being consumed to transmit the packets. The evaluation platform's architecture is depicted in Figure 5.12.

5.3.1 Synthesis Report

The XILINX ML605 board was specified as the target device. The synthesis data are recorded in Table 5.1.

	Number of Slice Registers	Number of Slice LUTs
Total Available	301440	150720
Encryption Engine	6544	11741
Decryption Engine	6428	14008
System	21197	26461

Table 5.1: Area Consumption

Both the decryption and the encryption engine consume around 2% of the available slice registers. The additional area consumed by the decryption engine stems form the check block, where we used $128 \times 2 = 256$ bits of data. This can be avoided by changing the architecture of the decryption engine in the future work. The majority of the area consumption comes from the pipeline. We further tested the area of one encryption stage is 229 slice registers and 1138 slice LUTs. Most of the LUT consumption comes from the sboxMatrix (640 LUTs). Hence, it's worthwhile to save one stage in the pipeline during future work. For example, we can re-order the encryption process and combine the mixColumn, shiftRows, and addRoundKeys in

one clock cycle. On the other hand, the keyBakup module consumes 2432 slice registers and 2048 LUTs. Should the key replacement happen not too often, it's more economic to give up the parallel processing of key expansion and data encryption so as to save hardware resources.
Moreover, the maximum frequency of both the decryption and encryption engine is 220 MHz. Since the critical path lies in the sbox substitution in the key expansion block, which is not modified in the decryption engine, the result is as expected. Considering that the system frequency is 125 MHz, this provides us with plenty margin to slower the circuit in order to reduce the circuit size.

5.3.2 Speed

We have tested different packet sizes described in RFC 2544[20]. The evaluation results are listed in Figure 5.13. The number in Figure 5.13 denotes the clock cycles being consumed to encrypt and transmit one packet of different sizes described in RFC 2544. Notice that the overhead of a packet is 13 clock cycles if there is no AES encryption engine between the packet generator and the packet sink. Further, the amount of the overhead remains the same for different packet sizes as long as the length of the packet is a multiple of 16. It can be observed that there is a slight drop for the 1518 bytes packet size. This is because 1518 is not a multiple of 16. Recall the architecture description in chapter 4. We intended to append X"00"s in the last data block in order to tackle this corner situation. Once the read stage has received an EOF signal, it then enters the wait_sending state. Since it has to wait the first encryption stage for 16 clock cycles, the time needed for the packet decoder to process the packet header is then, hidden. Thus introduced a drop in the overhead. The obtained maximum packet sending rate for different packet sizes defined in RFC 2544 is showed in Figure 5.14, based on the clock cycles obtained, while the system works at 125 MHz.

The overhead excess stems from two origins: (1) additional routing; (2) packet length bytes. The majority of the additional clock cycles come from the additional routing. Moreover, two additional bytes must be sent to tackle the corner situation, which is another contribution to the overhead. Notice that the red dashed line denotes the packet per second while using AES engine, compare to the blue dashed line without AES.

Figure 5.11: Validation Result

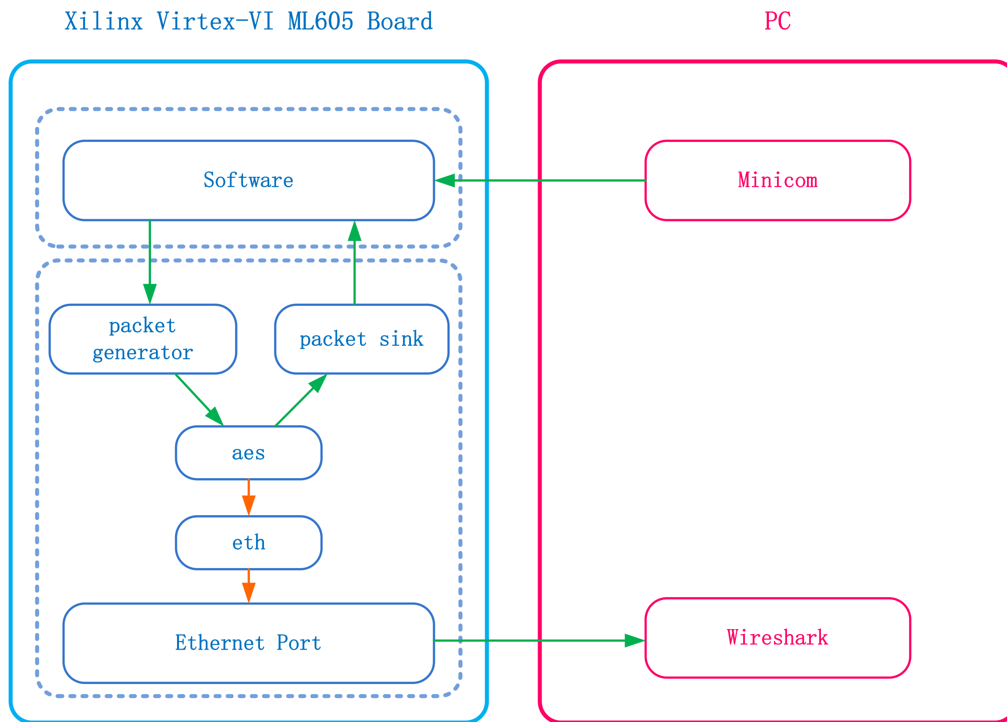


Figure 5.12: Evaluation Architecture

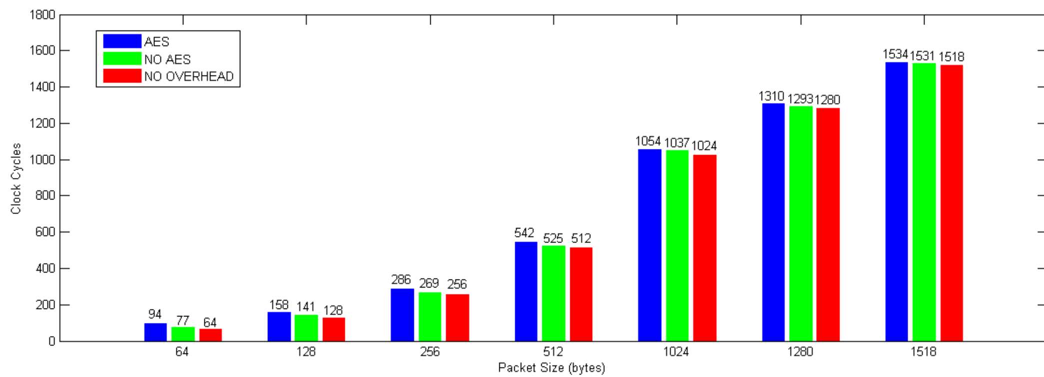


Figure 5.13: Clock Cycles for one Packet

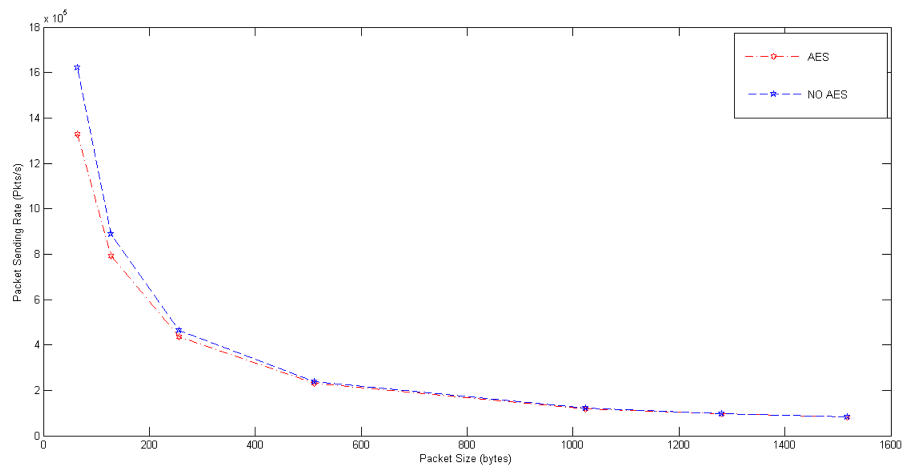


Figure 5.14: Maximum Packet Sending Rate (pps)

Chapter 6

Conclusion and Future Works

At the beginning of the project, we aimed to design and implement an AES encryption engine in hardware which is suitable for the EmbedNet system, whose throughput is 1Gb/s. In the end, both the encryption and decryption engine achieved that line speed. The architecture of the EmbedNet has showed its flexibility and user-friendly design interface. However, there is still plenty room for improvements. Future works may contain, but not confined to the following aspects:

First, if we need a smaller AES circuit, it's desired to save one encryption stage by re-ordering the steps in each encryption round.

On the contrary, if we need an engine of much higher throughput with moderate circuit area, unroll the sbox substitution. By deepening the pipeline, it's promising to achieve a comparable maximum frequency with relatively mild resource consumption. But before that, the width of the bus should be enlarged (we may have to change the test board as well).

As can be observed from the evaluation results, the inserted encryption engine has introduced an additional delay for processing packet headers, as compared to an architecture without AES engine. This is the cost of flexibility. So it's desired to implement a corresponding software encryption engine to see the difference of the two implementations. Should the hardware architecture show little advantage, it's desired to start with the actual bottleneck in the system.

Appendix A

How to Use

A.1 System Configuration

A.1.1 Install XILINX 13.3 tool chain

- Copy DVD to hard-disk.

- Change ownership of /opt:
  chown -hR <username> /opt

- Change the file mode of the install scripts to be executable:
  chmod $a + x$ <tmp dir>/xsetup <tmp dire>/bin/lin64/*

- Execute the install script:
  ./xsetup

- Add the following lines to the .bashrc:
  export XILINX_LICENSE_FILE=8181@lunghin.ee.ethz.ch
  (Notice that the address above might change)
  source /opt/Xilinx/13.3/settings64.sh

- Create the following link:
  ln /usr/bin/make /usr/bin/gmake

- Establish a VPN connection to the ETHZ domain[21].

A.1.2 Install Microblaze GNU Tools

- Clone into the git repository hosted by XILINX (big endian):
  git clone git://git.xilinx.com/xldk/microblaze_v2.0.git

- Unpack the archive:
  tar -xzf microblaze-unknown-linux-gnu.tgz

- Add the following line to the .bashrc:
  export PATH=$ PATH:/<path to git repo>/microblaze_v2.0/microblaze-unknown-linux-gnu/bin

A.2 Hardware System Synthesis

A.2.1 Package

The package used by hwt_aes is yyangPkg.vhd. Include it while using the AES encryption engine. The package contains all the .vhd files necessary to generate the AES Core.

A.2.2 Top Module Connections

Change the FB name in the system.mhs file while necessary. The FBs that have been used for the validation of the AES engine are hwt_s2h, hwt_h2s, hwt_eth, and hwt_aes.

A.2.3 Compilation

Open a console, go to the /hw/edk directory and type "make". Wait for around 30 minutes for the synthesis and the generation of system.bit (if successful, it would appear under /edk/implementation directory).
Download the bitstream to the FPGA board:
dow system.bit

A.3 How to Configure hwt_aes

Assume struct mbox e_mb_put and struct mbox e_mb_get are the message boxes used to communicate between software and hwt_aes.
Further assume:
u32 config_data_start=1;
u32 config_data_mode=16;
u32 config_data_key0=50462976 ; //X"03020100"
u32 config_data_key1=117835012; //X"07060504"
u32 config_data_key2=185207048; //X"0b0a0908"
u32 config_data_key3=252579084; //X"0f0e0d0c"
u32 config_data_key4=319951120; //X"13121110"
u32 config_data_key5=387323156; //X"17161514"
u32 config_data_key6=454695192; //X"1b1a1918"
u32 config_data_key7=522067228; //X"1f1e1d1c"
u32 exit_sig=4294967295;
Notice that only config_data_start= X"00000001" will trigger the encryption engine into the receiving cipher key state. Messages of any other value would be ignored.
Next, the last (lowest) byte of the config_data_mode is used to tell the Global Address and the Local Address, as well as the key size. The last two bits are used to specify the key size:
"00" : $AES-128$,
"01" : $AES-192$,
and both "10" and "11" are valid to denote a 256-bit key size.
The higher 6 bits of the byte tells the routing of the packet. For example, in the validation of the AES engine, "000100" was the address for hwt_eth. Thus a packet encrypted by a 128-bit key would be sent to the ethernet block if and only if the last byte of the config_data_mode= "00010000". For example, 16 is a valid data for config_data_mode.

Hence, whenever user wants to configure or re-configure the AES encryption engine, do the following steps:

- Send the request message to the hardware:
  mbox_put(&e_mb_put, config_data_start);
- Tell the hardware thread the routing and the key size being used:
  mbox_put(&e_mb_put, config_data_mode);
- Send the cipher key.
  mbox_put(&e_mb_put, config_data_key0); mbox_put(&e_mb_put, config_data_key1);
  mbox_put(&e_mb_put, config_data_key2); mbox_put(&e_mb_put, config_data_key3);
  mbox_put(&e_mb_put, config_data_key4); mbox_put(&e_mb_put, config_data_key5);
  mbox_put(&e_mb_put, config_data_key6); mbox_put(&e_mb_put, config_data_key7);
  For the simplicity of usage, send 8 words as cipher key no matter you are using AES-128, AES-192, or AES-256. Notice that for AES-128 and AES-192, only the first 4 or 6 words need to be appropriately set. The actual value of last words does not matter. As long as the last two bits of config_data_mode were right, the corresponding round keys would be generated.
- Wait for the Ack message after hwt_aes has received the configuration message:
  config_rcv=mbox_get(&e_mb_get);
- Make sure other hardware threads (e.g. hwt_s2h, hwt_h2s, and hwt_eth) are configured right.
- Send packets to hardware.
- Compile the software and store the obtained .ko file to the 2GB flash memory together with the Linux system.

A.4 Compile and Download Linux Kernel

- After downloading and extracting the proper Linux kernel[21], copy the device tree file from ReconOS reference design to the Linux source:
  cp reconos_v3/designs/ml605_linux_13.3/noc.dts linux-2.6-xlnx/arch/microblaze/boot/dts/
- Compile the Linux kernel:
  cd linux-2.6-xlnx
  make CROSS_COMPILE=microblaze-unknown-linux-gnu- ARCH=microblaze
  -j8 simpleImage.noc_sysace
- After the XILINX usb cable is installed and the NFS server is appropriately set up[21], download the kernel:
  dow $HOME/linux-2.6-xlnx/arch/microblaze/boot/simpleImage.noc_sysace

A.5 Operation in minicom

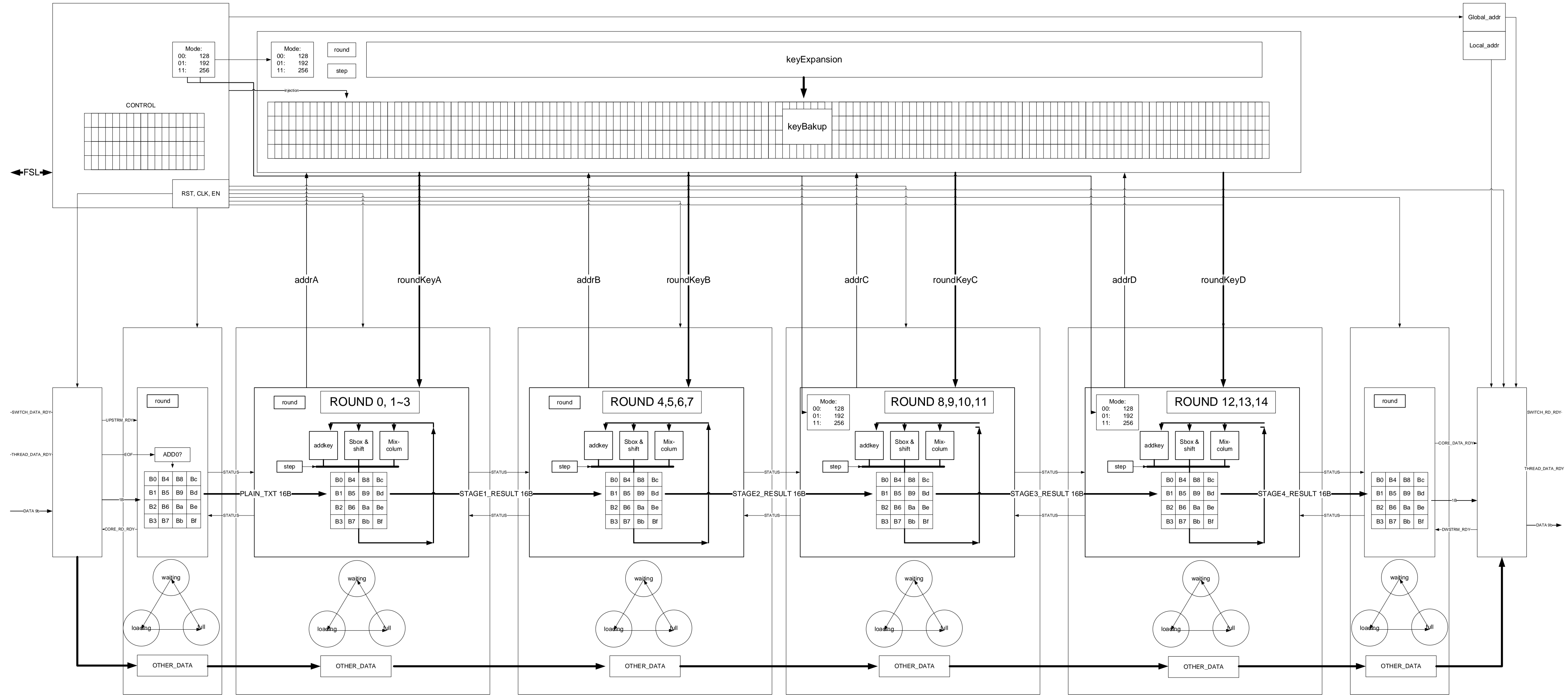
- Start minicom in a new console (root authority may required):
  minicom
  If error message occurs, start it this way:
  minicom -s
  a

Then change to either USB-0 or USB-1 according to your system configuration.

- Set the appropriate data rate. Wait for the kernel download to be finished.
- For functional validation, you may want to start wireshark and listen to the eth0 port before do the following things:
- Type the following command in the minicom window to initialize the system:
  Configure the hardware threads and send packets to hardware by:
  insmod hw_sw_interface.ko (assume this is the file obtained from software compilation)
  Alternatively, use script x.sh provided in the CD-R to replace all the command in this step.
- Observe the results from minicom window, or from wireshark.

Appendix B

AES Engine Architecture



FSL
CONTROL
Mode:
00: 128
01: 192
11: 256
RST, CLK, EN
keyExpansion
round
step
keyBakup
Global_addr
Local_addr
addrA
roundKeyA
addrB
roundKeyB
addrC
roundKeyC
addrD
roundKeyD
SWITCH_DATA_RDY
THREAD_DATA_RDY
DATA 8b
UPSTRM_RDY
EOF
CORE_RD_RDY
ADD0?
ROUND 0, 1~3
ROUND 4,5,6,7
ROUND 8,9,10,11
ROUND 12,13,14
addkey
Sbox & shift
Mix-colum
B0 B4 B8 Bc
B1 B5 B9 Bd
B2 B6 Ba Be
B3 B7 Bb Bf
STATUS
PLAIN_TXT 16B
STAGE1_RESULT 16B
STAGE2_RESULT 16B
STAGE3_RESULT 16B
STAGE4_RESULT 16B
CORE_DATA_RDY
DWSTRM_RDY
SWITCH_RD_RDY
THREAD_DATA_RDY
DATA 8b
waiting
loading
full
OTHER_DATA

Appendix C

Task Description

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Semester Thesis

Encryption for EmbedNet

Yiheng Yang

Advisor: Ariane Keller, ariane.keller@tik.ee.ethz.ch
Co-Advisor: Dr. Stephan Neuhaus, stephan.neuhaus@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

November 2012 - February 2013

1 Introduction

This semester thesis is in the context of the EPiCS project. The goal of the EPiCS project is to lay the foundation for engineering the novel class of proprioceptive computing systems. Proprioceptive computing systems collect and maintain information about their state and progress, which enables self-awareness by reasoning about their behaviour, and self- expression by effectively and autonomously adapting their behaviour to changing conditions. Concepts of self-awareness and self-expression are new to the domains of computing and networking; the successful transfer and development of these concepts will help create future heterogeneous and distributed systems capable of efficiently responding to a multitude of requirements with respect to functionality and flexibility, performance, resource usage and costs, reliability and safety, and security.

In this thesis we focus on the networking aspect of EPiCS which we call *EmbedNet*. EmbedNet uses the concepts of the network architecture developed in the ANA project as a basis. The ANA network architecture is a novel architecture that enables flexible, dynamic, and fully autonomous formation of network nodes. With EmbedNet we develop the ANA architecture further. On the one hand we develop mechanisms to adapt the functionality provided by the protocol stack at runtime, on the other hand we develop mechanisms that map the networking functionality dynamically to either hardware or software.

The objective of this Semester Thesis is to implement a functional block in hardware that encrypts packets using the AES algorithm.

2 Assignment

This assignment aims to outline the work to be conducted during this thesis. The assignment may need to be adapted over the course of the project.

2.1 Objectives

The goal of this thesis is to implement an encryption functional block that can be used in EmbedNet. It should use the AES algorithm. The block should be executed as a functional block in hardware. The key used for the AES algorithm can be stored directly in the hardware module. If the key needs to be changed, the module has to be recompiled and then the original module can be replaced with the new module.

2.2 Tasks

This section gives a brief overview of the tasks the student is expected to perform towards achieving the objective outlined above. The binding project plan will be derived over the course of the first three weeks depending on the knowledge and skills the student brings into the project.

2.2.1 Familiarization

- Study the available literature on ANA, EmbedNet and Reconos [1, 2, 3, 4, 6, 7, 8]. While it is important to understand the general ideas and concepts, not every detail needs to be understood.
- Setup an FPGA development environment with the Xilinx tools 13.3 (exact) and ModelSim SE version 6.1f (or higher).
- Familiarize yourself with git/github and clone the reconos source code repository [5].
- Download the xilinx Linux kernel and the corresponding cross compilation toolchain.
- Verify your toolchain by running the *protocol_graph* demo.
- In collaboration with the advisor, derive a project plan for your semester thesis. Allow time for the design, implementation, evaluation, and documentation of your code.

2.2.2 Architecture and hardware design

- Design the encryption engine of your hardware thread. You might have a look at the encryption cores available at [9].
- Optional: Design an interface where you can set the key by a software program.
- Optional: Design a software AES functional block.

2.2.3 Implementation

- Determine an appropriate version control system. You might consider to use github.
- Implement a dummy hardware thread that forwards all packets.
- Implement the encryption engine in your hardware thread.
- Optional: Implement the interface to set a key by a software program.
- Optional: Implement the software AES functional block.

2.2.4 Validation

- Validate the correct operation of your implementation after each implementation step. Take corner cases into account. Use for your evaluation different packet sizes (short, long, even or odd number of bytes, etc.).
- Check the resilience of the implementation, including its configuration interface, to uneducated users.
- Optional: Validate also the software module.

2.2.5 Evaluation

- Do a performance evaluation of your implementation.
- Determine the maximum packet sending rate in HW for the packet sizes described in RFC 2544.
- Determine the time it takes to process a packet in HW for the packet sizes described in RFC 2544.
- Do the same evaluations with your software module.

2.2.6 Documentation

- Appropriate source code documentation.
- Write a step-by-step how to that describes the compilation of your code, the loading of the code into the hardware and the execution of your code.
- Write a documentation about the design, implementation, validation and evaluation of your work.

3 Milestones

- Provide a "project plan" which identifies the mile stones.
- One intermediate presentations: Give a presentation of 10 minutes to the professor and the advisors. In this presentation, the student presents major aspects of the ongoing work including results, obstacles, and remaining work.

- Final presentation of 15 minutes in the CSG group meeting, or, alternatively, via teleconference. The presentation should carefully introduce the setting and fundamental assumptions of the project. The main part should focus on the major results and conclusions from the work.
- Any software that is produced in the context of this thesis and its documentation needs to be delivered before conclusion of the thesis. This includes all source code and documentation. The source files for the final report and all data, scripts and tools developed to generate the figures of the report must be included. Preferred format for delivery is a CD-R.
- Final report. The final report must contain a summary, the assignment, the time schedule and the Declaration of Originality. Its structure should include the following sections: Introduction, Background/Related Work, Design/Methodology, Validation/Evaluation, Conclusion, and Future work. Related work must be referenced appropriately.

4 Organization

- Student and advisor hold a weekly meeting to discuss progress of work and next steps. The student should not hesitate to contact the advisor at any time. The common goal of the advisor and the student is to maximize the outcome of the project.
- The student is encouraged to write all reports in English; German is accepted as well.
- The core source code will be published under the GNU general public license.

5 References

[1] ReconOS: Multithreaded Programming for Reconfigurable Computers: Description of the hardware/software architecture
[2] Reconifgurable Nodes for Future Networks: Description of how we would like to use the hw/sw architecture to build reconfigurable networks
[3] The Autonomic Network Architecture (ANA): Description of the ideas and sw prototype for configurable networks
[4] Deliverable D3-1: Architecture And Tool Flow, Deliverable D3.2: Operating System Layer For Autonomous Compute Node
[5] https://github.com/EPiCS/reconos/
[6] Deliverable D4.1: Flexible Protocol Graph Architecture
[7] Hardware Support for Dynamic Protocol Stacks
[8] Autonomic Protocol Graph Architecture for Pervasive Computing
[9] http://opencores.org Webpages:
http://www.ana-project.org
http://www.epics-project.eu
http://www.reconos.de

Appendix D

Declaration of Originality

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of Originality

This sheet must be signed and enclosed with every piece of written work submitted at ETH.

I hereby declare that the written work I have submitted entitled

____

is original work which I alone have authored and which is written in my own words.*

Author(s)

Last name	First name

Supervising lecturer

Last name	First name

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (http://www.ethz.ch/students/exams/plagiarism_s_en.pdf). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Place and date	Signature

*Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Print form

Appendix E

Time Line

Hardware Encryption for Embedded Systems

Time Line

Date (Week - Date):

1	2	3	4	5	5	6	7	8	9	10	11	12	13	14
Nov.29	Dec.6	Dec.13	Dec.20	*Christmas*	Jan.3	Jan.10	Jan.17	Jan.24	Jan.31	Feb.7	Feb.14	Feb.21	Feb.28	March.7

Project Plan	Weeks													
Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Familiarization														
1. EmbedNet, ReconOS														
2. FPGA environment, Xilinx Tool chain, Modelsim SE														
3. Xilinx Linux Kernel and cross compilation tool-chain														
Architecture														
1. Arch0 (calculating round keys, read bytes, processing data, send bytes, frequency: 125MHz)														
2. Arch1 (FSL, additional task: set key from sw and get status from hw)														
3. (build a decryption engine)														
Implementation														
Validation														
1. Basic functions (generate round keys, receiving, processing, sending data, examine results from wireshark)														
2. Set key from software														
Evaluation														
1. Determine the overhead for different packet sizes defined in RFC2544														
2. Determine the max packet processing rate (packets per second)														
Documentation														
1. Write a step-by-step how to that describes the compilation of code, loading and execution of code														
2. Write a documentation about the design, source code documentation														
Presentation														

References

[1] Ghazi Bouabene, Christophe Jelger, Christian Tschudin, Stefan Schmid, Ariane Keller, and Martin May, "The Autonomic Network Architecture (ANA)," *IEEE Journal on Selected Areas in Communications*, vol. 28, Jan 2010.

[2] "Advanced encryption standard (aes)," 2001. National Institute of Standards and Technology (NIST), Federal Information Processing Standards Publication 197.

[3] "Semester project task description." see Appendix C.

[4] A. Keller, D. Borkmann, and W. Mühlbauer, "Efficient implementation of dynamic protocol stacks," in *ANCS*, pp. 83–84, 2011.

[5] A. Keller, B. Plattner, E. Lübbers, M. Platzner, and C. Plessl, "Reconfigurable nodes for future networks," in *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pp. 357 –361, dec. 2010.

[6] A. Keller, D. Borkmann, and S. Neuhaus, "Hardware support for dynamic protocol stacks," in *ANCS*, pp. 75–76, 2012.

[7] E. Lübbers and M. Platzner, "Reconos: Multithreaded programming for reconfigurable computers," *ACM Trans. Embed. Comput. Syst.*, vol. 9, pp. 8:1–8:33, Oct. 2009.

[8] "www.reconos.de." last accessed 2013-2-23.

[9] E. Lübbers, M. Platzner, C. Plessl, A. Keller, and B. Plattner, "Towards Adaptive Networking for Embedded Devices based on Reconfigurable Hardware," in *Engineering of Reconfigurable Systems and Algorithms*, pp. 225–231, 2010.

[10] J. Daemen, J. Daemen, J. Daemen, V. Rijmen, and V. Rijmen, "Aes proposal: Rijndael," 1998.

[11] A. Hodjat and I. Verbauwhede, "A 21.54 gbits/s fully pipelined aes processor on fpga," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pp. 308 – 309, april 2004.

[12] "Autonomous heterogeneous multi-core architecture and basic tool flow," 2011. Engineering Proprioception in Computing Systems (EPiCS).

[13] R. Huber, "A dynamic hardware architecture for future networks," Master's thesis, ETHZ, 2011.

[14] "Flexible protocol graph architecture," 2012. Engineering Proprioception in Computing Systems (EPiCS).

[15] M. McLoone and J. McCanny, "Single-chip fpga implementation of the advanced encryption standard algorithm," in *Field-Programmable Logic and Applications* (G. Brebner and R. Woods, eds.), vol. 2147 of *Lecture Notes in Computer Science*, pp. 152–161, Springer Berlin Heidelberg, 2001.

[16] A. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An fpga-based performance evaluation of the aes block cipher candidate algorithm finalists," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 9, pp. 545 –557, aug. 2001.

[17] K. Gaj and P. Chodowiec, "Fast implementation and fair comparison of the final candidates for advanced encryption standard using field programmable gate arrays," in *Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA*, CT-RSA 2001, (London, UK, UK), pp. 84–99, Springer-Verlag, 2001.

[18] H. Qin, T. Sasao, and Y. Iguchi, "An fpga design of aes encryption circuit with 128-bit keys," in *Proceedings of the 15th ACM Great Lakes symposium on VLSI*, GLSVLSI '05, (New York, NY, USA), pp. 147–151, ACM, 2005.

[19] T. Good and M. Benaissa, "Aes on fpga from the fastest to the smallest," in *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop*, pp. 427–440, Springer, 2005.

[20] "http://www.ietf.org/rfc/rfc2544.txt." last accessed 2013-2-24.

[21] "https://github.com/rihuber/ma/wiki/_pages." last accessed 2013-2-24.