Hardware Implementation of the Advanced Encryption Standard

by

Jennifer Maurer

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Masters of Engineering in Electrical Engineering

at the Massachusetts Institute of Technology

January 17, 2003

Copyright 2003 Jennifer R. Maurer. All rights reserved.

The author hearby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author

Department of Electrical Engineering
January 17, 2003

Certified by

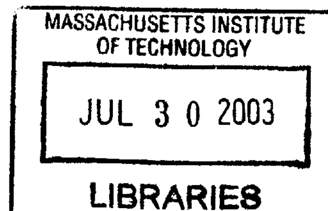
Jonathan H. Raymond
VI-A Company Thesis Supivisor

Certified by

Donald E. Troxel
M.I.T. Thesis Supivisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Theses


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 3 0 2003
LIBRARIES

BARKER

Hardware Implementation of the Advanced Encryption Standard

by
Jennifer R. Maurer

Submitted to the
Department of Electrical Engineering

January 17, 2003

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering

Abstract

This project implements a hardware solution to the Advanced Encryption Standard (AES) algorithm and interfaces to IBM's CoreConnect Bus Architecture. The project is IBM SoftCore compliant, is synthesized to the .18 micron CMOS double-well technology, runs at 133 MHz, and is approximately 706K for the 16x128 bit buffer implementation and 874K gates for the 32x128 bit buffer implementation. Data can be encrypted and decrypted at a throughput of 1Gbps. The work described in the paper was completed as a part of MIT's VI-A program in the ASIC Digital Cores III group of the Microelectronics Division at IBM.

VI-A Company Thesis Supervisor: Jonathan H. Raymond
M.I.T. Thesis Supervisor: Donald E. Troxel

Acknowledgements

This thesis would not have been possible without the guidance and direction of several people. First I would like to thank all of the people at IBM who helped me. My mentor, Jon Raymond, was involved with every phase of my project and provided me with direction. He was always willing to help me out no matter how busy he was. My managers, Dave Sobczak and Bob Fiorenza, were very supportive of my thesis and made every effort to make my time at IBM enjoyable. Andy Anderson was always available to give me advice and helped to edit my thesis. I would also like to thank my thesis advisor at MIT, Don Troxel, for reading this thesis and for being available whenever problems arose.

There are many other people who helped to keep me sane. Without the love and support of my fiance, Jeremy Lilley, I probably would have been tempted to give up. He provided me with many moments of encouragement and stress relief when I needed it most. I would like to thank my parents, Bob and Linda Maurer, for all of the little things they did for me. They were always there to listen to me and give me encouragement. My sister, Michelle Maurer, always managed to tell me an amusing story every time we talked and gave me many reasons to laugh. Caroline Hon was always there to talk on the phone, even if she was too far away to go out for buffalo wings. I would like to thank Rajul Shah for all of the time we spent together. I don't think I ever have, nor ever will, eat as much garlic and chilli powder as we did while working on our theses.

Table of Contents

1 Introduction....................10

2 Encryption Algorithm....................12

2.1 Encryption Overview....................12

2.2 Encryption Algorithm Selection....................12

2.3 Advanced Encryption Standard Algorithm....................13

2.3.1 Notation....................14

2.3.2 Mathematics....................14

2.3.3 Transformations for Encryption....................17

2.3.4 Transformations for Decryption....................20

2.3.5 Key Expansions....................22

3 IBM Core Connect Bus Architecture....................25

3.1 Core Connect Bus Architecture Overview....................25

3.2 Device Control Register Bus....................27

3.3 Processor Local Bus....................28

4 Encryption Algorithm Implementation....................29

4.1 Key Setup....................37

4.2 Encryption....................39

4.2.1 SubByte Implementation....................41

4.2.2 ShiftRows Implementation....................43

4.2.3 MixColumns Implementation....................44

4.2.4 Key Schedule Implementation....................46

4.2.5 AddRoundKey Implementation....................49

4.3 Decryption....................51

4.3.1 InvShiftRows Implementation....................53

4.3.2 InvSubBytes Implementation....................53

4.3.3 Inverse Key Schedule Implementation....................55

4.2.4 AddRoundKey Implementation....................58

4.2.5 InvMixColumns Implementation....................58

4.4 Implementation Flexibility 61
5 IBM Core Connect Bus Interface Implementation 62
5.1 Device Control Register Bus 63
5.2 Processor Local Bus 68
6 Verification 79
7 Synthesis 82
8 Timing 84
9 Future Work 88
Appendix A: Sample Key Expansion 90
Appendix B: Sample SubBytes Transformation 94
Appendix C: Sample Encryption 95
Appendix D: Sample Decryption 96
References 97

List of Figures

Figure 1.1:Top Level Block Diagram........ 10
Figure 2.1:128 Bit State Representation........ 14
Figure 2.2:Extended Euclidean Algorithm........ 16
Figure 2.3:AES Encryption Algorithm........ 17
Figure 2.4:Shift Rows Transformation........ 18
Figure 2.5:AES Decryption Algorithm........ 20
Figure 2.6:InvShiftRows Transformation........ 21
Figure 2.7:Algorithm for Key Expansion........ 23
Figure 3.1:Core Connect Diagram Based on System-On-a-Chip........ 25
Figure 3.2:DCR Block Diagram........ 27
Figure 3.3:PLB Block Diagram........ 28
Figure 4.1:AES Algorithm Block Diagram........ 29
Figure 4.2:AES Algorithm Flow Chart........ 31
Figure 4.3:Round Signal Diagram........ 32
Figure 4.4:Round 0 Block Diagram........ 33
Figure 4.5:Rounds 1-9 Block Diagram........ 34
Figure 4.6:Rounds 10-13 Block Diagram........ 35
Figure 4.7:Round 14 Block Diagram........ 36
Figure 4.8:GetDecKey Block Diagram........ 38
Figure 4.9:GetDecKey Timing Diagram........ 39
Figure 4.10:Encryption Round Timing Diagram........ 40
Figure 4.11:SubBytes Signal Diagram........ 41
Figure 4.12:SubBytes Timing Diagram........ 43
Figure 4.13:MixColumns Signal Diagram........ 44
Figure 4.14:MixColumns Timing Diagram........ 46
Figure 4.15:KeyScheduler Signal Diagram........ 47
Figure 4.16:KeyScheduler Timing Diagram........ 48
Figure 4.17:AddRoundKey Signal Diagram........ 49

Figure 4.18:AddRoundKey Timing Diagram.. 50
Figure 4.19:Decryption Round Timing Diagram..52
Figure 4.20:InvSubBytes Signal Diagram.. 53
Figure 4.21:InvSubByte Timing Diagram...55
Figure 4.22:InvKeyScheduler Signal Diagram... 56
Figure 4.23:KeyScheduler Timing Diagram...57
Figure 4.24:InvMixColumns Signal Diagram...58
Figure 4.25:InvMixColumns Timing Diagram... 60
Figure 5.1:Top Level Signal Diagram..62
Figure 5.2:DCR Interface Signal Diagram... 63
Figure 5.3:CPU Operation Flow Chart.. 66
Figure 5.4:Timing Diagram for Starting and Ending a Transation........................... 66
Figure 5.5:Fill Buffer Flow Chart.. 69
Figure 5.6:Send Data Flow Chart...70
Figure 5.7:Empty Buffers Flow Chart...70
Figure 5.8:PLB Interface Timing Diagram...72
Figure 5.9:PLB Interface Timing Diagram...73
Figure 5.10:PLB Interface Timing Diagram...74
Figure 5.11:PLB Interface Timing Diagram...75
Figure 5.12:PLB Interface Timing Diagram...76
Figure 5.13:PLB Interface Timing Diagram...77
Figure 5.14:PLB Interface Timing Diagram...78
Figure 6.1:Toolkit Block Diagram..79

List of Tables

Table 2.1:SubBytes Transformation Lookup Table.. 19

Table 2.2:InvSubBytes Transformation Lookup Table... 21

Table 2.3:Rcon Values..23

Table 4.1:AES Algorithm Signal Descriptions...30

Table 4.2:Round Signal Descriptions..33

Table 4.3:GetDecKey Signal Descriptions...38

Table 4.4:SubBytes Signal Descriptions..42

Table 4.5:MixColumns Signal Descriptions...45

Table 4.6:KeyScheduler Signal Descriptions..47

Table 4.7:AddRoundKey Signal Descriptions..49

Table 4.8:InvSubBytes Signal Descriptions...54

Table 4.9:InvKeyScheduler Signal Descriptions...56

Table 4.10:InvMixColumns Signal Descriptions...59

Table 5.1:DCR Bus Register Contents...64

Table 5.2:Error Types..67

Table 6.1:Verification Tests..81

Table 7.1:Synthesis Results..83

Table 8.1:Required Arrival Times for Inputs..85

Table 8.2:Required Arrival Times for Outputs..86

Table A.1:Key Expansion of a 128-bit Key...90

Table A.2:Key Expansion of a 192-bit Key...91

Table A.3:Key Expansion of a 256-bit Key...92

Table B.1:Extended Euclidean Algorithm..94

Table C.1:Encryption Example...95

Table D.1:Decryption Example...96

1 Introduction

As processor speeds become faster, methods used to implement data security become more important. Until recently, the Data Encryption Standard (DES) was enough for most purposes. However, processor speeds are now fast enough that the algorithm can be broken by trying every possible key.

In January of 1997 the National Institute of Standards and Technology (NIST) announced that they were going to begin an effort to find a new, more secure, algorithm to replace the DES. After many tests and careful evaluation by the encryption community, the Rijndael encryption algorithm was officially approved for the Advanced Encryption Standard (AES) in December of 2001. This paper will discuss one implementation of the AES algorithm, which will be referred to as the AES Encryption Core.

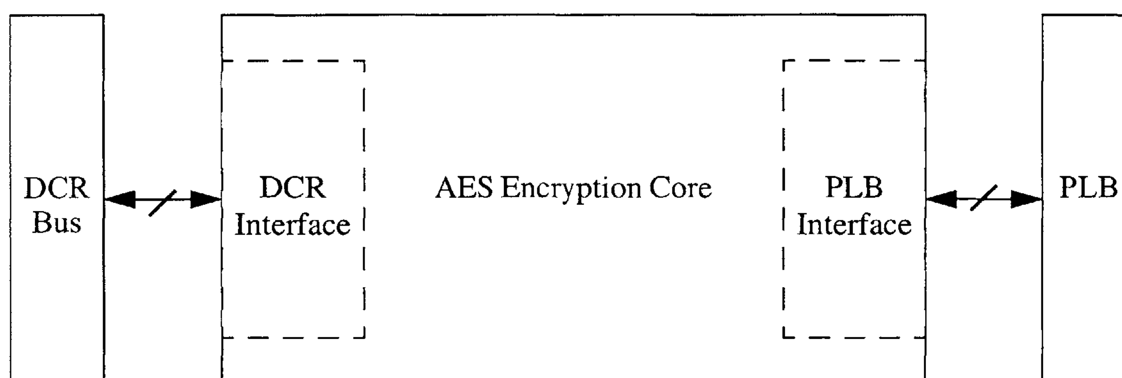


Figure 1.1 Top Level Block Diagram. The AES Encryption Core contains interfaces to the Processor Local Bus (PLB) and the Device Control Register (DCR) Bus.

The AES Encryption Core is a soft core[1] compliant with IBM's Softcore methodology and is capable of encrypting and decrypting data at a throughput of 1Gbps using a 133 MHz clock in 0.18 micron double-well CMOS (SA27E) technology. The architecture minimizes the area while meeting the 1Gbps throughput. This project was completed through architecture design, verification, synthesis, and static timing.

1. A soft core is supplied to a customer as VHDL or Verilog netlists and is verified to meet test and timing requirements.

The AES Encryption Core is implemented modularly, interfacing with the IBM Core Connect Bus Architecture. This allows the Core Connect Interface to be easily removed and replaced with another interface. The Encryption Core interfaces with the Core Connect Bus Architecture through the Device Control Register (DCR) Bus and the Processor Local Bus (PLB). The PLB is a high performance bus used to access memory, while the DCR Bus is used for configuration purposes. In the case of the AES Encryption Core, the DCR Bus is used to configure an encryption or decryption transaction. Given this configuration information, the core first reads data from memory using the PLB, then processes the data, and finally writes the data back to memory using the PLB. The interfaces are shown in figure 1.1.

2 Encryption Algorithm

2.1 Encryption Overview

Encryption is a way to keep data secure by using mathematical transformations on a sequence of bits. These transformations use a set sequence of bits known as a key. There are two well known types of encryption: public key/private key pairs and symmetric keys.

The advantage of public key/private key pairs is that they are more secure because anyone can use the public key to encrypt data, but only the private key owner can decrypt the data. This keeps the private key uncompromised. The problem is that the algorithms require most public key/private key pairs to have a large number of bits (usually of at least 1000 bits) to keep the private key secure, making the encryption or decryption slow. Because this type of encryption is slow, public key/private key pairs are most often used to transfer keys over an insecure line or are used for authentication. It is not used for encrypting or decrypting large amounts of data. The symmetric keys are changed often, so that a compromise of a single key provides access to a limited amount of data.

Symmetric key encryption is secure given that the key is securely distributed. Most algorithms use anywhere from 56 to 256 bit keys, and can be much faster than the public key/private key encryption. A single key is used for both encryption and decryption and must be kept secret. Symmetric keys are well suited for encrypting large amounts of data.

2.2 Encryption Algorithm Selection

The Advanced Encryption Standard (AES) algorithm was selected for several reasons. The Encryption Core will need to support a wide range of applications and will be used to encrypt large amounts of data. The core will go into a library that other designers can use as a black box design. Using a standard algorithm instead of a non-

standard algorithm may allow more designs to use the Encryption Core. In this case, we assume that data that is encrypted will be decrypted at some point in time. In some applications, data gets encrypted and decrypted on different blocks. If so, both blocks need to use the same, ideally standard algorithm.

Two standard algorithms will be compared: the Advanced Encryption Standard (AES) and the Data Encryption Standard (DES). Both of these are symmetric key algorithms. The DES algorithm supports a key length of 56 bits. The DES is older and has a much smaller key space to exhaust. The AES algorithm supports key lengths of 128, 192, and 256 bits. Given that a block of encrypted data and a block of decrypted is known, if the 56-bit DES algorithm could be broken in 1 second simply by trying every single key, the same method using the 128-bit AES algorithm will take approximately 1.5×10^{14} years to break; the 192-bit AES algorithm, 2.8×10^{33} years; the 256-bit AES algorithm, 5.1×10^{52} years. It is easy to see that an algorithm with more bits has a much greater impact on the security.[2]

2.3 Advanced Encryption Standard Algorithm

The AES algorithm is based on simple mathematical transformations whose inverses are difficult to compute without the key. The algorithm has 4 basic transformations that are repeated 10, 12, or 14 times, depending on what key size is being used. Repeating the transformations multiple times helps to ensure that breaking the algorithm will be more difficult to compute than trying every single key. Currently it is believed that no simplification of the transformations will allow a shortcut to break the AES algorithm. This belief is held because the transforms are simple and allow thorough analysis.[6]

---

2. For a detailed discussion on security of the AES algorithm, see AES Proposal: Rijndael.

All four transformations are applied to the 128-bit state, represented in Figure 2.1 where each square represents one byte. Transformations on the state can be applied to each individual byte, the columns, or the rows.

S_0	S_4	S_8	S_{12}
S_1	S_5	S_9	S_{13}
S_2	S_6	S_{10}	S_{14}
S_3	S_7	S_{11}	S_{15}

Figure 2.1 128 Bit State Representation. Each square represents one byte of the state.

2.3.1 Notation

{xx} is the representation of a byte in hexadecimal.

$x \bullet y$ is the representation for finite field multiplication.

$x \oplus y$ is the representation for an xor.

Nb is the number of 32 bit words in the state.

Nk is the number of 32 bit words in the key.

Nr is the number of rounds

2.3.2 Mathematics[3]

The AES algorithm is based on addition and multiplication using finite field elements. The finite field elements can be represented in several ways: polynomial and hexadecimal are two examples shown in equation 2.1.

$$x^5 + x^4 + x = \{32\} \qquad \textbf{(Equation 2.1)}$$

3. See the Specification for the Advanced Encryption Standard for a more detailed description of the mathematics behind the AES algorithm.

Addition is simply the xor of two numbers. Multiplication can be thought of as the multiplication of two polynomials modulo an irreducible polynomial. To multiply a byte by x ({02}) the following steps should be taken. First, the byte is shifted to the left by one bit. If the highest order bit is a 1, the modulo of the irreducible polynomial consists of the xor of the shifted byte and the irreducible polynomial. If the highest order bit is a zero, the shifted byte is already in reduced form. Equation 2.2 shows how to multiply a byte by x^2 and Equation 2.3 illustrates how to multiply a byte by x^3. Equation 2.4 demonstrates how the distributive property reduces a multiplication to use the multiplication by x algorithm described above.

$$\{32\} \bullet \{04\} = (\{32\} \bullet \{02\}) \bullet \{02\} \qquad \textbf{(Equation 2.2)}$$

$$\{32\} \bullet \{08\} = ((\{32\} \bullet \{02\}) \bullet \{02\}) \bullet \{02\} \qquad \textbf{(Equation 2.3)}$$

$$\begin{aligned} \{32\} \bullet \{26\} &= \{32\} \bullet (\{20\} \oplus \{04\} \oplus \{02\}) \\ &= ((\{32\} \bullet \{20\}) \oplus (\{32\} \bullet \{04\}) \oplus (\{32\} \bullet \{02\})) \end{aligned} \qquad \textbf{(Equation 2.4)}$$

One of the transformations requires the multiplicative inverse of a byte. The inverse of b(x) can be found by applying the extended Euclidean algorithm (outlined in Figure 2.2) to Equation 2.5 to find a(x) and c(x).[19] Equation 2.5 leads to equation 2.6, which leads to equation 2.7, resulting in the multiplicative inverse of b(x). An example of this algorithm can be found in Appendix B.

$$\begin{aligned} b(x)a(x) + m(x)c(x) &= 1 \\ m(x) &= x^8 + x^4 + x^3 + x + 1 \end{aligned} \qquad \textbf{(Equation 2.5)}$$

$$a(x) \bullet b(x) \bmod m(x) = 1 \qquad \textbf{(Equation 2.6)}$$

$$b^{-1}(x) = a(x) \bmod m(x) \qquad \textbf{(Equation 2.7)}$$

1. $a_2(x)=1$, $a_1(x)=0$, $c_2(x)=0$, $c_1(x)=1$
2. While $m(x) /= 0$ do the following:
   2.1 $q(x)=b(x)$ div $m(x)$, $r(x)=b(x)-m(x)q(x)$
   2.2 $a(x)=a_2(x)-q(x)a_1(x)$, $c(x)=c_2(x)-q(x)c_1(x)$
   2.3 $b(x)=m(x)$, $m(x)=r(x)$
   2.4 $a_2(x)=a_1(x)$, $a_1(x)=a(x)$, $c_2(x)=c_1(x)$, $c_1(x)=c(x)$
3. $a(x)=a_2(x)$, $c(x)=c_2(x)$

Figure 2.2 Extended Euclidean Algorithm. [1]

2.3.3 Transformations for Encryption

There are four transformations used for encryption: SubBytes, ShiftRows, MixColumns, and AddRoundKey. Each of these transformations are used in each round. For a key size of 128-bits there are 10 rounds; for a 192-bit key, 12 rounds; for a 256-bit key, 14 rounds. In addition the AddRoundKey function is used one additional time in round 0. The last round does not use the MixColumns transformation. Figure 2.3 shows which transformations are applied in each round.

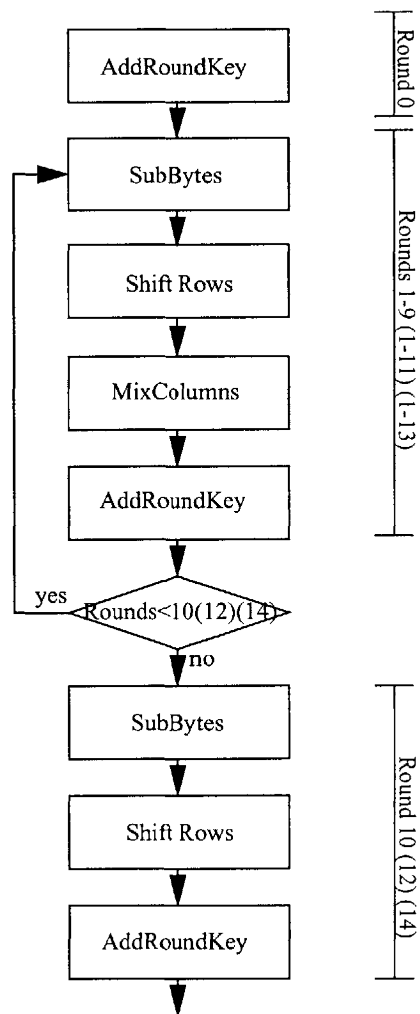


Figure 2.3 AES Encryption Algorithm.

ShiftRows is a cyclic transformation that is applied to each row of the state. Figure 2.4 shows which bytes need to be swapped.

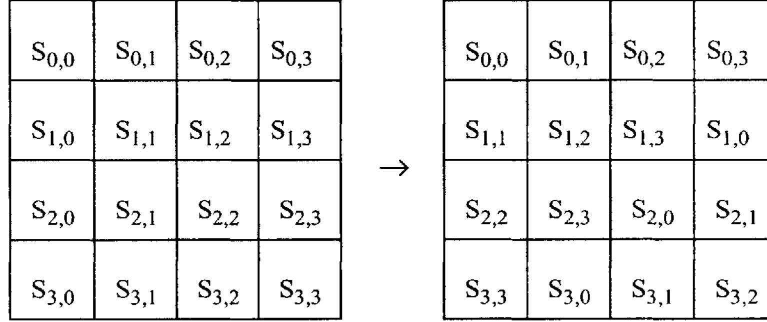


Figure 2.4 Shift Rows Transformation. [19]

SubBytes is a non-linear transformation applied to each byte of the state. The transformation is expressed in Equation 2.8 [19] where b is the multiplicative inverse of the byte that is being transformed and b_x represents one bit of the byte. The multiplicative inverse is found using the algorithm described in Section 2.3.2. In this case, the irreducible polynomial is equal to $x^8+x^4+x^3+x+1$. Note that when multiplying the two matrices in Equation 2.8, finite field addition should be used. Equation 2.9 [19] shows how to calculate b_0'.

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad \textbf{(Equation 2.8)}$$

$$b_0' = b_0 \oplus b_4 \oplus b_5 \oplus b_6 \oplus b_7 \oplus 1 \qquad \textbf{(Equation 2.9)}$$

Table 2.1 shows all values the SubBytes transformation produces with reference to an arbitrary byte {xy}. For example the transformation of byte {xy}={21} is {fd}. An example of how values in this table are computed is found in appendix B.

Table 2.1: SubBytes Transformation Lookup Table.[19]

		y															
		0	***1***	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	***fd***	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
x	**7**	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

MixColumns is the transformation shown in Equation 2.10.[19] It uses finite field multiplication where the irreducible polynomial m(x) is equal to x^4+1. Note that when multiplying the two matrices finite field addition should be used.

$$\begin{bmatrix} s_{0,c}' \\ s_{1,c}' \\ s_{2,c}' \\ s_{3,c}' \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

(Equation 2.10)

Add RoundKey is a transformation that takes the xor of the 128-bit state and the round key, an intermediate 128-bit key for each round of the algorithm. A description of how to calculate the round key can be found in Section 2.3.5.

The designers of the AES algorithm chose these transformations because they are simple, provide resistance against known attacks, they minimize the correlations between inputs and outputs, and they are invertible.

2.3.4 Transformations for Decryption

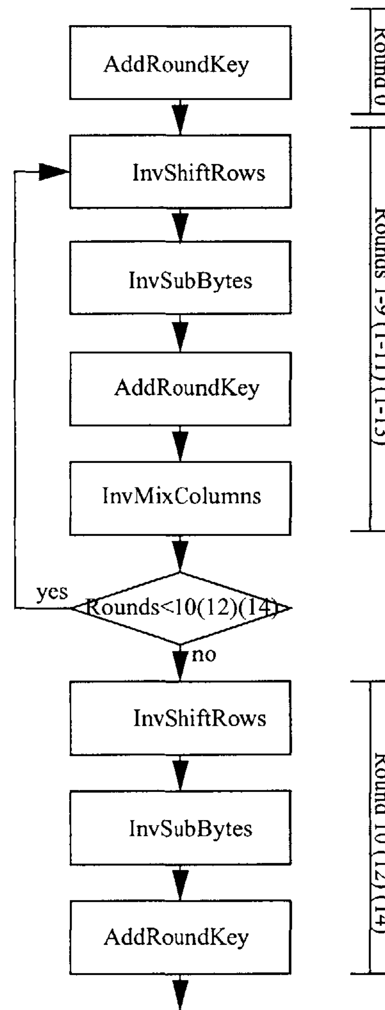


Figure 2.5 AES Decryption Algorithm.

There are also four transformations for decryption: InvSubBytes, InvShiftRows, InvMixColumns, and AddRoundKey. These transformations are the inverses of the transformations described in the previous section. As in encryption, each of these

transformations are used in each round plus the AddRoundKey is used one additional time in round 0. Figure 2.5 shows the order the transformations are applied in.

InvShiftRows, shown in Figure 2.6, is the inverse of ShiftRows, a cyclic transformation that is applied to each row of the state.

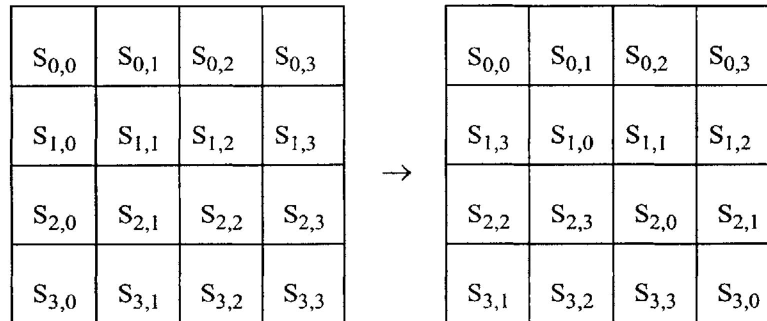


Figure 2.6 InvShiftRows Transformation. [19]

InvSubBytes is the inverse of SubBytes, a non-linear transformation that is applied to each byte of the state. Table 2.2 shows the results of the transformation. For example, the transformation of byte {xy}={21} is {7b}.

Table 2.2: InvSubBytes Transformation Lookup Table. [19]

		y															
		0	***1***	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	***7b***	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
x	**7**	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

InvMixColumns is the inverse of MixColumns, a transformation that uses finite field multiplication using the irreducible polynomial x^4+1. The transformation can be expressed by equation 2.11 [19].

$$\begin{bmatrix} s_{0,c}' \\ s_{1,c}' \\ s_{2,c}' \\ s_{3,c}' \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \textbf{(Equation 2.11)}$$

AddRoundKey for decryption is the same as for encryption. It is a transformation that takes the xor of the state and the round key.

2.3.5 Key Expansions

The AES algorithm supports 128, 192, or 256 bit keys. That key is used to produce a 128-bit intermediate key (round key) for each round of the algorithm. The first round key is used by round 0 and is the first 128 bits of the key. If the key is 256 bits then the second round key is the last 128 bits of the key. If the key is 192 bits, then the first 64 bits of the second round key is the last 64 bits of the key. The last 64 bits of the round key are found by taking a transformation of the original key. If the key is 128 bits then the second round key is a transformation of the first round key. All of the other round keys are found by transforming the previous round key if the key size is 128 bits or the previous two round keys if the key size is 192 or 256 bits. Figure 2.7 shows the algorithm for computing all of the round keys for a particular key. Sample key expansions can be found in Appendix A.

```
KeyExpansion(byte key[4 * Nk], word w[Nb * (Nr + 1)], Nk)
begin
    i = 0
    while (i < Nk)
        w[i] = word[key[4*i],key[4*i+1],key[4*i+2],key[4*i+3]]
        i = i + 1
    end while

    i = Nk
    while (i < Nb * (Nr + 1))
        word temp = w[i-1]
        if (i mod Nk=0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk = 8 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i]=w[i-Nk] xor temp
        i = i + 1
    end while
end
```

Figure 2.7 Algorithm for Key Expansion.[19]

Rcon is a function that produces a round constant of the form given by equation 2.13. Table 2.3 lists the values the Rcon function produces.

$$Rcon(i) = [x^{i-1}, \{00\}, \{00\}, \{00\}] \qquad \textbf{(Equation 2.12)}$$

Table 2.3: Rcon Values.

i	Rcon(i)
1	[{01},{00),{00},{00}]
2	[{02},{00),{00},{00}]
3	[{04},{00),{00},{00}]
4	[{08},{00),{00},{00}]
5	[{10},{00),{00},{00}]
6	[{20},{00),{00},{00}]
7	[{40},{00),{00},{00}]
8	[{80},{00),{00},{00}]
9	[{1b},{00),{00},{00}]
10	[{36},{00),{00},{00}]

RotWord is a function that performs a cyclic permutation on a four bit word. Each byte is shifted to the left by one. See an example in equation 2.12.

$$[\{01\}, \{23\}, \{45\}, \{67\}] \Rightarrow [\{23\}, \{45\}, \{67\}, \{01\}]$$ **(Equation 2.13)**

SubWord is simply the SubByte transformation applied to each byte of the word.

3 IBM Core Connect Bus Architecture

3.1 Core Connect Bus Architecture Overview

The IBM Core Connect Bus Architecture is a standard used for System-On-a-Chip (SOC) designs. The two busses used in the AES Encryption Core are the Processor Local Bus (PLB) and the Device Control Register (DCR) Bus. The PLB is used for high performance, low latency devices. The OPB is a secondary bus that is used for low-bandwidth devices. The DCR Bus is a low performance bus that is primarily used to configure a device through reading and writing to control registers. Figure 3.1 shows an

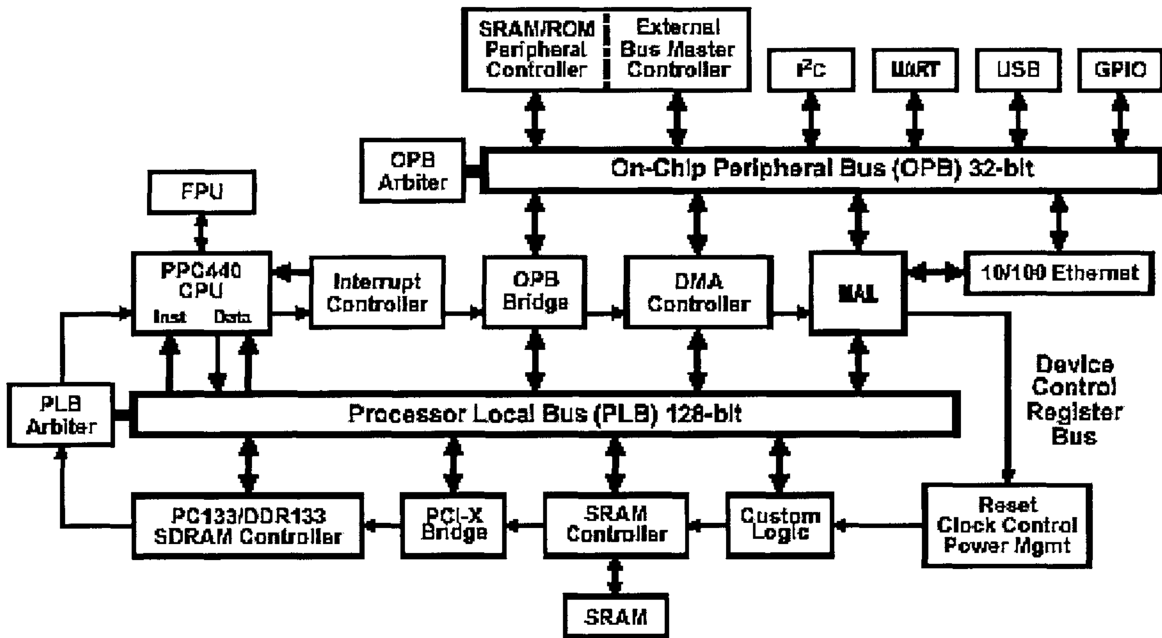


Figure 3.1 Core Connect Diagram Based on System-On-a-Chip.[15]

example of what types of devices might be on each bus.

The PLB and DCR Bus were chosen to interface with the AES Encryption Core, as the project requires an interface to the Core Connect Bus Architecture. The DCR Bus is used to configure the core with control information such as where to get the data to be

processed, where to send the processed data to, how many blocks to encrypt or decrypt, and when to start the transaction. The AES Encryption Core will take this data, process it, and request the key and data to encrypt or decrypt over the PLB. Once the data is received and processed the Encryption Core will write the data back out to memory over the PLB. Control and status information can be read from the DCR Bus.

3.2 Device Control Register Bus

The DCR Bus accesses the status and control registers for the OPB and PLB masters and slaves without using OPB and PLB bandwidth. The DCR slaves are organized in a ring architecture. The DCR Bus has a 10-bit address bus and a single 32-bit read/write data bus. The slaves and master do not need to be clocked at the same frequency. Figure 3.2 diagrams the DCR bus.

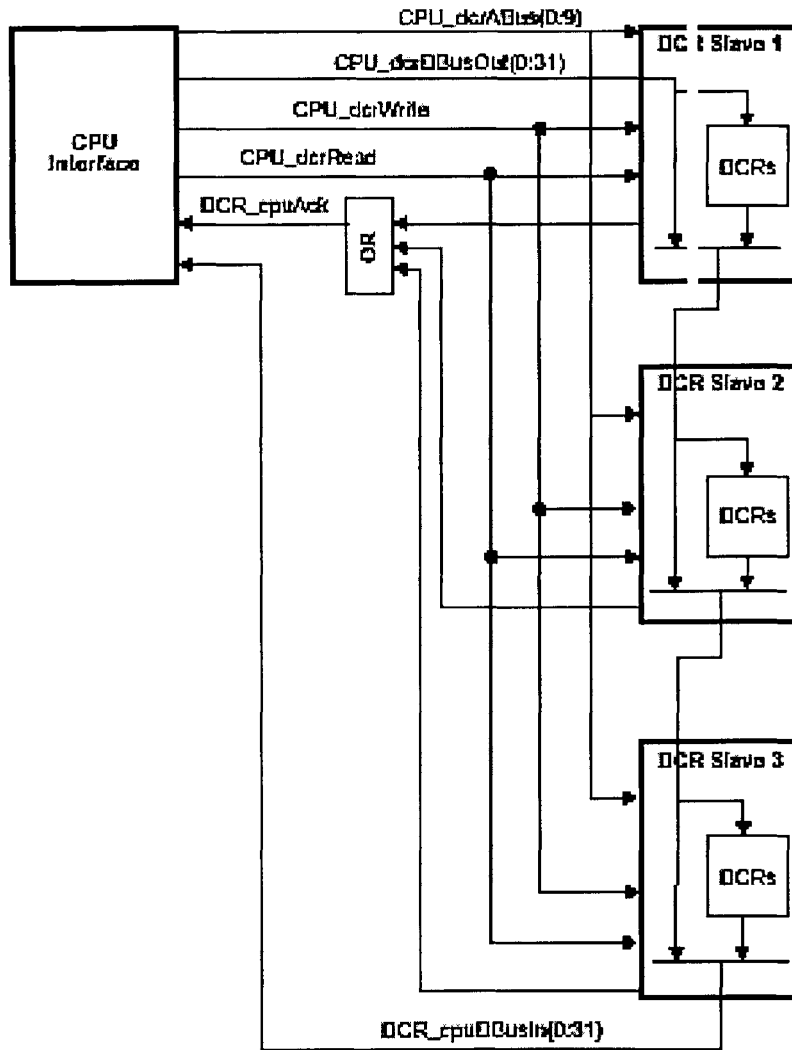


Figure 3.2 DCR Block Diagram.[17]

3.3 Processor Local Bus

The PLB is a high-performance bus that supports 16, 32, and 64-bit address and 32, 64, 128, and 256-bit data widths. There are two data busses, one for reading and the other for writing. The PLB supports single and bursting reads and writes with address pipelining. It has an arbiter that decides which master gets access to the bus. All masters and slaves must be connected to the same clock. Figure 3.3 shows how the masters and slaves are connected to the bus.

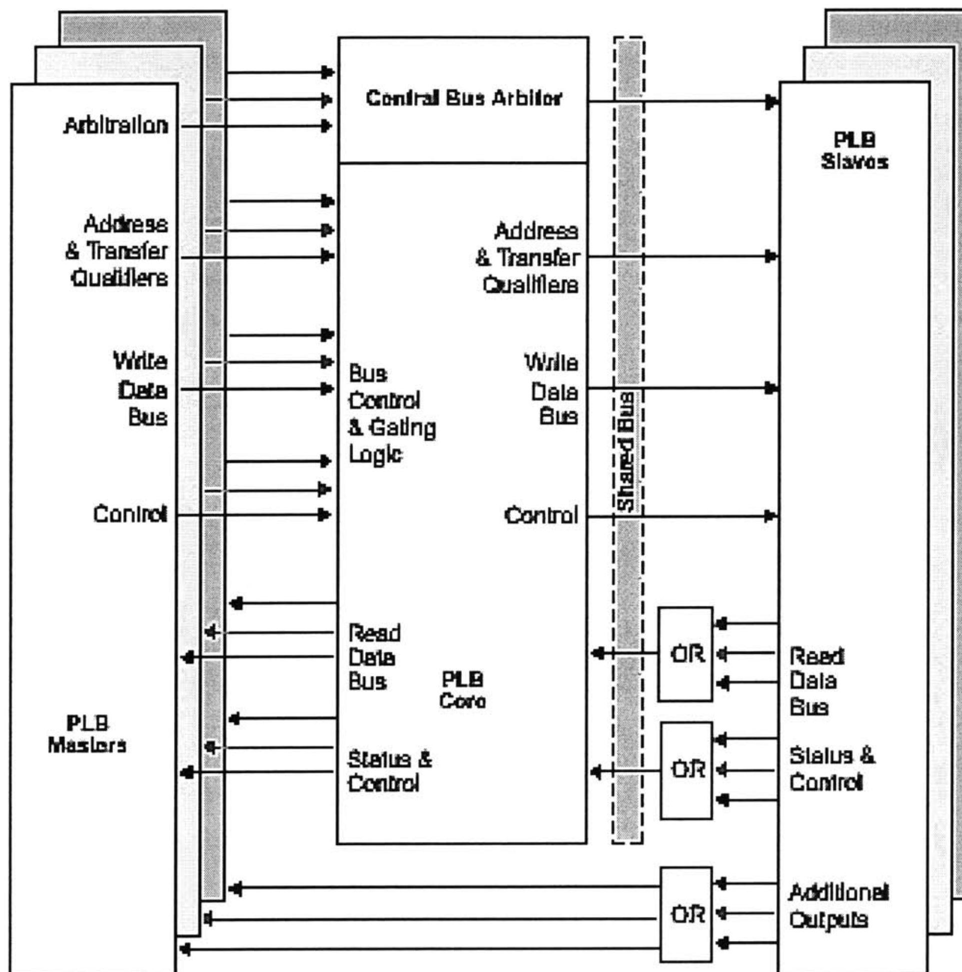


Figure 3.3 PLB Block Diagram.[21]

4 Encryption Algorithm Implementation

This AES implementation has a throughput of 1 Gbps using a 133 MHz clock. The design is pipelined so that one 128-bit block of data is processed every 16 clock cycles, meeting the target throughput. There are 15 pipeline stages giving a latency of 240 clock cycles to process one block of data.

There are two blocks in the top level AES algorithm: GetDecKey and Encrypt/ Decrypt Data. (Figure 4.1.) The GetDecKey block will iterate through the round keys and output the last one. The Encrypt/Decrypt Data block processes data using the key from the GetDecKey block. Once a key has been loaded into the system, data can be processed continuously. Table 4.1 describes the input and output signals of the AES algorithm block.

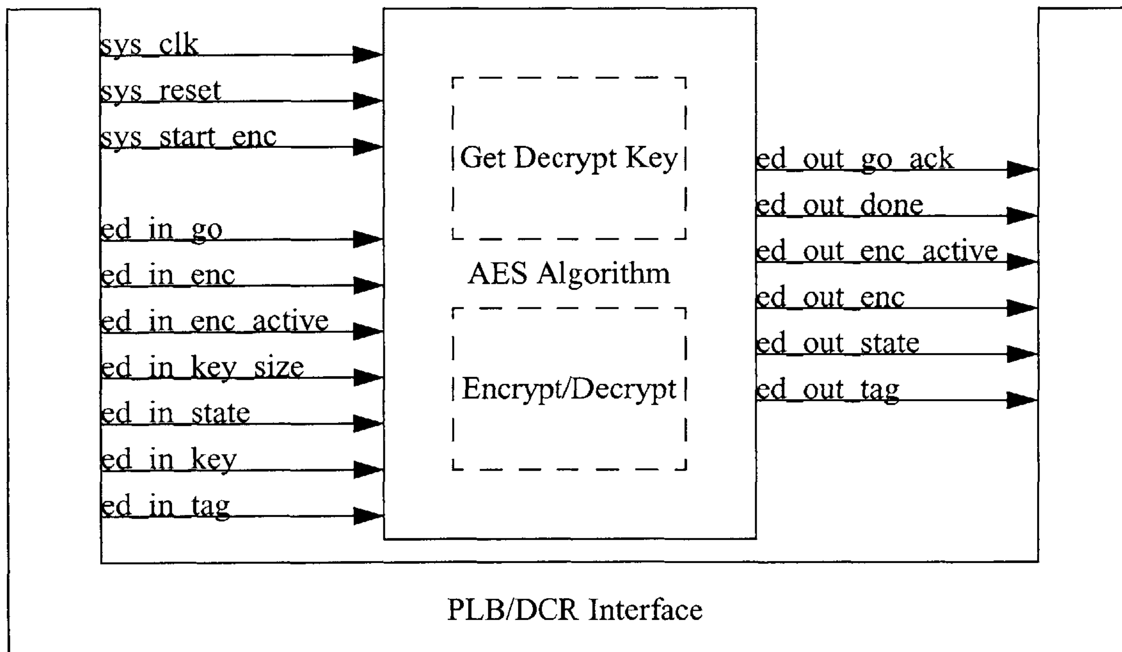


Figure 4.1 AES Algorithm Block Diagram

Table 4.1 AES Algorithm Signal Descriptions.

signal name	description
sys_clk	System clock.
sys_reset	System reset.
sys_start_enc	16 cycle pulse. One 128-bit block processed every pulse.
ed_in_enc	Encrypt/Decrypt data selector for input data.
ed_in_enc_active	Input data is valid.
ed_in_go	Start a transaction, process key information.
ed_in_key	Key for transaction.
ed_in_key_size	Size of key.
ed_in_state	Data to be encrypted or decrypted.
ed_in_tag	Signals that correspond to the input data and are to be used by the PLB interface.
ed_out_enc	Encrypt/Decrypt data selector for output data.
ed_out_enc_active	Output data is valid.
ed_out_go_ack	The key has been processed and the encrypt/decrypt data block is ready for data.
ed_out_state	Data that has been encrypted or decrypted.
ed_out_tag	Signals that correspond to the output data and are to be used by the PLB interface.

Figure 4.2 is a flow chart that outlines a transaction, where a transaction is an encryption or decryption operation that uses the same key. To start a transaction the go signal, *ed_in_go*, needs to be asserted (set to logical '1'). When this signal is asserted, the Get Decrypt Key block must process the key. When the key has been processed the go acknowledge, *ed_out_go_ack*, must be asserted. After the go acknowledge has been asserted, data can be sent through the Encrypt/Decrypt Data block. To enter data into the Encrypt/Decrypt Data block the data active signal, *ed_in_enc_active*, must be high on the rising edge of the start signal, *sys_start_enc*. If the data active signal is high on the rising edge of the start signal, then the input signals are latched into the first round. Each time the start signal pulses, the data will pass from one round to the next. When all of the data has been sent for a particular key and transaction type, then *ed_in_go* may be asserted again to process a new key.

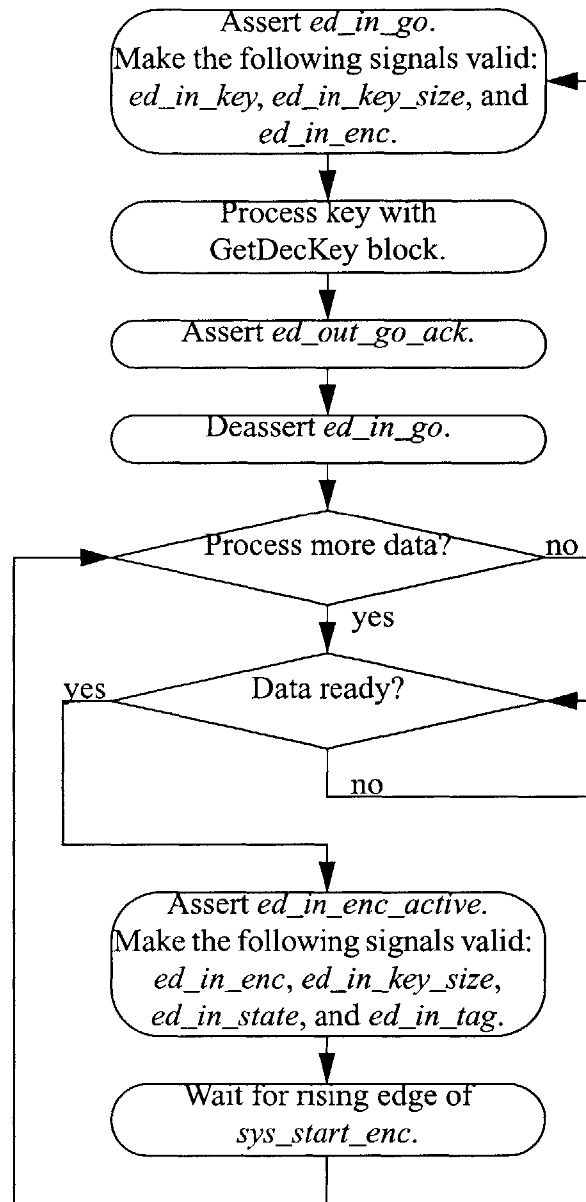


Figure 4.2 AES Algorithm Flow Chart.

Input to the Encrypt/Decrypt Data block is sent through a 15-round pipeline. Each round is 16 clock cycles. The inputs are latched into the round on the rising edge of the start signal. If the key size is 128, then rounds 10 through 13 are not used and the input data is simply latched into the next round. Rounds 12 and 13 are not used for a key size of 192. Within each round, up to four transformations are computed. The transformations for a round use most of the 16 clock cycles and minimize the logic. If the transaction is an encryption then the SubBytes, ShiftRows, and MixColumns transformations may be used. If the transaction is a decryption then InvSubBytes, InvShiftRows, and InvMixColumns may be used. Both transaction types use the AddRoundKey transformation. Figure 4.3 displays a signal diagram of a round and Table 4.2 describes each of the round signals. A block diagram for each round is specified in Figures 4.4 through 4.7. Note that there are 80 bits reserved for a tag to be used by the interface.

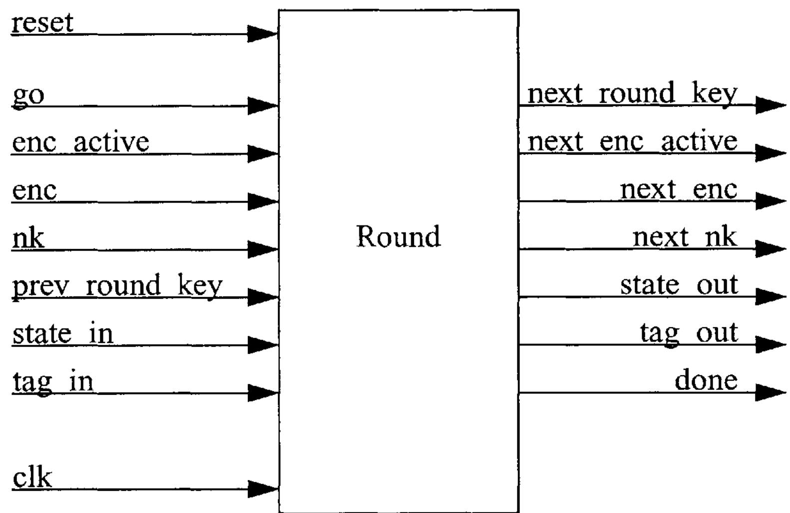


Figure 4.3 Round Signal Diagram.

Table 4.2 Round Signal Descriptions.

Signal Name	Signal Description
generic: round	Indicates the round number to set up constants in the VHDL.
clk	System clock.
reset	System reset.
go	16 cycle pulse. One 128-bit block processed every pulse.
enc_active	Encrypt/Decrypt data selector for input data.
enc	Indicates input data is valid.
nk(3:0)	Indicates the number of words in the key.
prev_round_key(255:0)	Holds the value of the two previous round keys
state_in(127:0)	Holds the value of the input state.
tag_in(79:0)	Holds the value of the tag.
next_round_key(255:0)	Holds the value of the previous round key and the current round key. This will be sent to the next round.
next_enc_active	Indicates the value of enc_active for the next round.
next_enc	Indicates the value of enc for the next round.
next_nk(3:0)	Indicates the value of nk for the next round.
state_out(127:0)	Holds the value of the output state. This will be sent to the next round.
tag_out(79:0)	Holds the value of the tag for the next round.
done	Indicates that all of the output signals are valid.

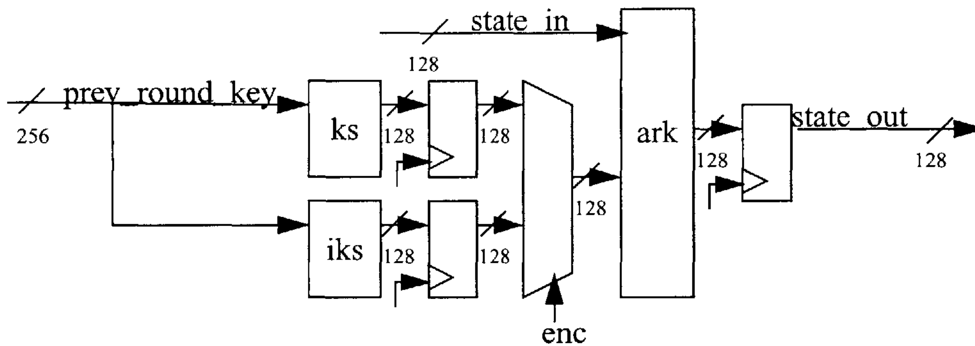


Figure 4.4 Round 0 Block Diagram. Not all signals are shown.

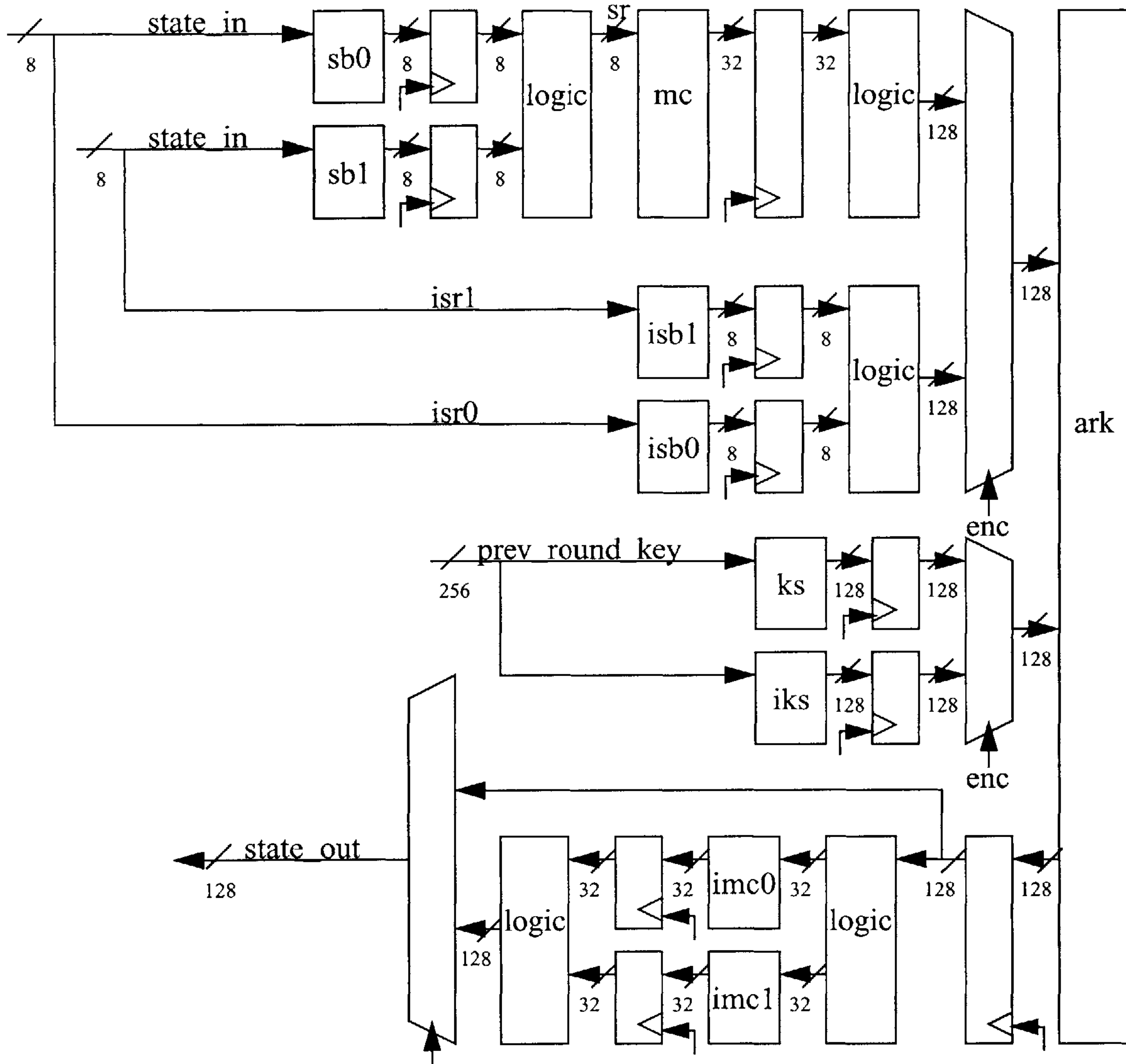


Figure 4.5 Rounds 1-9 Block Diagram. Not all signals are shown.

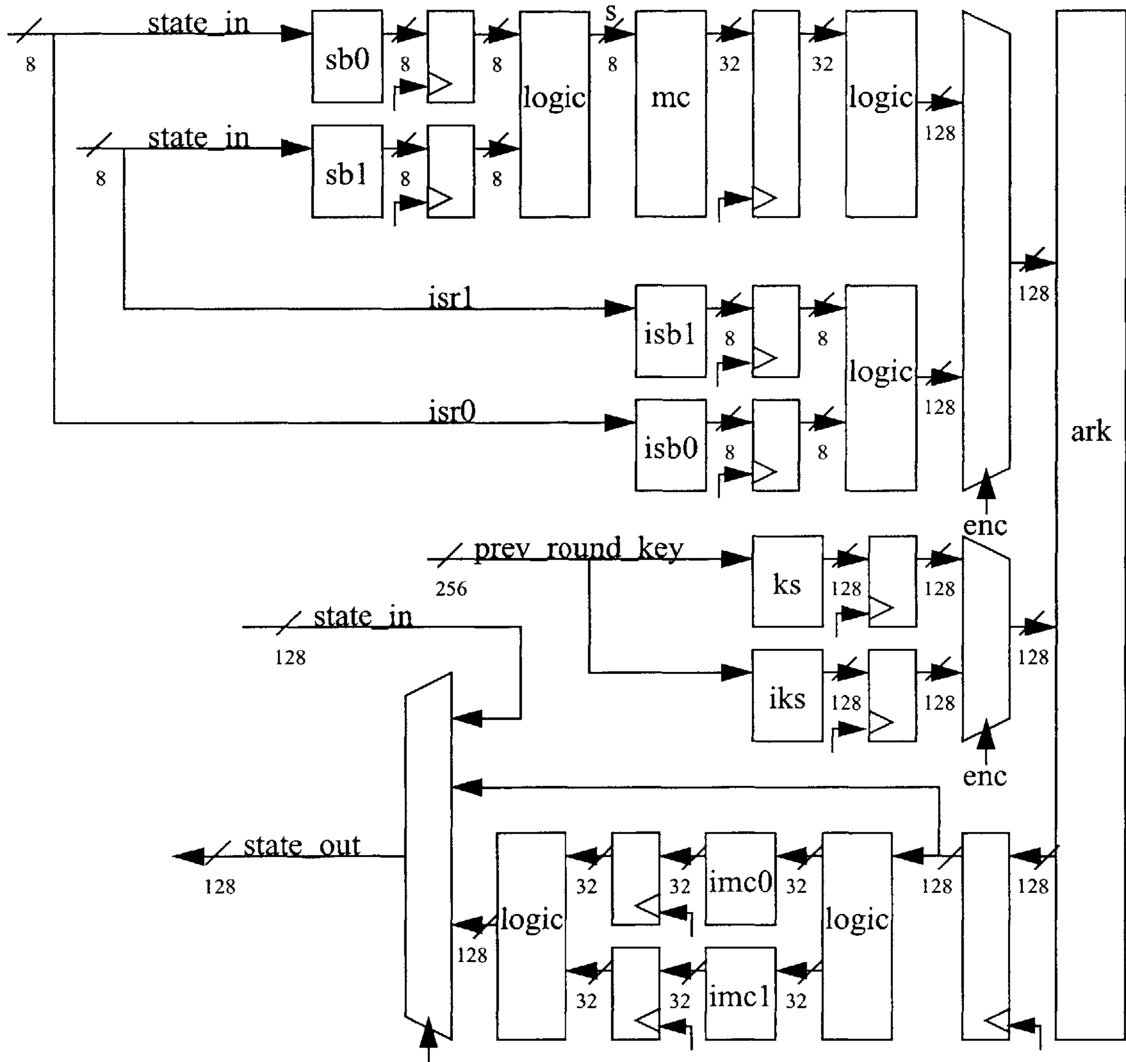


Figure 4.6 Rounds 10-13 Block Diagram. Not all signals are shown.

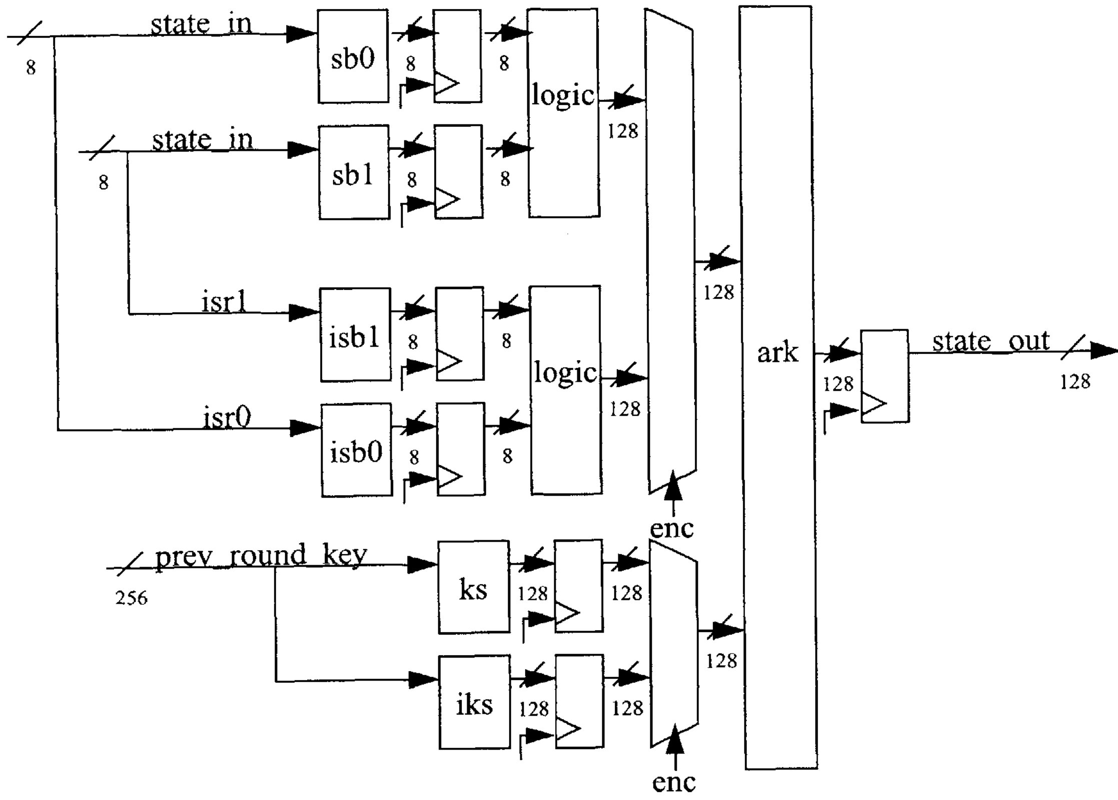


Figure 4.7 Round 14 Block Diagram. Not all signals are shown.

4.1 Key Setup

When a new key is received, it needs to be processed so that it can be sent into the first round. If the transaction is an encryption transaction then the first 128 bits are used as the round key for round 0. If the key size is 192 or 256, then the last 64 or 128 bits are saved to be used as the round key for round 1. If the transaction is a decryption transaction, then the key needs to be processed to get the last round key. For a key size of 128, there are 10 round keys; for a key size of 192, 12 round keys; for a key size of 256, 14 round keys. The algorithm for finding each round key can be seen in Section 2.3.5.

To implement the key setup for decryption, each key is found one at a time. Each round key requires one SubWord transformation, which uses the SubByte function. This implementation replicates the SubByte function four times, once for each time the SubByte function is used in the SubWord transformation, reducing the latency to find the decryption key. The design is optimized for encrypting or decrypting large amounts of data while using a single key. For large amounts of data, the key setup time is negligible and it may make more sense to not replicate the SubByte function. However, for small amounts of data, the latency is improved by replicating the SubByte function. The Rcon function is implemented once as it is only used one time per round key. Inputs to the GetDecKey Block indicate which round key needs to be found. A signal diagram of the GetDecKey Block is displayed in Figure 4.8 and descriptions of the signals are listed in Table 4.3.

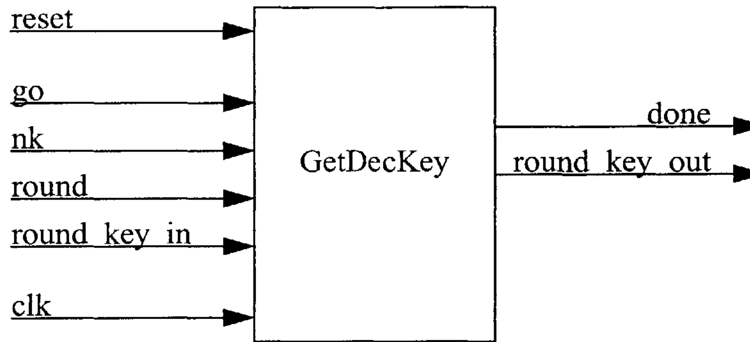


Figure 4.8 GetDecKey Block Diagram.

Table 4.3 GetDecKey Signal Descriptions.

Signal Name	Description
reset	System reset.
clk	System clock.
go	Indicates that the inputs are valid and that the next round key should be found.
nk(3:0)	Indicates the number of words in the key.
round(3:0)	Indicates the round number that should be found.
round_key_in(255:0)	key_size=128: Bits 127:0 hold the previous round key. Other bits unused. key_size/=128: Holds the previous two round keys. The upper bits hold the round key from two rounds before and the lower bits hold the previous round key. (For round 0 and 1, bits 127:0 hold the round key for round 0 and bits 255:128 hold the round key for round 1.)
done	Indicates that round_key_out is valid.
round_key_out(255:0)	Holds the previous round key in the upper bits and the round key for the current round in the lower bits.

A timing diagram for the GetDecKey block can be found in Figure 4.9. When the go signal is asserted, the other inputs must be valid. At this point the GetDecKey block latches in all of the inputs. The round and key size will be applied to the input key to get the next round key. When the transformation is complete the done signal will be asserted with a valid output round key. The done signal is asserted for one clock cycle.

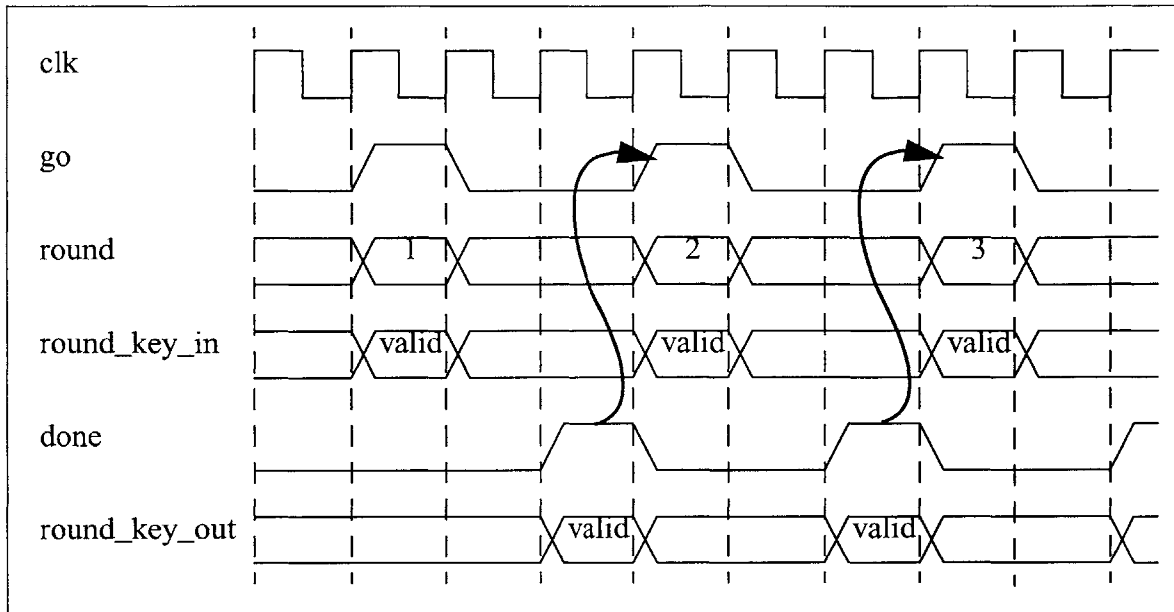


Figure 4.9 GetDecKey Timing Diagram. Note that this is not to scale. From the time go is asserted, it takes 6 cycles for done to be asserted.

4.2 Encryption

There are four main transformations used on the state for encryption: SubBytes, ShiftRows, MixColumns, and AddRoundKey. The key scheduler is a transformation applied to the key. Each of these transformations are described in Chapter 2.

A timing diagram of an encryption round is displayed in Figure 4.10. A go signal is sent every 16 clock cycles. If *enc* and *enc_active* are both high when the go signal is received, then the output state and the next round key are computed. First the SubByte block and the Key Scheduler begins processing. When the SubByte block finishes, the ShiftColumns transformation is applied to the bits and the resulting data is sent to the MixColumns block. After both the Key Scheduler and the MixColumns block complete, the state and the key is sent to the AddRoundKey block. When the AddRoundKey block completes, the key and state are valid output until the next go signal is received.

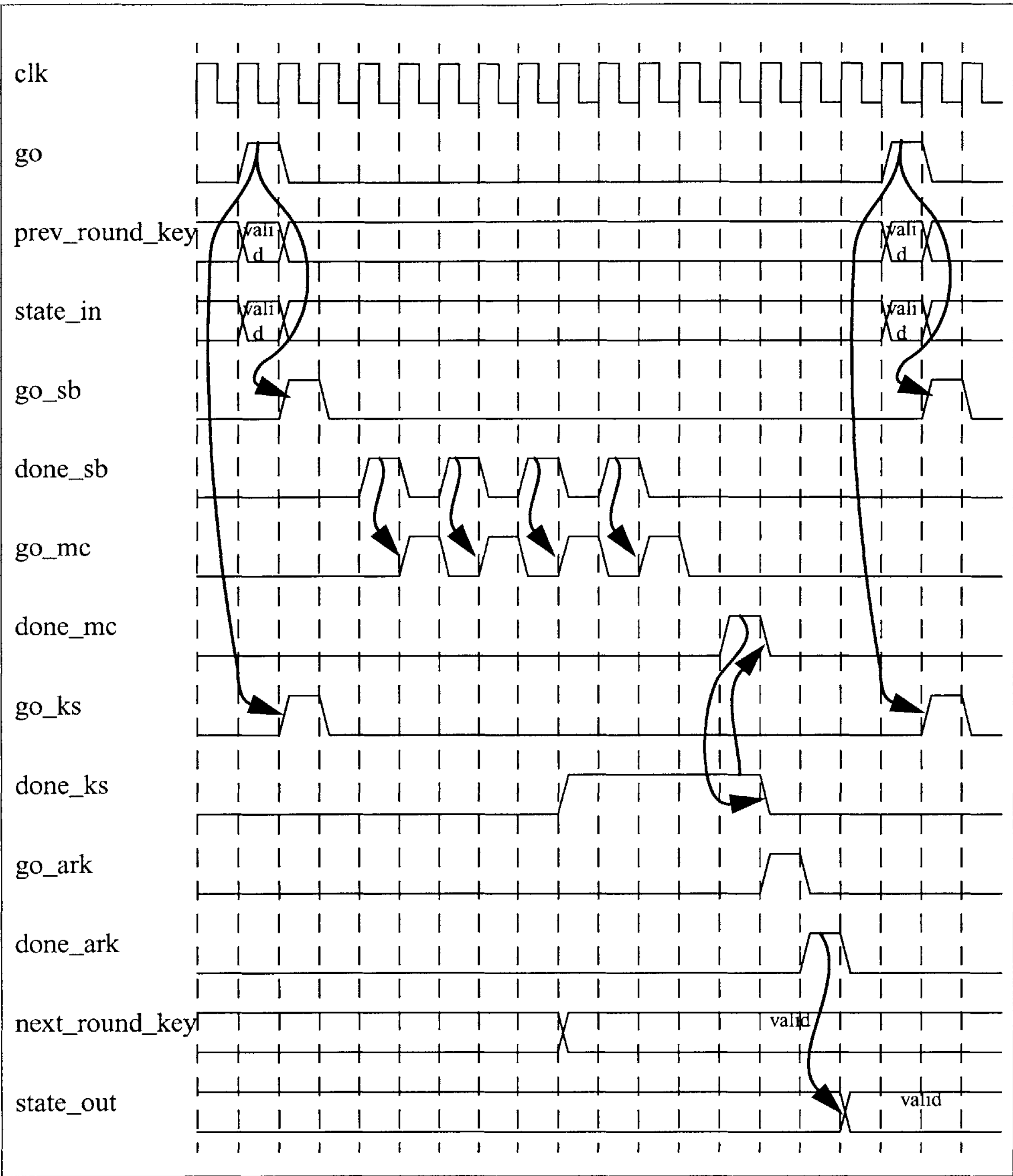


Figure 4.10 Encryption Round Timing Diagram.

4.2.1 SubBytes Implementation

SubBytes is used in rounds 1 through 14 and is implemented by a lookup table (Table 2.1) with 256 entries, each of which is 8 bits wide. The lookup table was implemented because the computation of the multiplicative inverse can take a large number of cycles to complete. Because the number of cycles to compute the inverse is large, more rounds would need to be inserted in parallel with the current rounds. While the implementation of the multiplicative inverse and the additional xors could be smaller than the lookup table, the extra replication that would be needed to achieve the target throughput did not yield a significant savings. The SubBytes transformation is replicated twice for each round. Each of these replicas are used 8 times per round, processing a total of 16 bytes. A block diagram of the SubBytes transformation is found in Figure 4.11 and a list of signal descriptions are found in Table 4.4.

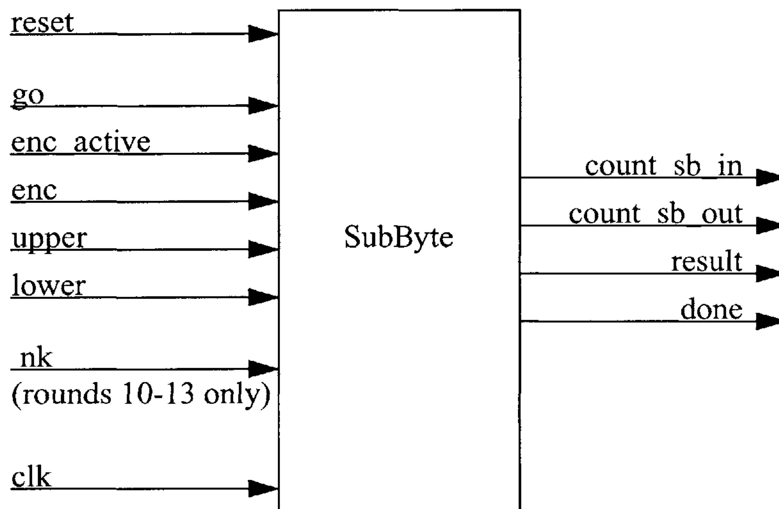


Figure 4.11 SubBytes Signal Diagram.

Table 4.4 SubBytes Signal Descriptions.

Signal Name	Signal Description
generic: num_inputs(4:0)	Indicates the number of bytes that need to be transformed in a particular round. Set to 8.
generic: num_outputs(2:0)	Indicates the number of locations the output bytes can go to. Set to 2.
clk	System clock.
reset	System reset.
go	Indicates inputs are ready and SubByte transformation should begin.
enc_active	Indicates that the round is processing data.
enc	Indicates whether the transaction is an encryption or decryption
upper(3:0)	Upper four bits of the byte to be transformed.
lower(3:0)	Lower four bits of the byte to be transformed.
nk(3:0)	Indicates the number of works in the key.
count_sb_in(4:0)	Indicates which byte should be input to the SubByte block. (Max of num_inputs.)
count_sb_out(2:0)	Indicates which byte is being output from the SubByte block.
result(7:0)	Result of the SubByte transformation.
done	Indicates that the last of the bytes specified by count_sb_out have been processed.

A timing diagram for the SubByte block is found in figure 4.12. When the SubByte transformation is needed, the go signal is asserted. If the *enc_active* and the *enc* signals are high, then the SubByte block will process the inputs, otherwise the result is hex "00". The *count_sb_in* signal indicates which byte to input. Before the go is asserted, the *count_sb_in* signal is set to 0. *Count_sb_in* counts up to the number of inputs, *num_inputs*. The result is valid the clock cycle after the input is valid. At this point, the system has one clock cycle to make the inputs valid. As the results become valid, the *count_sb_out* signal counts up to the number of outputs, *num_outputs*. In this implementation, there are eight bytes that will be input and four sets of two bytes are output. Each time the done signal is asserted, two bytes are loaded into the MixColumns

transformation. When the SubBytes block is finished *count_sb_in* and *count_sb_out* are set back to 0.

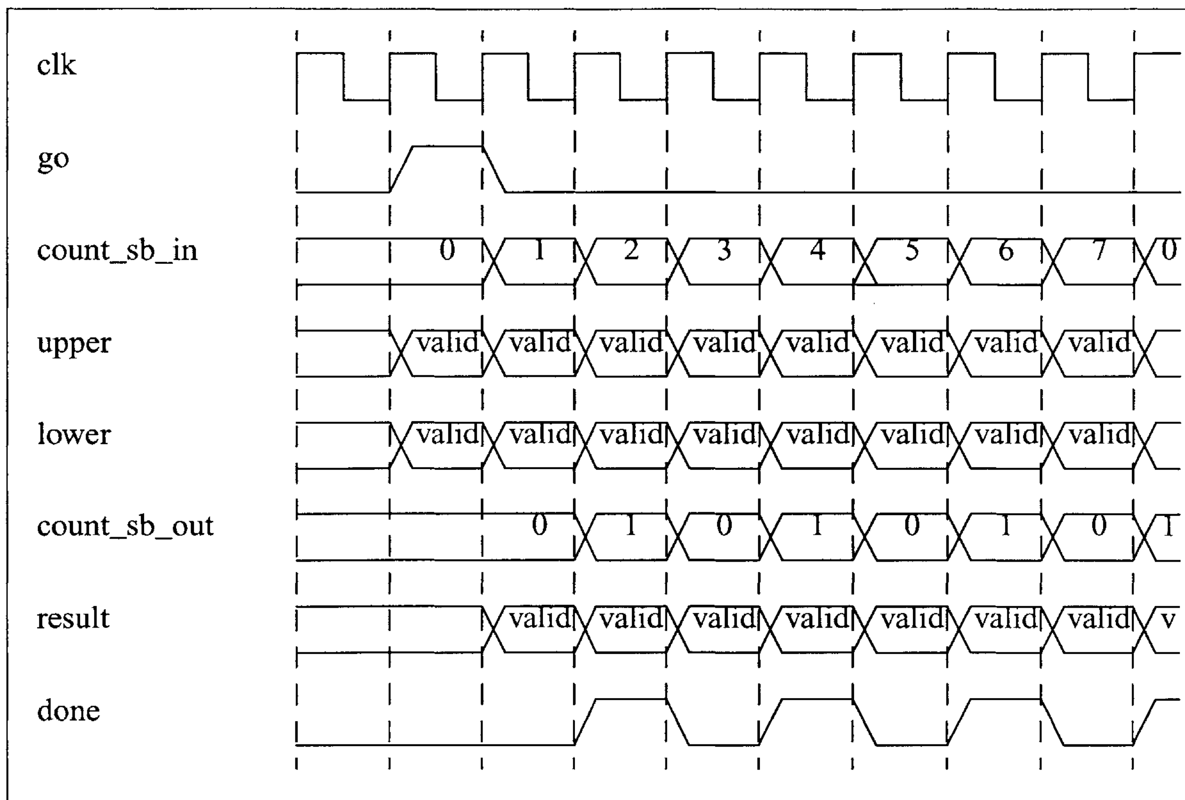


Figure 4.12 SubBytes Timing Diagram

4.2.2 ShiftRows Implementation

After the SubBytes transformation is applied, the ShiftRows transformation is implemented by changing the order of the bits in the state according to Figure 2.4. This does not take any logic or clock cycles. ShiftRows is used in rounds 1 through 14.

4.2.3 MixColumns Implementation

After the bits have been rearranged, the MixColumns transformation is applied in rounds 1 through 13. The MixColumns transformation is implemented with combinational logic using the multiplication by x algorithm described in Section 2.3.2. This multiplication takes one cycle to complete. The MixColumns transformation is not replicated and is reused four times, once for each word, taking a total of 4 clock cycles. The block diagram of the MixColumns transformation can be found in Figure 4.13 and the signal descriptions are listed in Table 4.5.

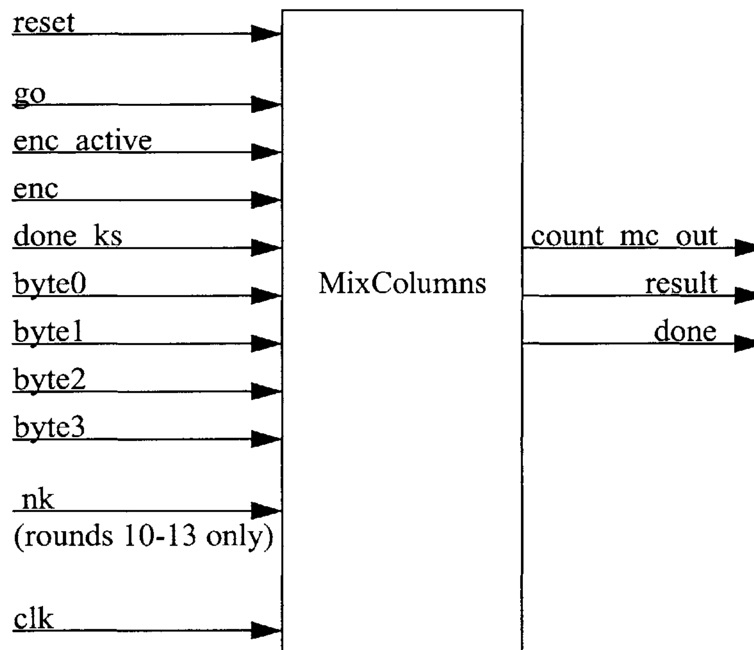


Figure 4.13 MixColumns Signal Diagram.

Table 4.5 MixColumns Signal Descriptions

Signal Name	Signal Description
generic: num_sb_blocks(4:0)	Indicates the number of time the SubBytes block is replicated. Set to 2.
generic: num_outputs(2:0)	Indicates how many locations the result can go to. Set to 4.
clk	System clock.
reset	System reset.
go	Indicates that valid data is on the inputs and that the MixColumns transformation should begin.
enc_active	Indicates that the round is processing data.
enc	Indicates whether the transaction is an encryption or decryption
done_ks	Indicates that the round key is ready and that the done signal can be deasserted.
byte0(7:0)	Bits 7:0 of the input word.
byte1(7:0)	Bits 15:8 of the input word.
byte2(7:0)	Bits 23:16 of the input word.
byte3(7:0)	Bits 31:24 of the input word.
nk(3:0)	Indicates the number of words in the key.
count_mc_out(2:0)	Indicates which word is being output.
result(31:0)	Result of the MixColumns transformation.
done	Indicates that the result holds valid data.

The done signals from the SubBytes block are used as the go signals for the MixColumns block. The MixColumns block gets two bytes of data from each of the SubByte blocks. When all four bytes of data are ready, the data is processed and the result is valid on the next clock cycle. The data stays valid until *count_mc_out* changes. In this implementation, the number of outputs, *num_outputs*, is set to 4. When there is no valid data, *count_mc_out* stays at 4. When all four words of data have been output, the done signal becomes high and stays high until it sees that the done signal from the key scheduler, *done_ks*, is also high.

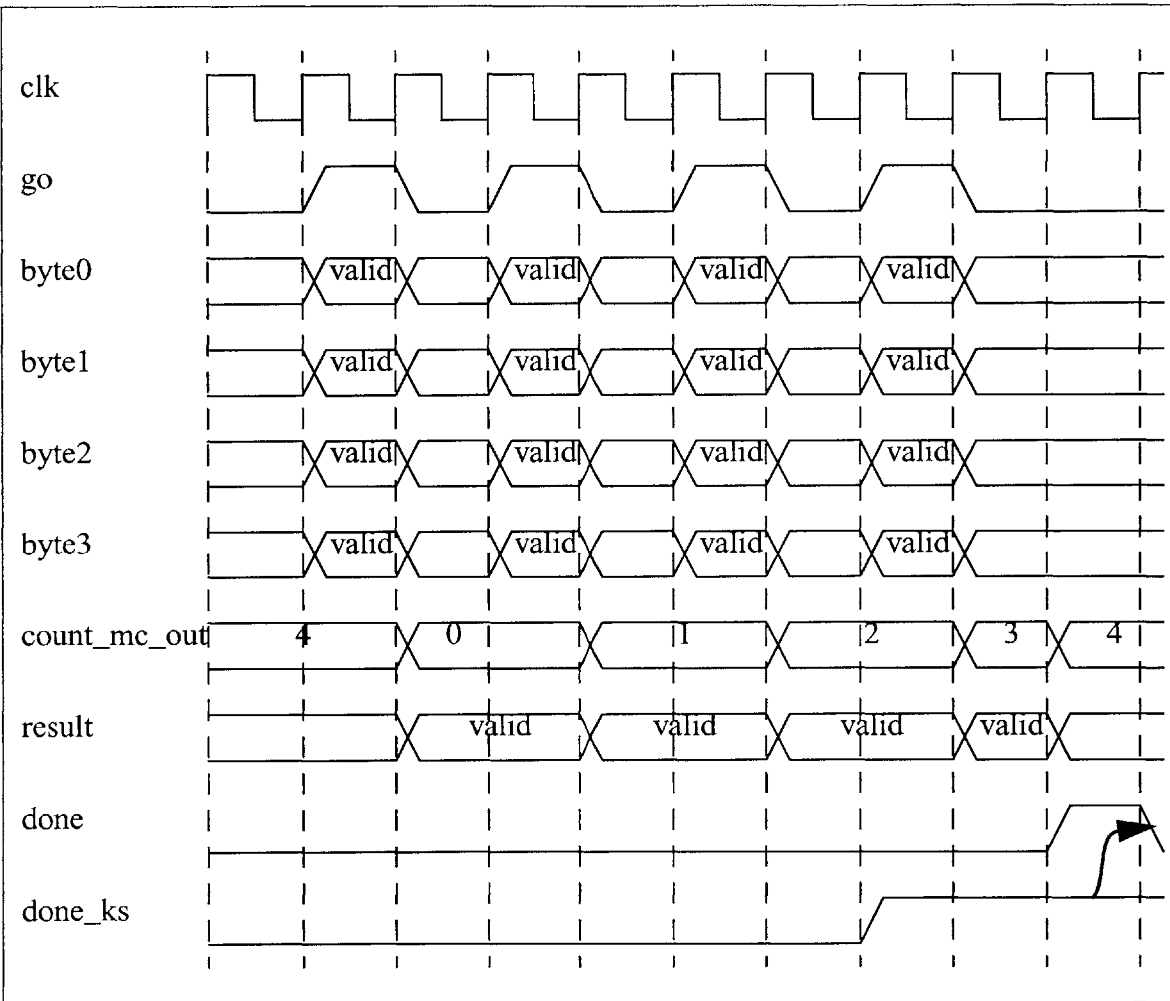


Figure 4.14 MixColumns Timing Diagram.

4.2.4 Key Scheduler Implementation

The key scheduler computes the round key for each round using the previous round key. (For key sizes of 192 or 256 the key scheduler uses the previous two round keys.) The Rcon transformation is implemented once. The SubByte transformation is replicated 4 times, once for each time it is used in the SubWord transformation. The other logic is implemented custom to each round. A signal diagram is displayed in figure 4.15 and table 4.6 gives a description of each of the signals.

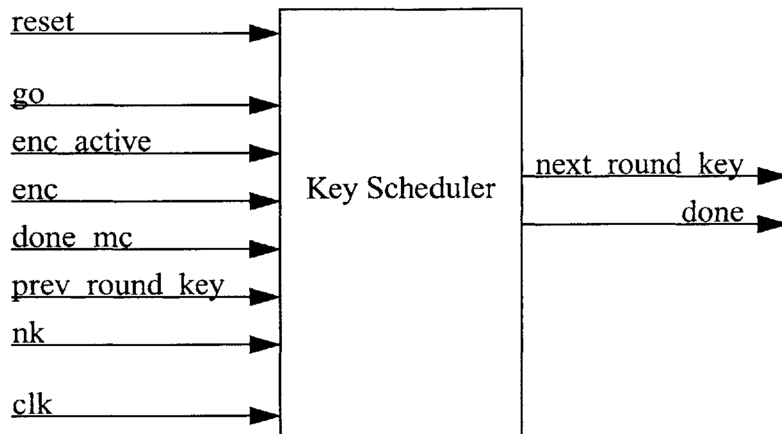


Figure 4.15 KeyScheduler Signal Diagram.

Table 4.6 KeyScheduler Signal Descriptions

Signal Name	Signal description
generic: num_mc_blocks(3:0)	Indicates the number of times the MixColumns block was replicated. Set to 1.
generic: round(3:0)	Indicates the round number
generic: num_sb_blocks(4:0)	Indicates the number of times the SubByte block is replicated. Set to 2.
clk	System clock.
reset	System reset.
go	Indicates that valid data is on the inputs and that the Key Scheduler should begin.
enc_active	Indicates that the round is processing data.
enc	Indicates whether the transaction is an encryption or decryption
done_mc	Indicates that the MixColumns transformation is complete and that the done signal can be deasserted. (For round 14 this signal is done_sb and it indicates that the SubBytes transformation is complete.)
prev_round_key(255:0)	Holds the value of the last two round keys.
nk(3:0)	Indicates the number of words in the key.
next_round_key(255:0)	Result of the Key Scheduler. Holds the value of the current round key and the previous round key. For round 14 this signal is only 127 bits and holds the value of the current round key.
done	Indicates that the next_round_key is valid.

Figure 4.16 gives an example of how the key scheduler works. The key scheduler latches in the previous two round keys when it receives a go signal. First, the SubWord transformation is performed. After the result is valid, it is used to find the first word of the next round key. All of the other words in the round key are then found using the previous word and the previous round key. When all of the words of the round key have been found, the done signal is asserted. It stays asserted until the done signal from the MixColumns transformation has been received.

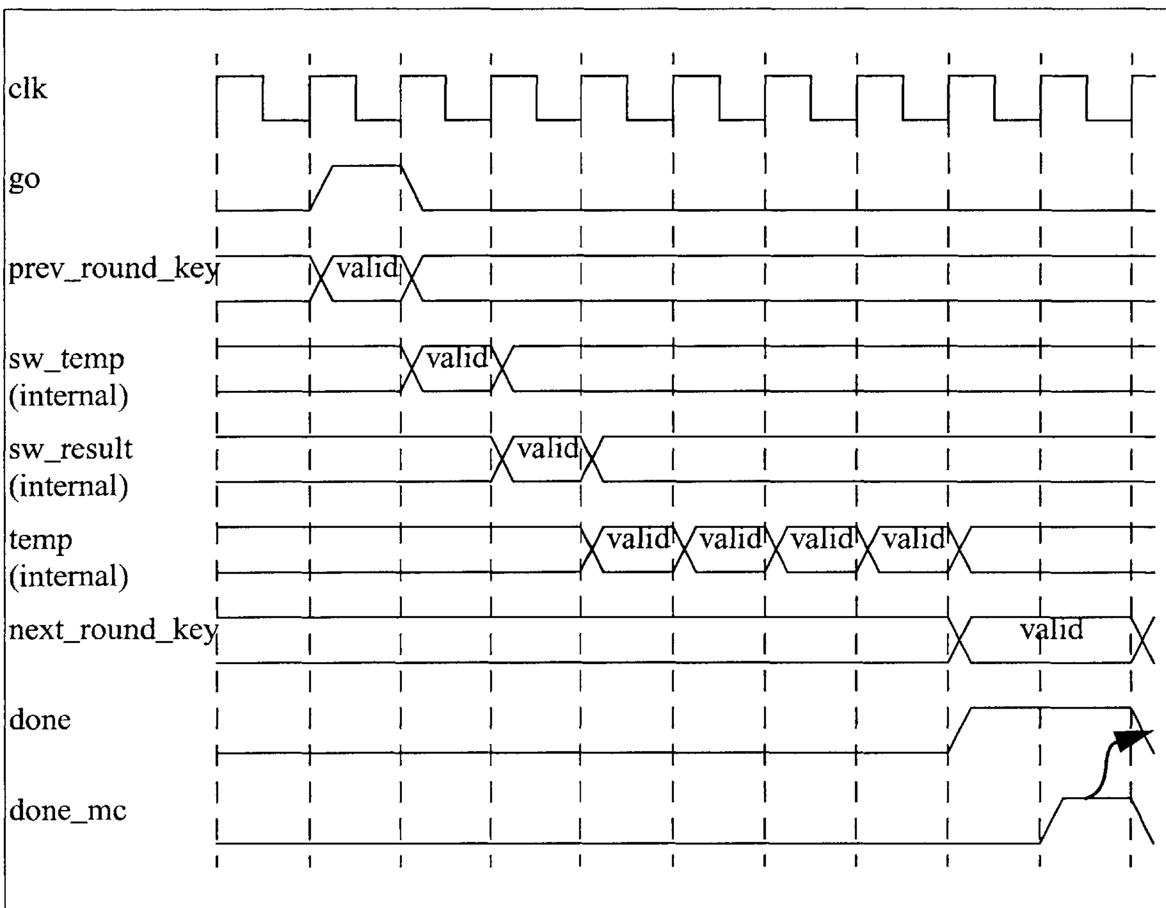


Figure 4.16 KeyScheduler Timing Diagram.

4.2.5 AddRoundKey Implementation

Once the MixColumns transformation and the key scheduler has completed, the AddRoundKey transformation is applied. This is a simple xor of the result of the MixColumns transformation and the round key. Figure 4.17 displays a signal diagram of the AddRoundKey block and Table 4.7 provides descriptions of the signals.

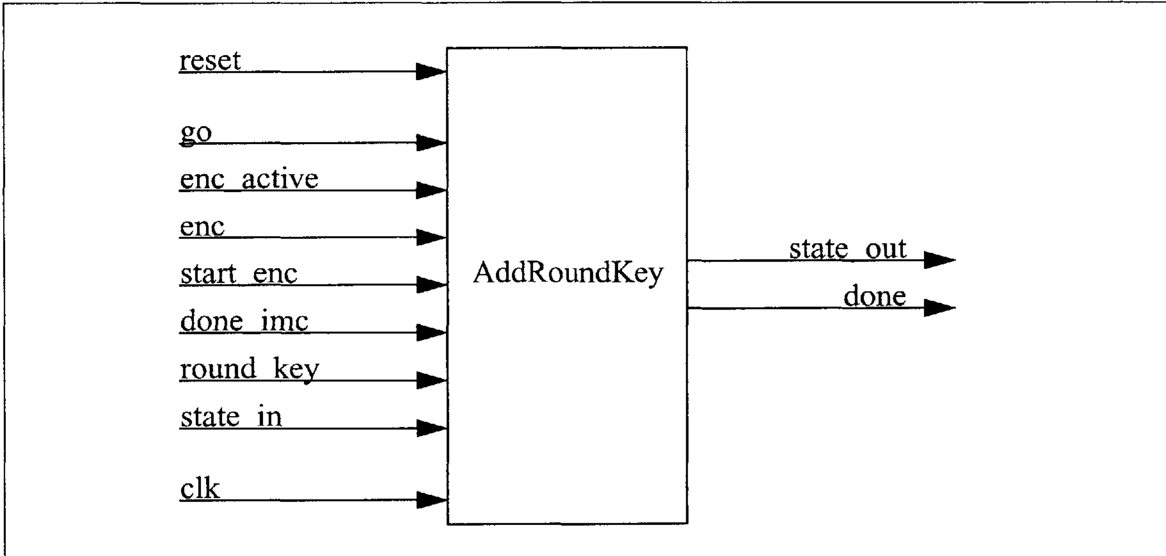


Figure 4.17 AddRoundKey Signal Diagram.

Table 4.7 AddRoundKey Signal Descriptions

Signal Name	Signal Description
generic: num_imc_blocks(3:0)	Indicates the number of times the InvMixColumns block is replicated. Set to 2.
clk	System clock.
reset	System reset.
go	Indicates that valid data is on the inputs and that the AddRoundKey transformation should begin.
enc_active	Indicates that the round is processing data.
enc	Indicates whether the transaction is an encryption or decryption
start_enc	Indicates that new data is coming in on the pipeline.
done_imc	Indicates that the InvMixColumns transformation is complete and that the done signal can be deasserted.

Table 4.7 AddRoundKey Signal Descriptions

Signal Name	Signal Description
round_key(127:0)	Holds the value of the current round key.
state_in(127:0)	Holds the value of the current state.
state_out(127:0)	Holds the value of the result of the AddRoundKey transformation.
done	Indicates that state_out is valid.

A timing diagram for the AddRoundKey transformation can be seen in Figure 4.18. The AddRoundKey block will get a go signal when the round key and the input state are valid. The transformation takes one clock cycle to complete and the done signal is asserted with the output state.

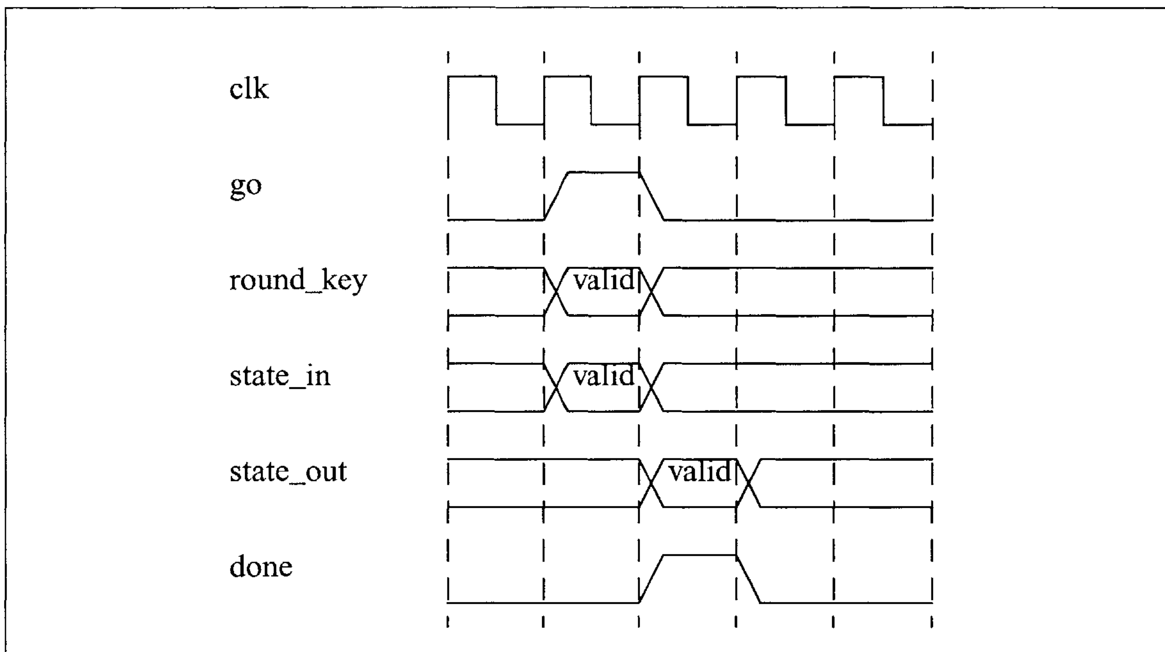


Figure 4.18 AddRoundKey Timing Diagram.

Appendix C contains a worked out example of an encryption.

4.3 Decryption

The AES decryption algorithm is simply the inverse of the encryption algorithm. The four transformations that are used on the state are: InvShiftRows, InvSubBytes, AddRoundKey, and InvMixColumns. Section 2.3.4 gives a detailed description of the transformations. Before a decryption round begins, the input key must be processed by the GetDecKey block to provide the first round key for decryption. The GetDecKey block is described in Section 4.1.

A timing diagram of a decryption round is found in Figure 4.19. A go signal is sent every 16 clock cycles. When the go is received and *enc_active* is high and *enc* is low, then the output state and the next round key are computed. First, the state bits will be switched around according to the InvShiftRows transformation described by figure 2.6. Next the InvSubBytes and the Key Scheduler blocks will begin processing. The round keys will be computed in the reverse order as in encryption. When InvSubBytes and the Key Scheduler are complete, the state and the round key are input to the AddRoundKey block. After the AddRoundKey block are completed the state is sent to the InvMixColumns block. Once the InvMixColumns block completes, the output state is valid.

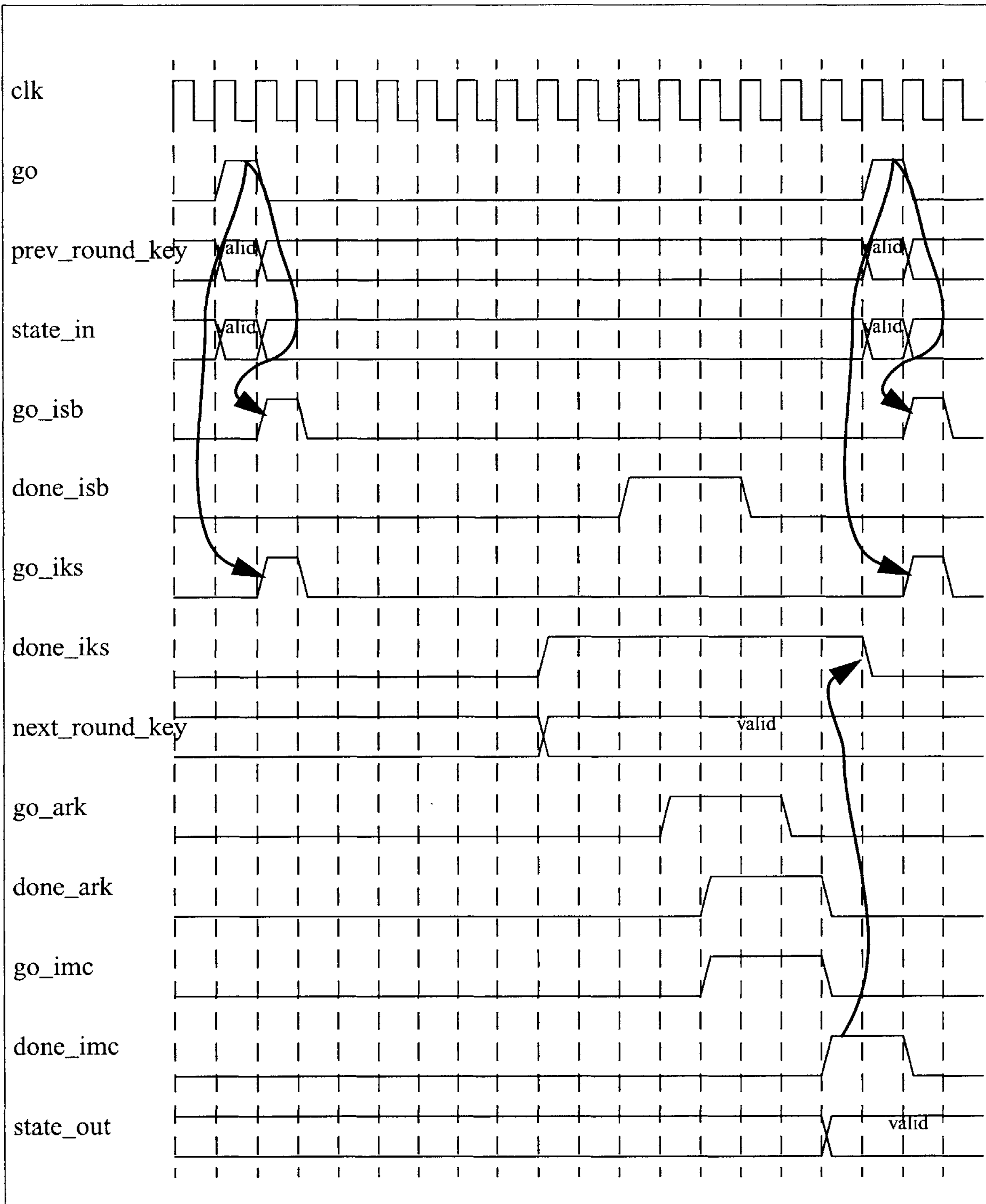


Figure 4.19 Decryption Round Timing Diagram.

4.3.1 InvShiftRows Implementation

The InvShiftRows transformation is applied first in decryption. This is a simple rearrangement of bits in the state and is the inverse of the ShiftRows transformation. This does not use any additional logic.

4.3.2 InvSubBytes Implementation

The InvSubBytes transformation is applied after the InvShiftRows transformation. It is the inverse of the SubBytes transformation and is implemented as a lookup table for the same reasons that the SubBytes transformation is implemented as a lookup table. The lookup table is shown in Section 4.2.1. InvSubBytes is replicated twice, each of which are reused 8 times per round, processing a total of 16 bytes of data. A signal diagram of the InvSubBytes transformation is found in Figure 4.20 and a list of signal descriptions is found in table 4.8.

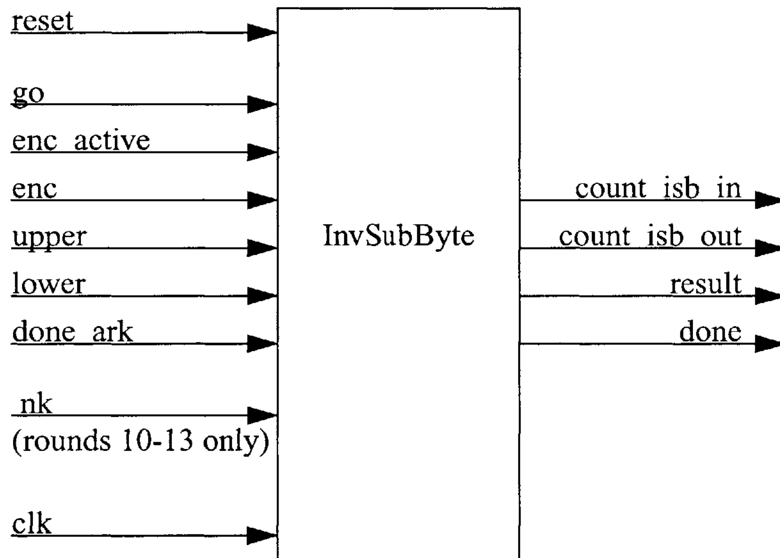


Figure 4.20 InvSubBytes Signal Diagram.

Table 4.8 InvSubBytes Signal Descriptions.

Signal Name	Signal Description
generic: num_inputs(4:0)	Indicates the number of bytes that need to be transformed in a particular round. Set to 8.
generic: num_outputs(2:0)	Indicates the number of locations the output bytes can go to. Set to 8.
clk	System clock.
reset	System reset.
go	Indicates inputs are ready and SubByte transformation should begin.
enc_active	Indicates that the round is processing data.
enc	Indicates whether the transaction is an encryption or decryption
upper(3:0)	Upper four bits of the byte to be transformed.
lower(3:0)	Lower four bits of the byte to be transformed.
done_ark	Indicates when the AddRoundKey block has completed.
nk(3:0)	Indicates the number of works in the key.
count_isb_in(4:0)	Indicates which byte should be input to the InvSubByte block. (Max of num_inputs.)
count_isb_out(4:0)	Indicates which byte is being output from the InvSubByte block.
result(7:0)	Result of the SubByte transformation.
done	Indicates that the last of the bytes specified by count_sb_out have been processed.

A timing diagram for the InvSubByte block is found in Figure 4.21. If the *enc_active* signal is high and the *enc* signal is low when the go signal is asserted, the InvSubByte block should begin processing. *Count_isb_in* indicates what byte should be input to the block and *count_isb_out* indicates what byte is being output. When all eight bytes have been output, the done signal should be asserted until the done signal from the AddRoundKey block, *done_ark*, has been asserted.

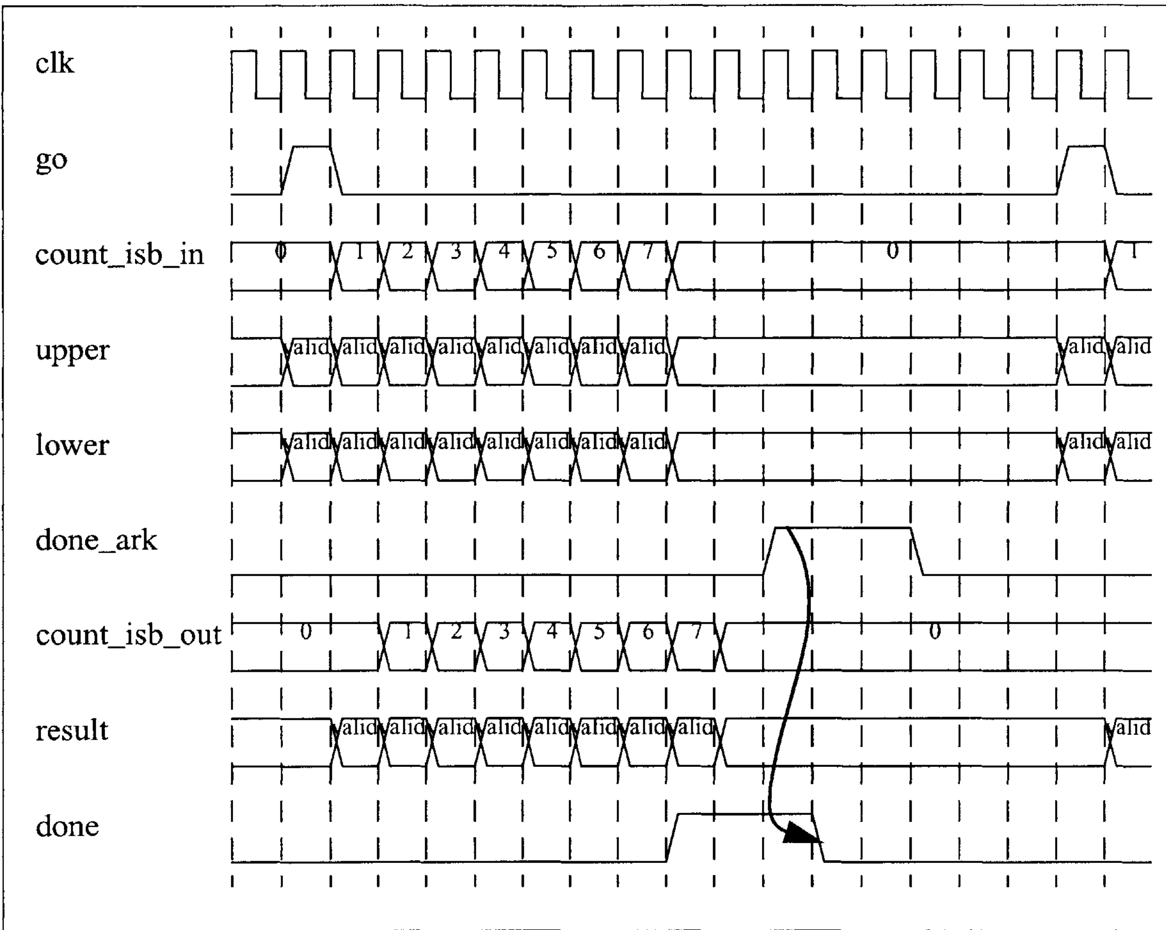


Figure 4.21 Timing diagram for the InvSubByte block.

4.3.3 Inverse Key Scheduler Implementation

In round 0, the decryption algorithm starts with the last round key. In each round the inverse key scheduler computes the previous round key. The RotWord and Rcon[(i-1)/nk] transformation is implemented once and the SubBytes transformation is replicated 4 times, once for each time it is used in the SubWord transformation. A signal diagram is found in figure 4.22 and a list of signal descriptions is found in Table 4.9.

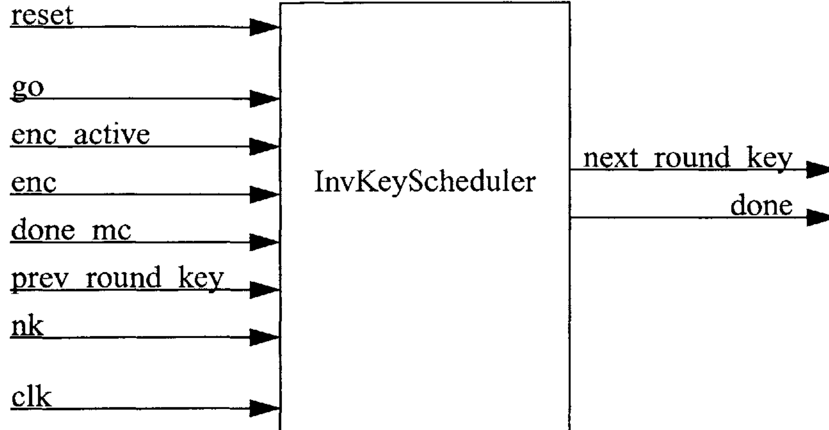


Figure 4.22 InvKeyScheduler Signal Diagram.

Table 4.9 InvKeyScheduler Signal Descriptions.

Signal Name	Signal description
generic: num_imc_blocks(3:0)	Indicates the number of times the InvMixColumns block was replicated. Set to 2.
generic: round(3:0)	Indicates the round number
generic: num_isb_blocks(4:0)	Indicates the number of times the InvSubByte block was replicated. Set to 2.
clk	System clock.
reset	System reset.
go	Indicates that valid data is on the inputs and that the Key Scheduler should begin.
enc_active	Indicates that the round is processing data.
enc	Indicates whether the transaction is an encryption or decryption
done_imc	Indicates that the MixColumns transformation is complete and that the done signal can be deasserted. (For round 14 this signal is done_sb and it indicates that the SubBytes transformation is complete.)
prev_round_key(255:0)	Holds the value of the last two round keys.
nk(3:0)	Indicates the number of words in the key.
next_round_key(255:0)	Result of the Key Scheduler. Holds the value of the current round key and the previous round key. For round 14 this signal is only 127 bits and holds the value of the current round key.
done	Indicates that the next_round_key is valid.

A timing diagram for the Inverse Key Scheduler block is found in Figure 4.23. The Inverse Key Scheduler should begin computing when the go signal is received. The last words of the round key are found in reverse order as the round keys are found in reverse order. When the round key has been found, the done signal should be asserted and held high until the done signal from the AddRoundKey block, *done_ark*, has been asserted.

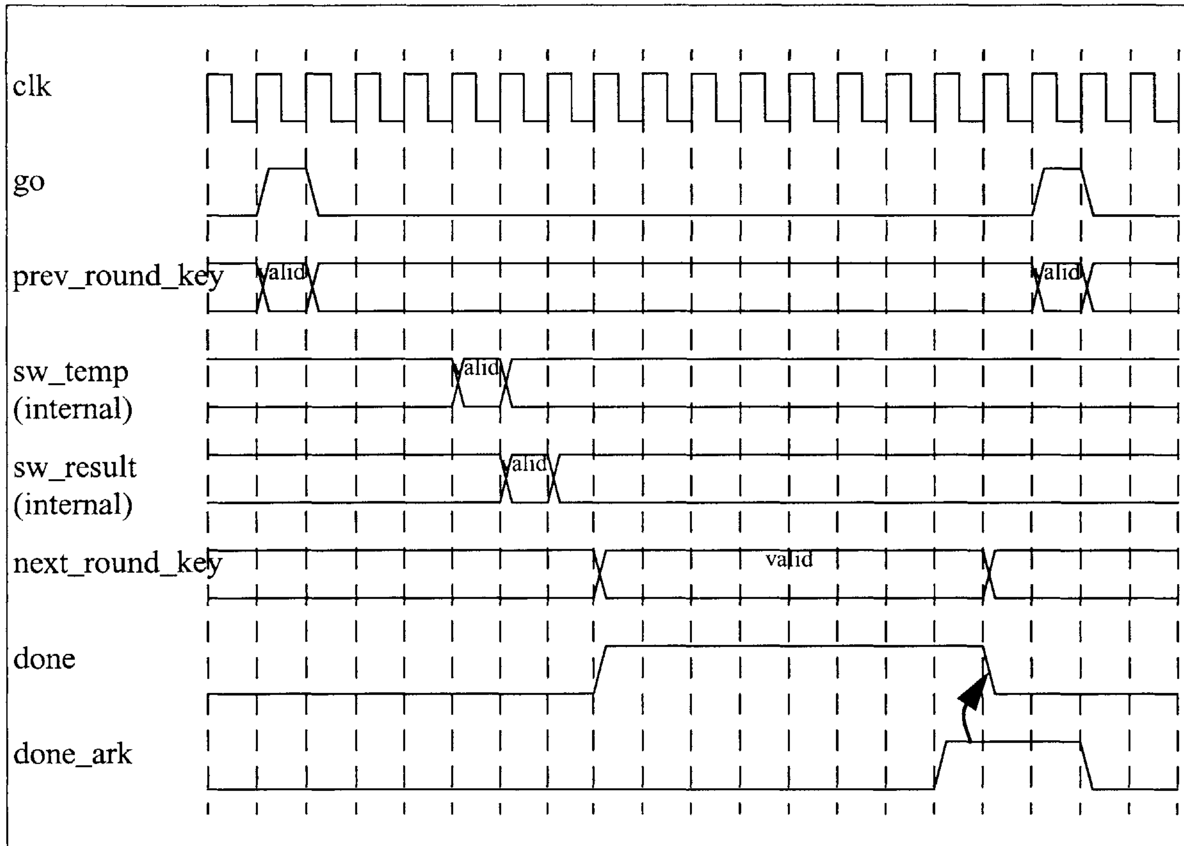


Figure 4.23 KeyScheduler Timing Diagram.

4.3.4 AddRoundKey Implementation

After the InvSubBytes transformation and the inverse key scheduler has completed, the AddRoundKey transformation is applied. The AddRoundKey transformation for decryption is the same as the AddRoundKey used for encryption, a simple xor of the round key and the state. Because the encryption and decryption units found within a round are not used at the same time, they share the logic for the AddRoundKey transformation described in section 4.2.5.

4.3.5 InvMixColumns Implementation

The InvMixColumns transformation is applied last. This is the inverse of the MixColumns transformation and is implemented with combinational logic. The transformation is replicated twice, each of which are used two times per round. The InvMixColumns needs to be replicated because it must wait until after the inverse key scheduler and the AddRoundKey transformation have completed. Figure 4.24 displays a signal diagram and Table 4.10 is a list of signal descriptions.

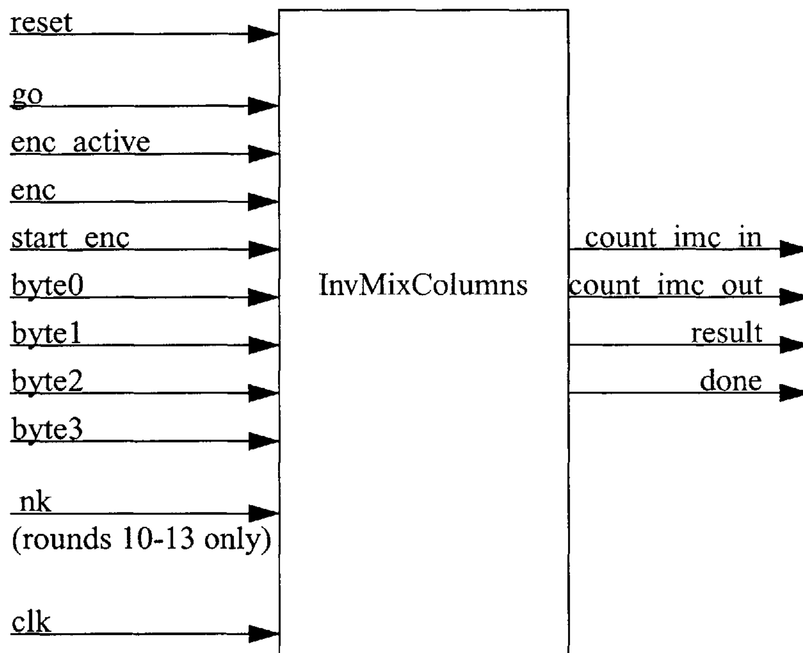


Figure 4.24 InvMixColumns Signal Diagram.

Table 4.10 InvMixColumns Signal Descriptions.

Signal Name	Signal Description
generic: num_sb_blocks(4:0)	Indicates the number of time the SubBytes block is replicated. Set to 2.
generic: num_outputs(2:0)	Indicates how many locations the result can go to. Set to 2.
clk	System clock.
reset	System reset.
go	Indicates that valid data is on the inputs and that the MixColumns transformation should begin.
enc_active	Indicates that the round is processing data.
enc	Indicates whether the transaction is an encryption or decryption
start_enc	Indicates that a new block of data is being processed.
byte0(7:0)	Bits 7:0 of the input word.
byte1(7:0)	Bits 15:8 of the input word.
byte2(7:0)	Bits 23:16 of the input word.
byte3(7:0)	Bits 31:24 of the input word.
nk(3:0)	Indicates the number of words in the key.
count_imc_in(3:0)	Indicates which word is being input.
count_imc_out(3:0)	Indicates which word is being output.
result(31:0)	Result of the MixColumns transformation.
done	Indicates that the result holds valid data.

A timing diagram for the InvMixColumns block is found in Figure 4.25. The go signal is asserted when the first word becomes valid. *Count_imc_in* indicates which word is being input. *Count_imc_out* indicates which word the result holds. When the result is invalid, *count_imc_out* holds the value *num_outputs*. When all words have been output, the done signal is asserted and held high until *start_enc* is asserted.

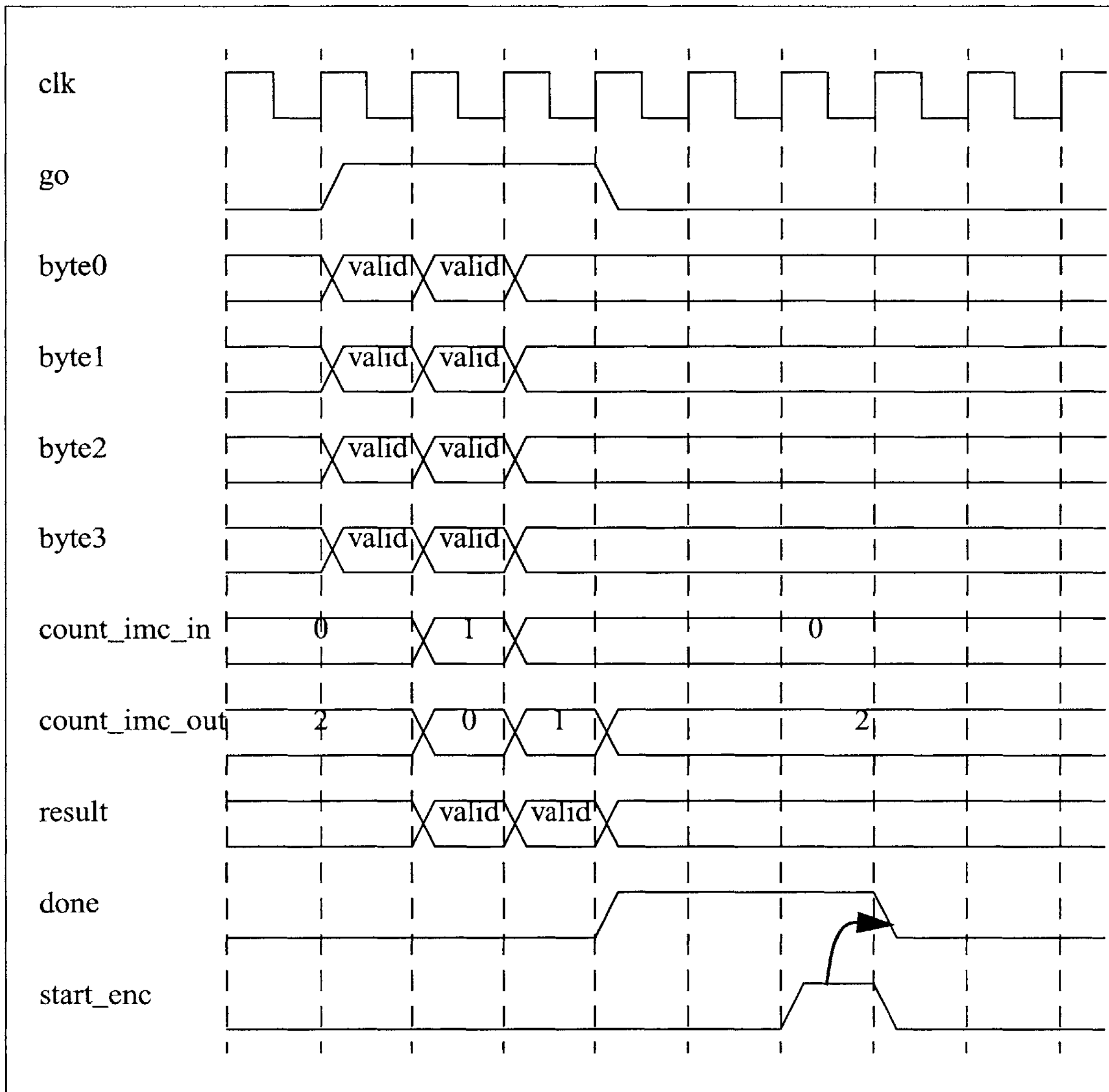


Figure 4.25 InvMixColumns Timing Diagram.

Appendix D contains a worked out example of a decryption.

4.4 Implementation Flexibility

The implementation of the encryption and decryption algorithms is flexible. In the VHDL code, each transformation uses generics to indicate how many times the block is replicated. By changing the number of times the block is replicated, the throughput of the design can be altered. For different implementations, the VHDL file for the round will need to be modified by changing the generics. To get the fastest throughput, the SubBytes block should be replicated 16 times, the MixColumns block should be replicated 4 times, and the AddRoundKey block should be replicated once. This implementation will increase the area by a factor of 4 while only raising the throughput to 1.55 Gbps. This is limited by the key scheduler and could be pushed to 5.67 Gbps with architecture changes. To get the slowest throughput, but the smallest area, each of the blocks should be replicated only once. The area will be decreased by a factor of 4. This will give a throughput of 709 Mbps.

5 IBM Core Connect Bus Interface Implementation

The AES Encryption Core interfaces with the Processor Local Bus (PLB) and the Device Control Register (DCR) Bus found in the IBM Core Connect Bus Architecture described in Chapter 3. The top level block diagram can be found in Figure 5.1. The Encryption Core functions as a PLB master and a DCR slave. As a DCR slave, the Encryption Core receives configuration and start instructions from CPU write instructions to the Encryption Core's Control Registers. Once the Encryption Core is configured, the Core gets and sends data to main memory through the PLB.

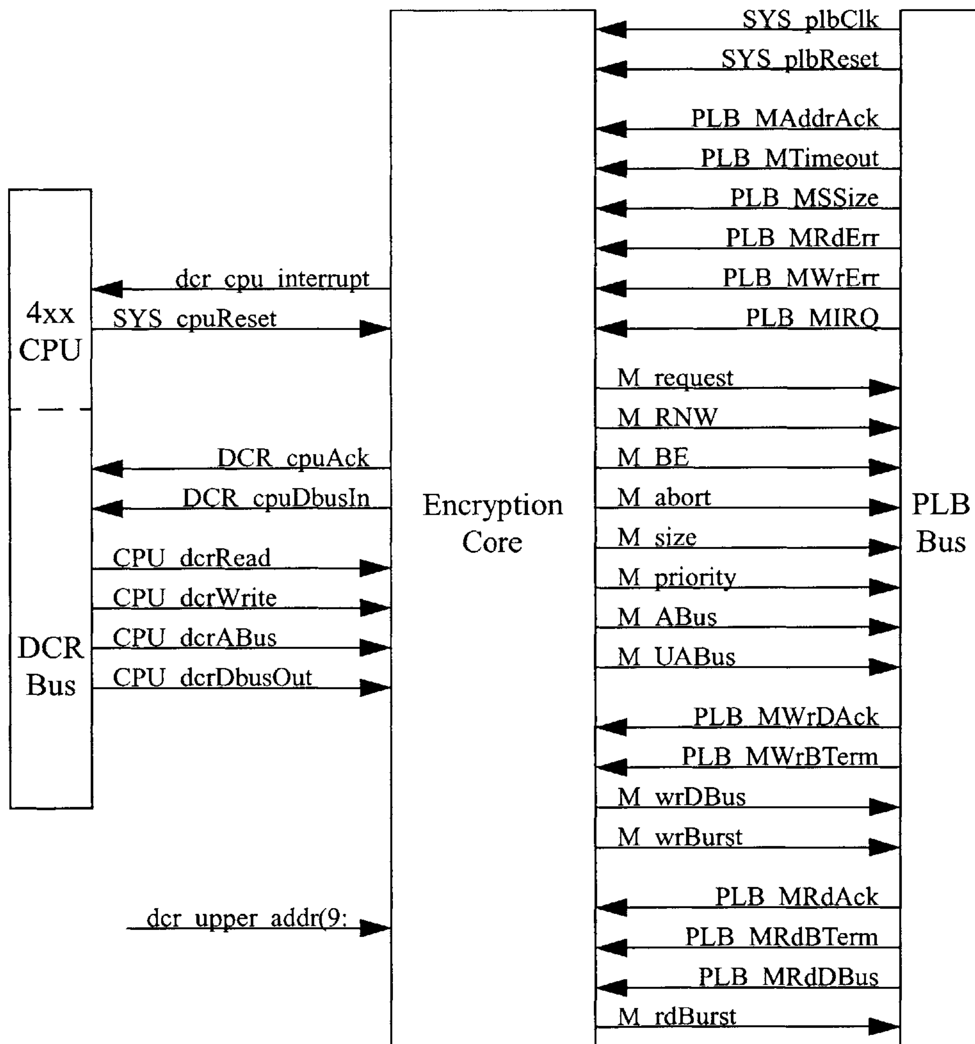


Figure 5.1 Top Level Signal Diagram.

5.1 Device Control Register Bus

A block diagram of the DCR interface is found in Figure 5.2. The AES Encryption Core uses nine registers for configuration. The upper 6 bits of the addresses are set by *dcr_upper_addr* during reset and the lower 4 bits are used to access the Encryption Cores control registers. Register 0 is the control register, registers 1 through 6 are the address registers, and registers 7 and 8 are the error registers. Table 5.1 describes the signals that are held in these 9 registers.

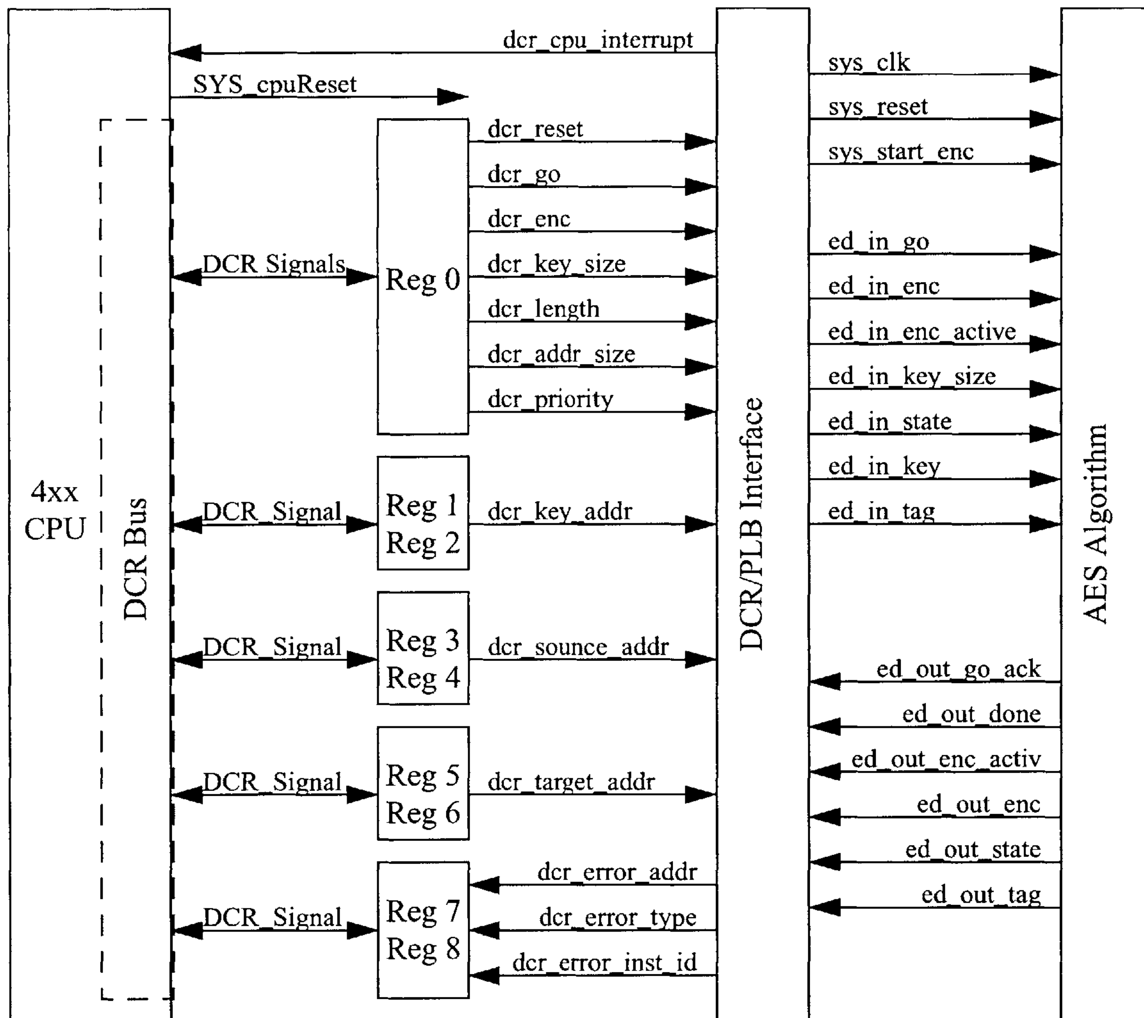


Figure 5.2 DCR Interface Signal Diagram. Not all DCR Bus logic is shown. See DCR Bus Specification.[17]

Table 5.1 DCR Bus Register contents. The Core Connect Architecture specifies that the highest order bit of a signal is zero. This is reversed in the interface to the AES Encryption Algorithm.

Signal Name	Register Number	Register Bits	Description
dcr_reset	0	31	Indicates that a transaction should stop and the pipeline should be flushed.
dcr_priority	0	29:30	Indicates what priority the transaction should have on the PLB. "00" is lowest, "11" is highest.
dcr_length	0	13:28	Indicates how many consecutive 128-bit blocks of data need to be processed.
dcr_key_size	0	11:12	Indicates what size the key is. "00"=>128, "01"=>192, "10"=>256
dcr_inst_id	0	7:10	Tags the instruction with a 4 bit ID to help decode error messages.
dcr_go	0	6	Indicates registers are ready and the Encryption Core can start a transation.
dcr_enc	0	5	Indicates whether the transaction should encrypt ('1') or decrypt ('0').
dcr_addr_size	0	4	Indicates whether the memory address is 32 or 64 bits.
dcr_key_addr (31:0)	1	0:31	Lower 31 bits of the PLB key address.
dcr_key_addr (63:32)	2	0:31	Upper 31 bits of the PLB key address.
dcr_source_addr (31:0)	3	0:31	Lower 31 bits of the first PLB source address.
dcr_sounce_addr (63:32)	4	0:31	Upper 31 bits of the first PLB source address.
dcr_target_addr (31:0)	5	0:31	Lower 31 bits of the first PLB target address.
dcr_target_addr (63:32)	6	0:31	Upper 31 bits of the first PLB target address
dcr_error_type	7	29:31	Indicates if there is an error and what type of error occurred.
dcr_error_inst_id	7	25:28	Indicates the instruction ID of the transaction that was being processed when and error occurred.
dcr_error_addr (31:0)	7	0:24	Lower bits of the PLB error address. Lowest seven bits are assumed to be zero because an address must be quad-word aligned.
dcr_error_addr (63:32)	8	0:31	Upper 31 bits of the PLB error address.

To configure a transaction, the flow chart found in Figure 5.3 should be used. If 64-bit PLB addressing is used, all registers are needed. However, if 32-bit addressing is used, the upper address registers are not needed. The control register should be written last because this register holds the go signal, *dcr_go*, which indicates when all of the other registers are ready and a transaction can begin. Because the registers are latched into the Encryption Core on the rising edge of the go signal, the CPU can prepare the next transaction by setting up all of the address registers for the next transaction before the interrupt is received. Once the go signal is set high, a new transaction will start. After the results have been written back into memory through the PLB, an interrupt signal is sent to the CPU. At this time the CPU should read the error register to see why the interrupt was sent. If the error register indicates that no error occurred, then the CPU should reset the go signal. After reseting the go signal the CPU can send another transaction request. If the go signal is deasserted and reasserted before the interrupt is received, the transaction will be ignored. This is described in the timing diagram in Figure 5.4.

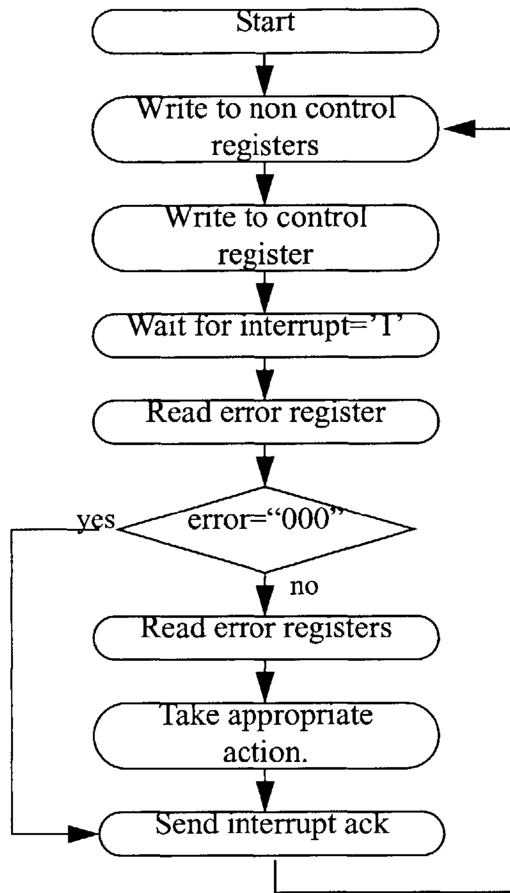


Figure 5.3 CPU Operation Flow Chart.

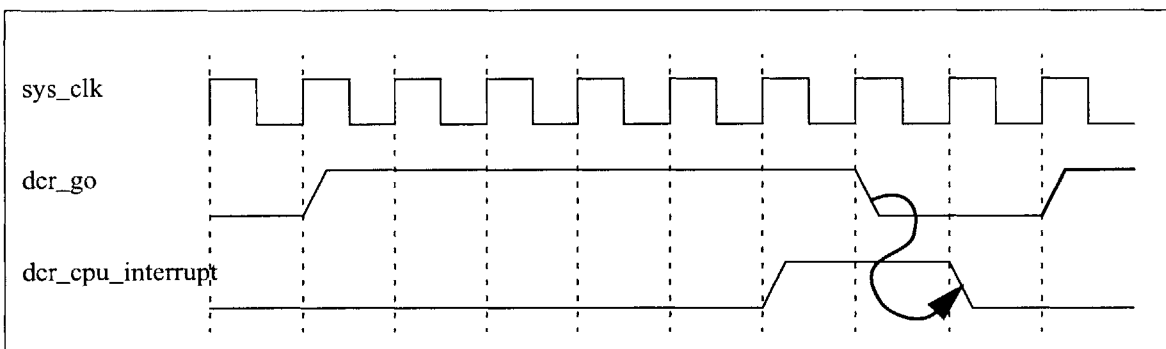


Figure 5.4 Timing Diagram for Starting and Ending a Transaction. Not to scale.

If the CPU finds that an error has occurred, then the CPU should take steps to recover from the error. The different types of errors are listed in Table 5.2. The error registers hold the upper 57 bits of the address of the last attempted PLB write. The lower bits are assumed to be zero because all addresses for the Encryption Core must be quad-word aligned. The error registers also contain the error type and the instruction ID that the error occurred on.

Table 5.2 Error Types.

Error Bits	Error Type
000	no error
001	timeout error
010	read error
011	write error
100	MIRQ
101	slave wrong size
110	dcr reset error
111	unused

5.2 Processor Local Bus

The PLB is used to read and write data in main memory. Figures 5.5, 5.6, and 5.7 are flow charts for the PLB interface operation. After the Encryption Core is configured through the DCR Bus, the Encryption Core uses the PLB to fetch the key. Once the key has been fetched, the Core will use the source address to get data to encrypt. If the number of blocks to be encrypted is greater than one, then the Encryption Core issues a burst read on the bus and will store the data into a buffer. There are two implementations of the PLB, one with a buffer size of 16 and one with a buffer size of 32. Each implementation has two buffers. Initially, one buffer will be filled as much as possible. For example, if the number of blocks is less than the buffer length, the buffer will not be filled completely.

The data is processed using a ping-pong buffer scheme. After the buffer is filled with data from the PLB, the data will be sent out to be processed. As the data is processed, the data in the buffer is overwritten. At the same time the data from the first buffer is being processed, if there is more data that needs to be processed, data is requested over the PLB and stored in the empty buffer. Once all of the data has been overwritten, then the data is sent back out to memory through the PLB. After the first buffer has sent all of its data into the pipeline, the second buffer starts sending data into the pipeline. This allows the first buffer to send processed data to the PLB, then be refilled with new data, all while the second buffer is sending data to be processed.

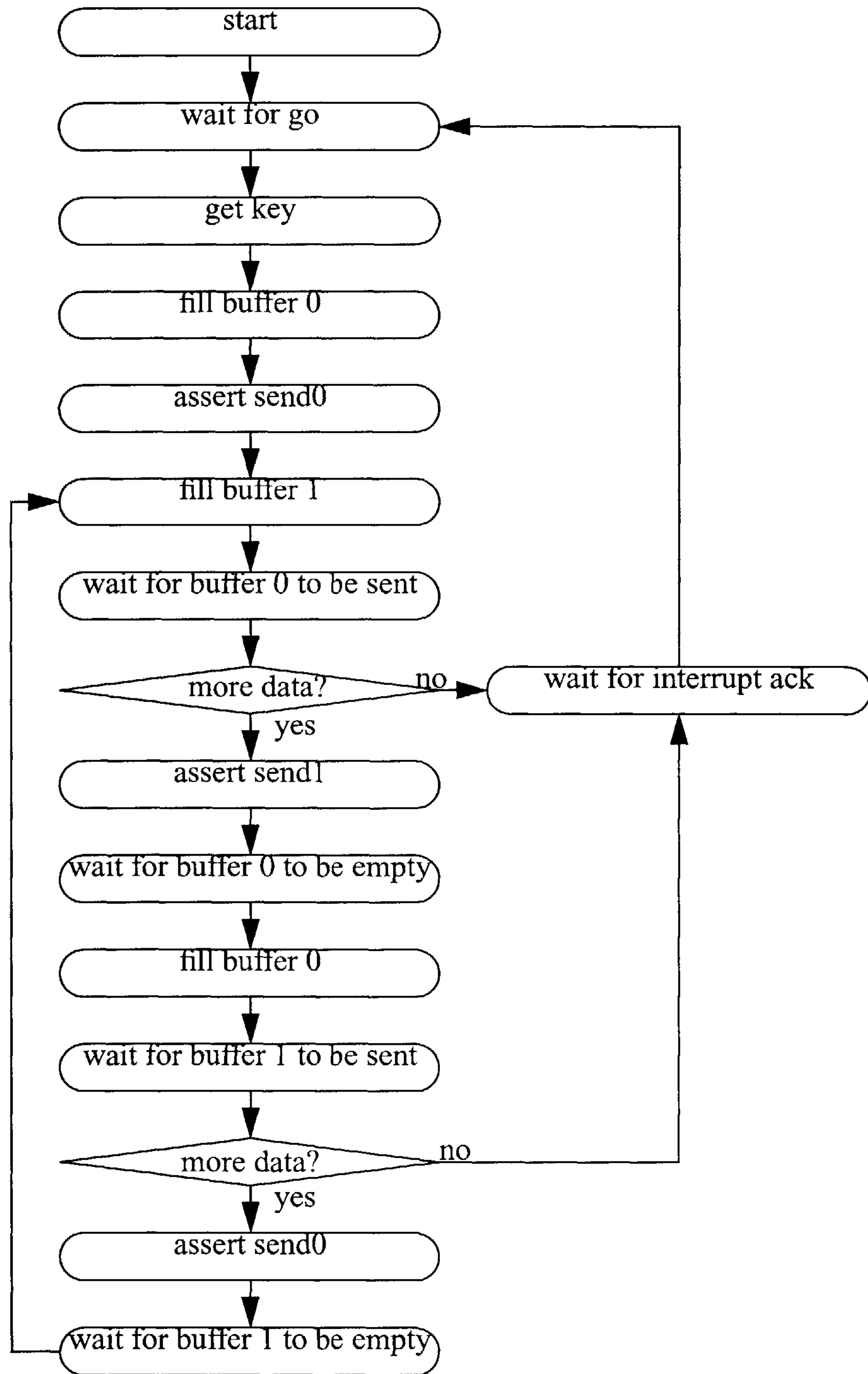


Figure 5.5 Fill Buffer Flow Chart.

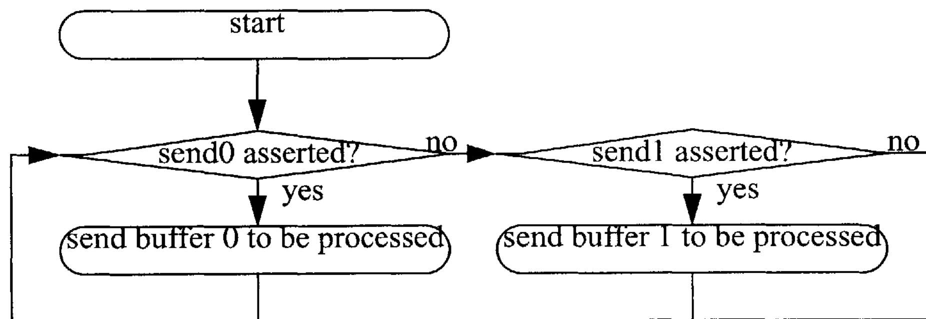


Figure 5.6 Send Data Flow Chart.

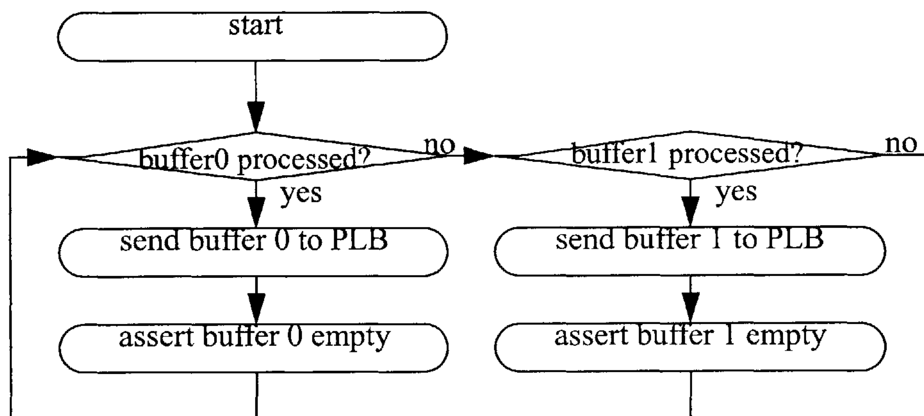


Figure 5.7 Empty Buffers Flow Chart.

When reading data, the interface issues burst transactions if possible. This reduces the time to read and write data because only one address acknowledgement is needed for multiple blocks of data. On average, the address acknowledgement and each subsequent data acknowledgement takes 7.5 clock cycles. If each block of data has 16 cycles

allocated to it, then using Equation 5.1, the buffer size should be a minimum of 15. There is one read and one write per block, so the average acknowledgement time needs to be multiplied by 2. It should be noted that the average acknowledge time could be increased so that the a 1Gbps throughput will not be achievable. If this is not acceptable, a different interface should be implemented. The buffer size should not become larger than 32 to limit PLB bus utilization by the AES core. Because there are other devices on the PLB, the bus should not be monopolized for long periods of time.

$$\frac{2 \cdot avgacktime + 2 \cdot avgacktime \cdot buffersize}{blockallocatetime \cdot buffersize} < 1 \qquad \textbf{(Equation 5.1)}$$

The PLB operations are shown in Figures 5.8 through 5.14. All of the read transactions are either single or burst reads or writes. The master ends the burst requests. However, if the slave should end a burst read or write early, the Encryption Core reissues the read or write request and continues from where the slave stopped the operation.

The Encryption Core assumes that the key for a transaction is stored in a secure location somewhere in memory. The DCR Bus provides the address of the key during configuration. When the DCR bus indicates a transaction should start, the Encryption Core uses the PLB to fetch the key. Figure 5.8 shows how 128-bit keys are fetched. Figure 5.9 shows how 192- or 256-bit keys are fetched. After the key has been fetched, the go acknowledge, *ed_out_go_ack*, will be set high.

After the go acknowledge is set high, data to be encrypted or decrypted will need to be fetched. Figure 5.10 shows how a single block of data will be fetched and figure 5.11 shows how multiple blocks of data will be fetched. Figure 5.12 shows what will happen if the number of blocks that need to be encrypted or decrypted is greater than the buffer size.

Once all of the data has been processed, the results will be written back out to memory. Figure 5.13 shows how a single block of data is sent to memory and Figure 5.14 illustrates how multiple blocks of data are sent to memory.

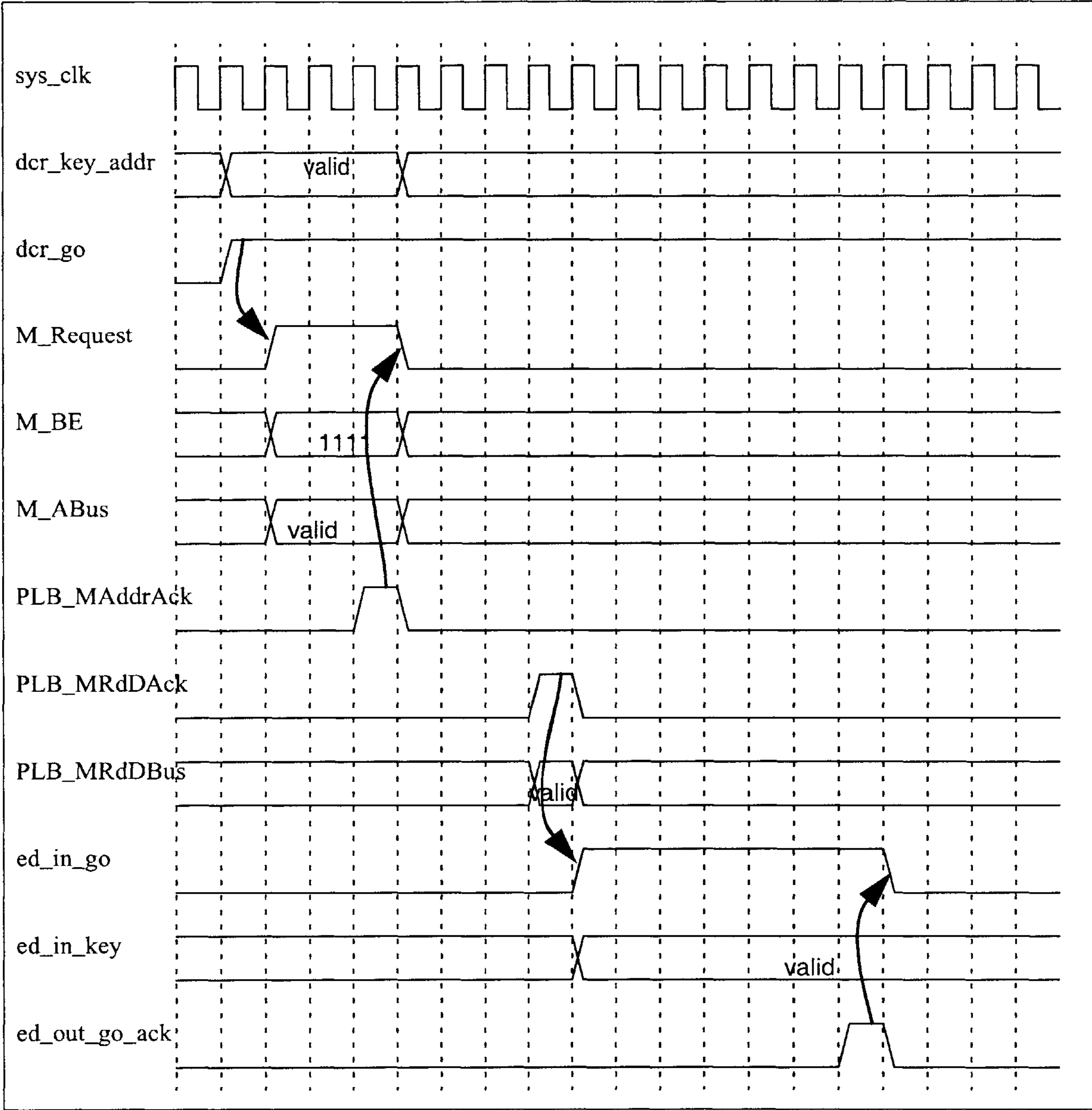


Figure 5.8 PLB Interface Timing Diagram. Get Key, key size=128. M_RNW=1, M_RdBurst=0.

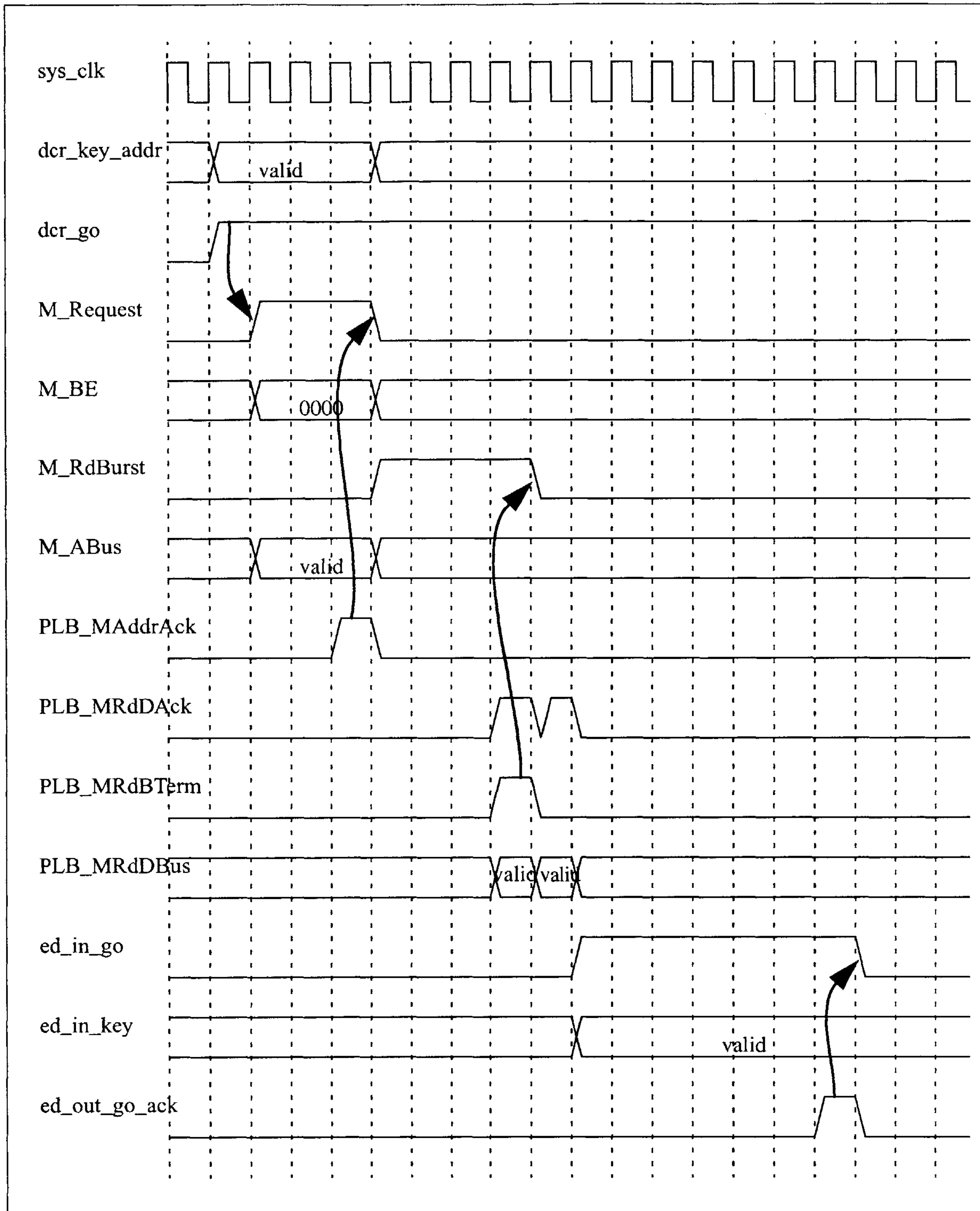


Figure 5.9 PLB Interface Timing Diagram. Get Key, key size/=128. M_RNW=1.

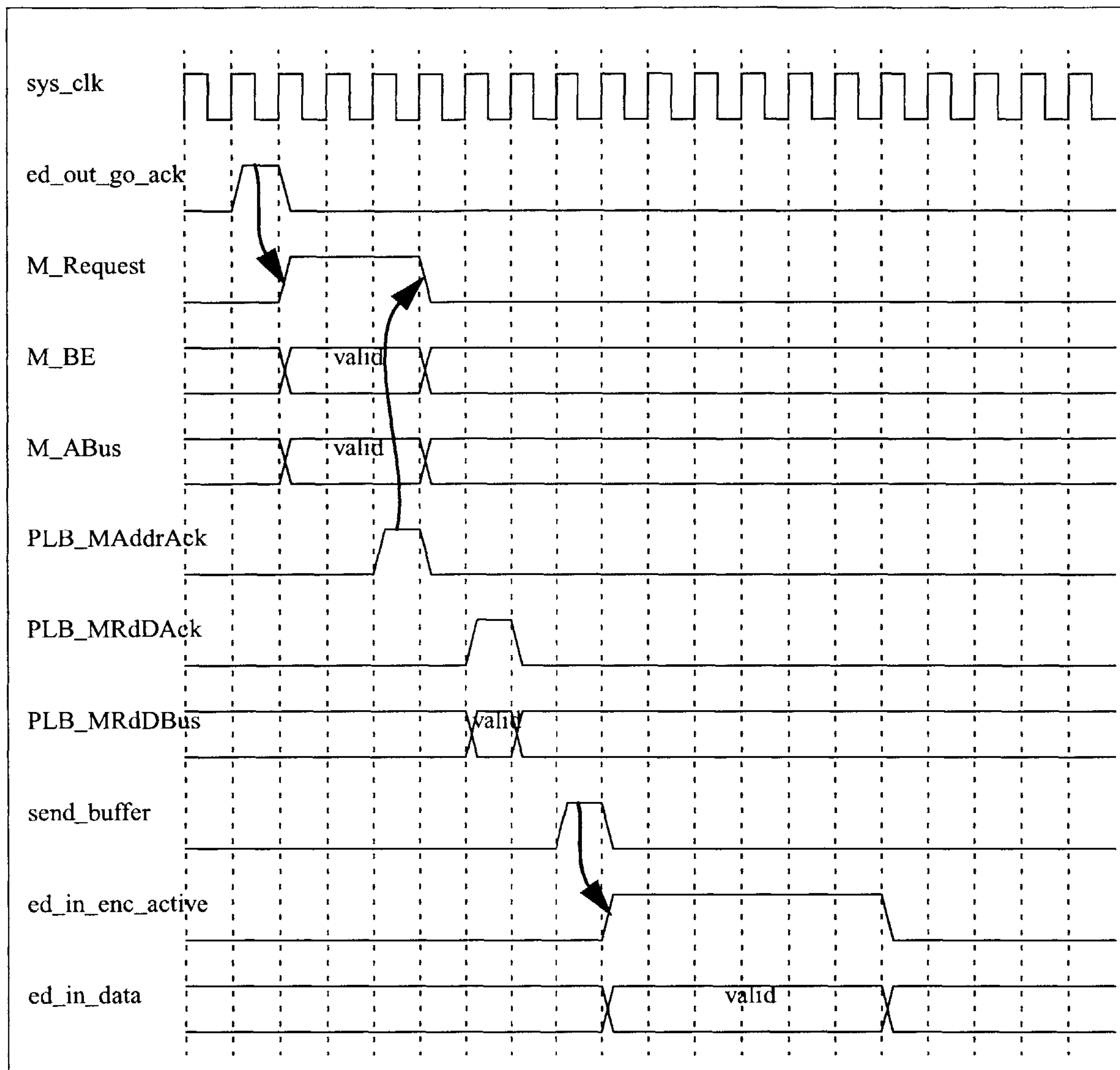


Figure 5.10 PLB Interface Timing Diagram. Get Data, Data=1. M_RNW=1, M_RdBurst=0. (Not to scale, ed_in_enc_active and ed_in_data should be valid for 16 clock cycles.)

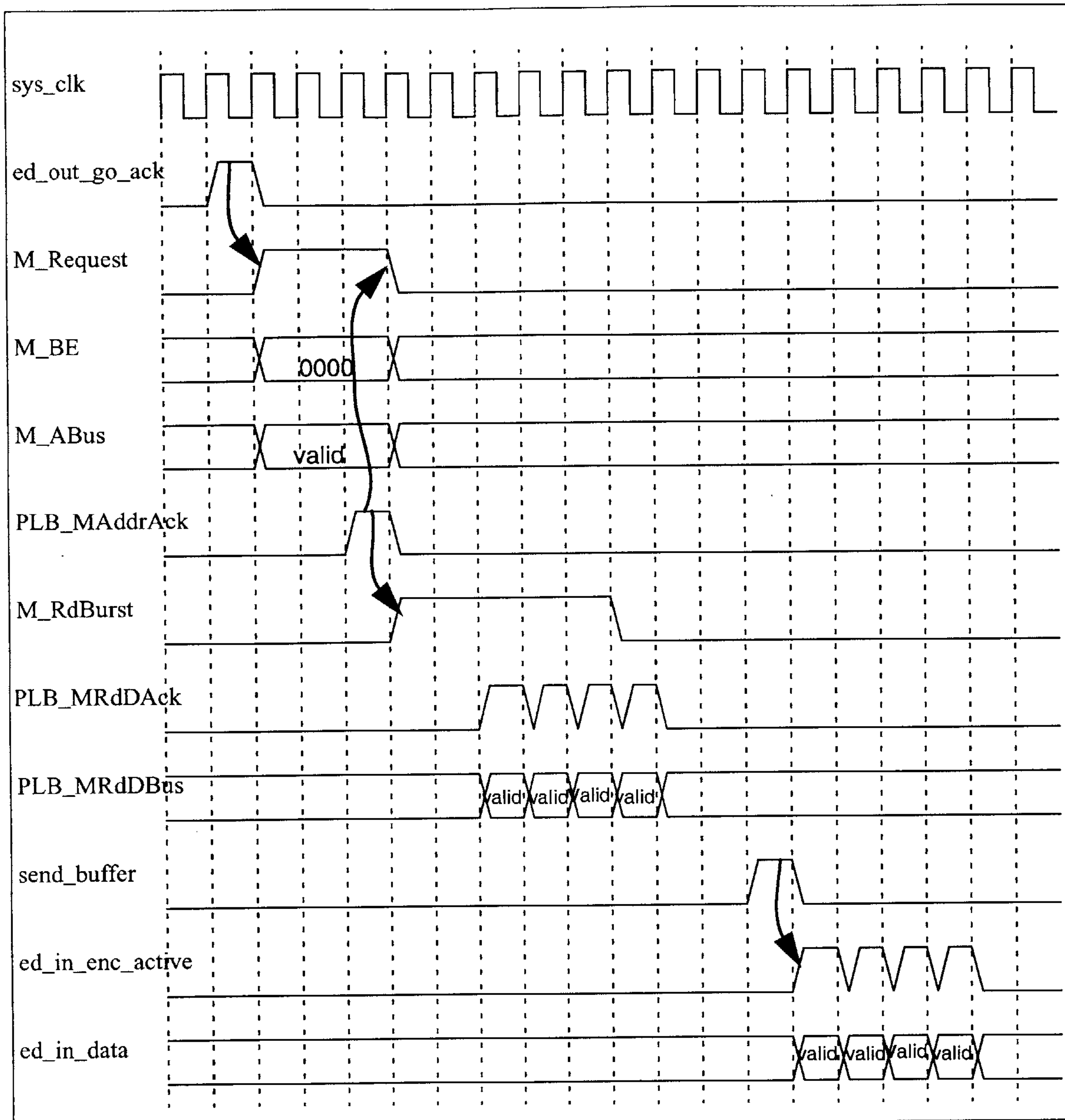


Figure 5.11 PLB Interface Timing Diagram. Get Data, 1<Data<=16. In this example, Data=4. M_RNW=1. (Not to scale, for example, ed_in_enc_active and ed_in_data should be valid for 16 clock cycles.)

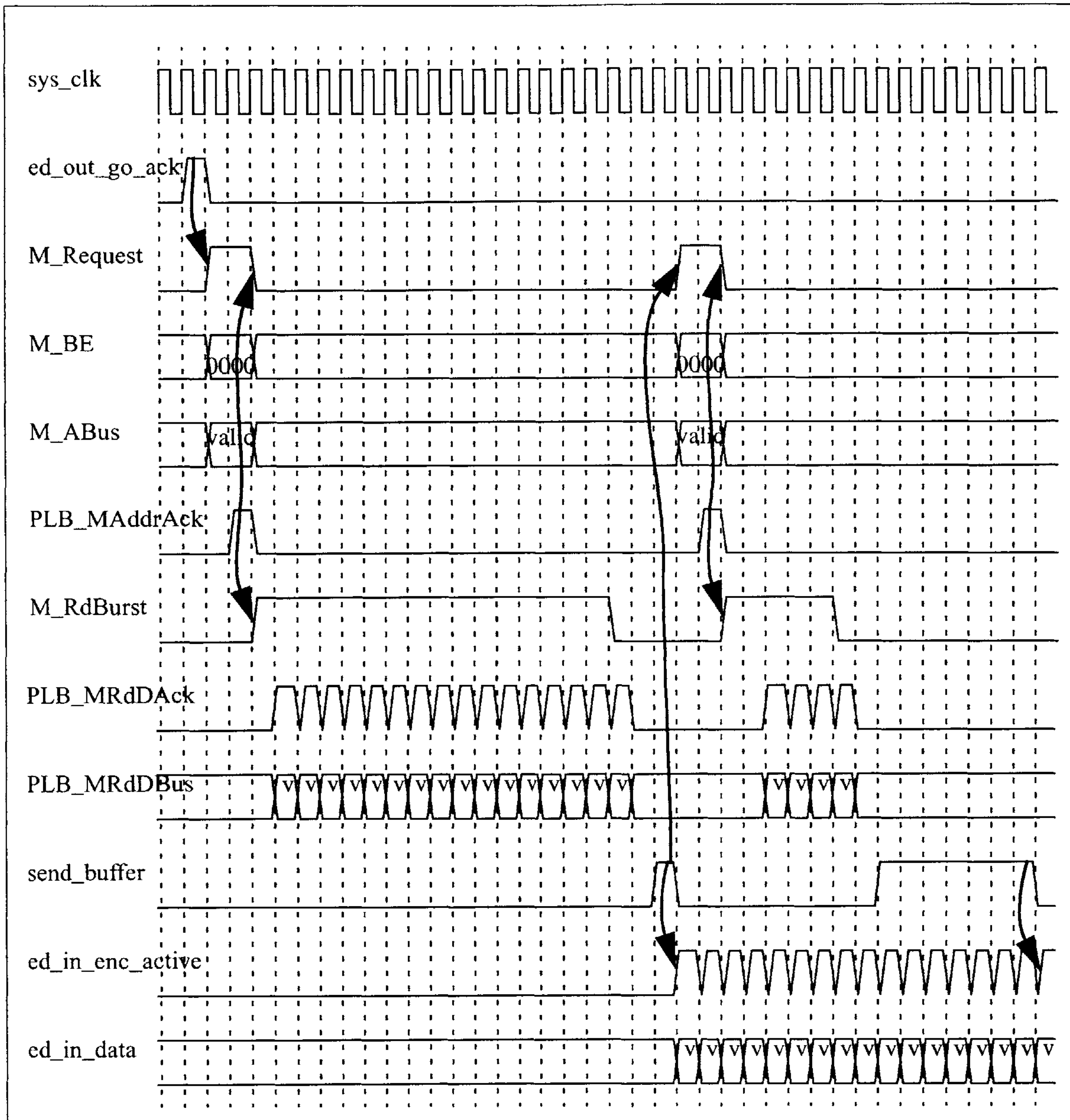


Figure 5.12 PLB Interface Timing Diagram. Get Data, Data>16. In this example, Data=20. M_RNW=1. This example ends like the timing diagram for data<=16. (Not to scale, for example, ed_in_enc_active and ed_in_data should be valid for 16 clock cycles.)

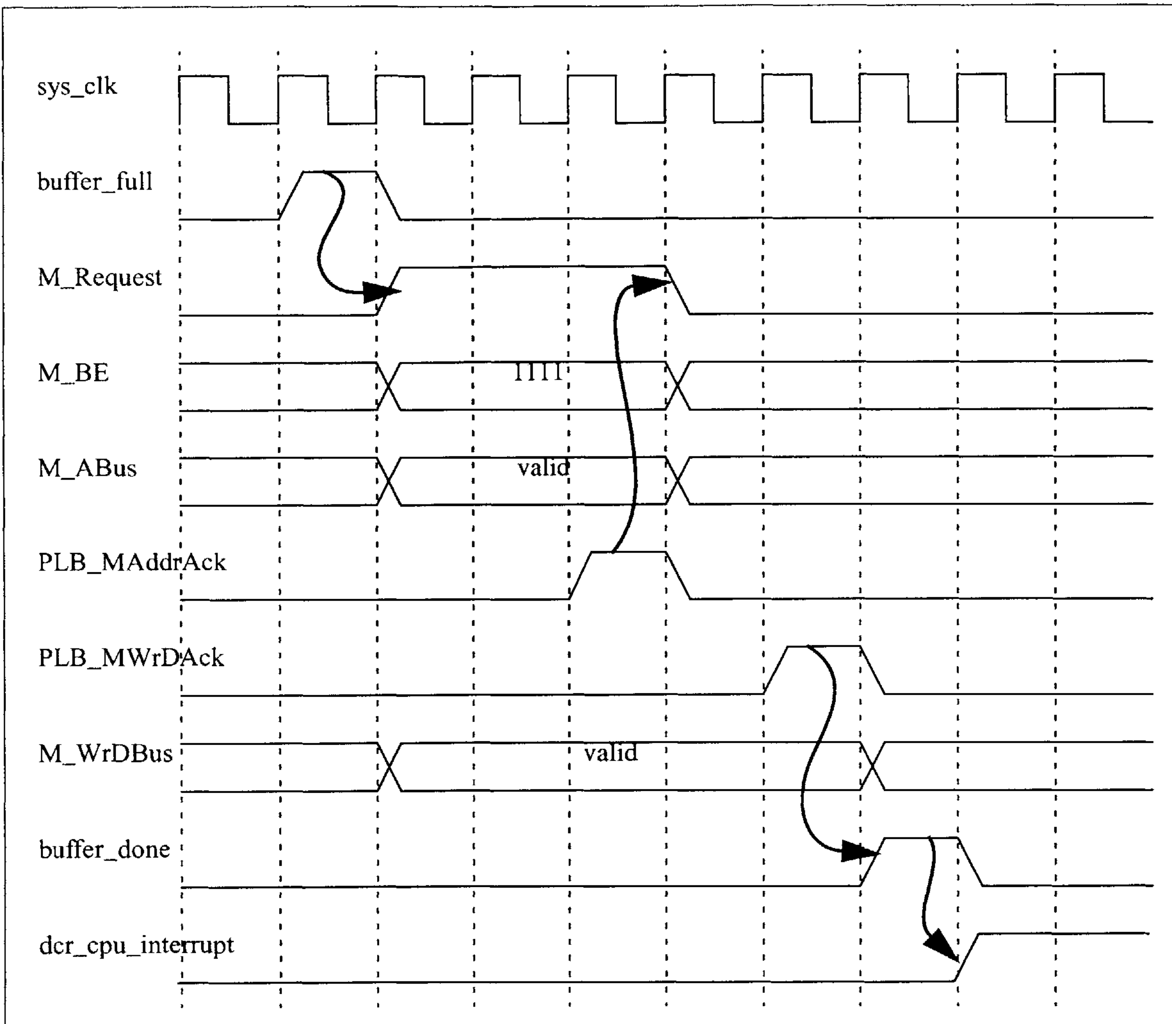


Figure 5.13 PLB Interface Timing Diagram. Send Data, Data=1. In this example, Data=4. M_RNW=0. (Not to scale.)

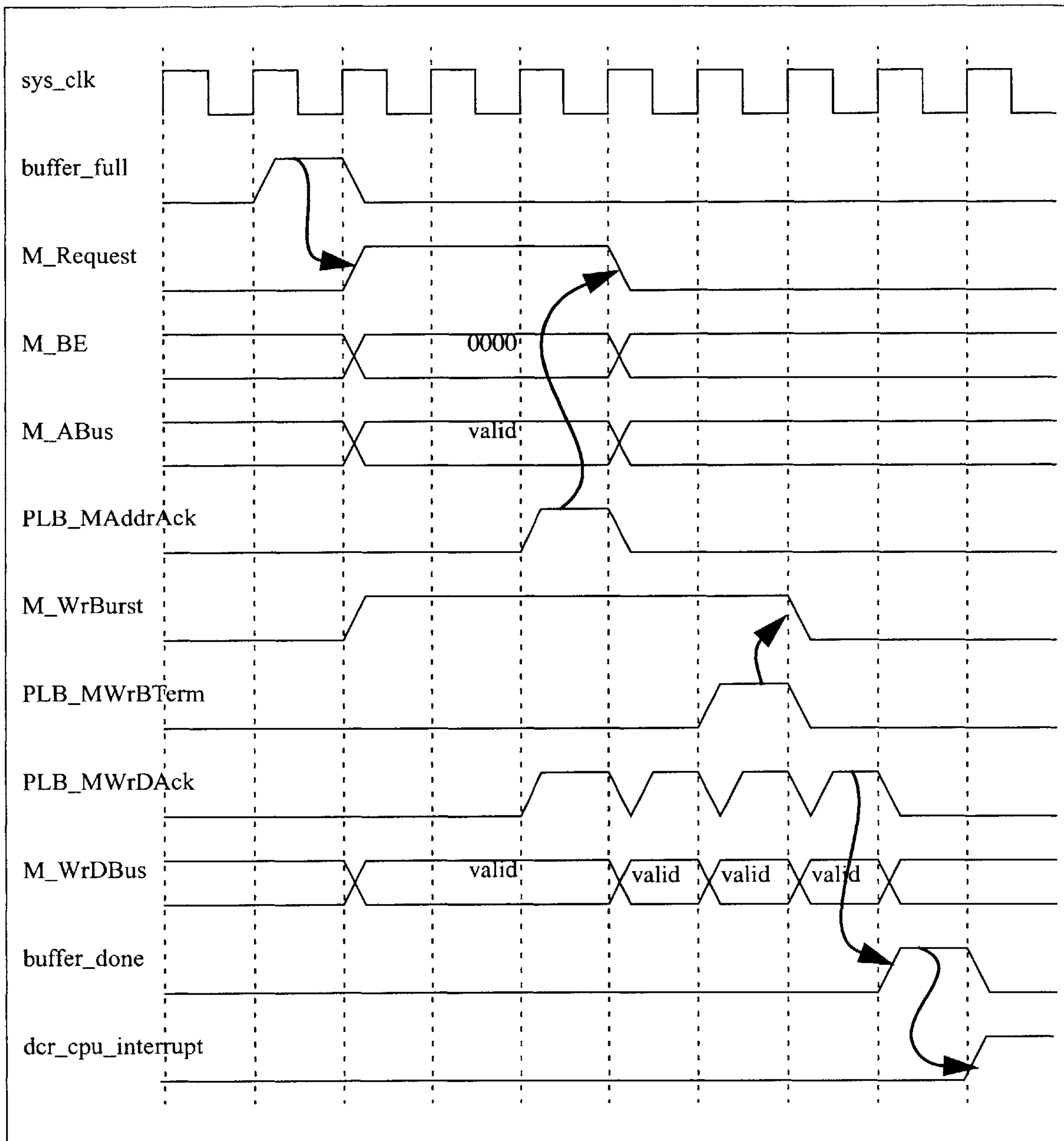


Figure 5.14 PLB Interface Timing Diagram. Send Data, Data>1. In this example, Data=4. M_RNW=0. (Not to scale.)

6 Verification

Verification of the encryption algorithm and the bus operations were completed using a VHDL testbench that connects to the DCR and PLB interfaces of the AES Encryption Core. The testbench checks for PLB and DCR bus violations and ensures that data is encrypted and decrypted correctly. The monitors in IBM's PLB and DCR Functional Model Toolkits are used to check for bus violations. To ensure that the AES algorithm implementation functions correctly, a set of known values were input to the system to be encrypted or decrypted. These values can be found on the AES web page.

The AES Encryption Core connects to the PLB and DCR Toolkits as shown in Figure 6.1. The toolkits provide support to implement other slaves and masters on the buses. The Encryption Core testbench supports inclusion of up to 8 PLB slaves, 7 PLB masters (not including the AES Encryption Core), and 1 DCR master. In the existing tests, the PLB slaves function as memory devices and the PLB masters add activity to the PLB. The DCR master reads and writes to the control registers, sending instructions to the Encryption Core.

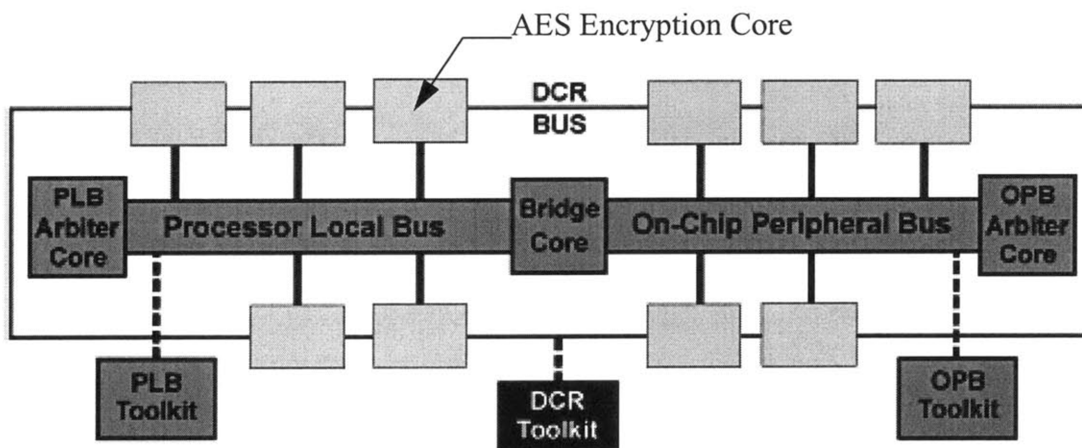


Figure 6.1 Toolkit Block Diagram. [18]

The toolkits provide a VHDL file that serves as a model for the top level testbench. One of the PLB master implementations and the DCR master implementation was replaced with the AES Encryption Core. One of the synchronization signals was designated as a "test complete" signal. The *dcr_upper_addr* signal was set to all zeros. Logic was added to handle interrupts and to instruct the DCR master to issue the next instruction in the queue. When all of the transactions are complete, the DCR master sets the "test complete" signal indicating that the PLB slaves should check their memory contents. If the memory contents contain incorrect data, error messages will be output. A message will also be output indicating when the test is complete.

The toolkits provide a language called the "Bus Functional Language" to write models for masters and slaves. The PLB slaves are used as memory devices. One slave holds the keys for all of the transactions, one holds the encrypted data, one holds the decrypted data, and the last is used to store all of the data that the transactions produce. The last slave checks all of its memory contents when all of the transactions are complete to verify that the correct data is there. The slaves are configured so that they automatically respond to a read or write request, ending the operation with one of the legal responses. The legal responses include timeouts, read errors, write errors, interrupt requests, or regular operation. A monitor on the bus checks that all bus specifications are adhered to. If any specifications are violated, a message is printed to the screen.

The AES web page provides Known Answer Tests (KAT's) which are a set of values that can be used to check that the encryption and decryption operations are working properly. For 128-bit keys the KAT's have 128 sets of values for variable key, single data block encryption; 192 sets for 192-bit keys; 256 sets for 256-bit keys. There are 128 sets of values for single key, variable data block encryption for each of the different key sizes. There are also a more extensive set of values, most of which were not tested due to time constraints. Because the tests require the Encryption Core to run in the most non-optimal

fashion, the estimated time to run the additional tests on the RTL using a single processor is 100 days. Running the tests on the gate level would take five to ten times longer.

Modeltech's MTI version 5.4e was used for simulation. The tests did encryptions and decryptions on transactions of length 1 to 128. They checked normal operation, read errors, write errors, timeouts, and interrupt requests. The tests are summarized in Table 6.1.

Table 6.1 Verification Tests.

Test Name	Test Description
ecb_burst_dec	Tests decryption transactions of varying lengths. Compares results against known answers.
ecb_burst_enc	Tests encryption transactions of varying lengths. Compares results against known answers.
ecb_vk_dec	Tests decryptions of length one. Compares results against known answers.
ecb_vk_enc	Tests encryptions of length one. Compares results against known answers.
ecb_vt_dec	Tests decryptions of length 128. Compares results against known answers.
ecb_vt_enc	Tests encryptions of length 128. Compares results against known answers.
test_mirq	Tests to ensure that interrupt requests from the PLB are handled properly.
test_read_error	Tests to ensure that read errors from the PLB are handled properly.
test_reset	Tests to ensure that all resets are handled properly.
test_slave_wrong_size	Tests to ensure that responses from PLB slaves of incorrect size are handled properly.
test_timeout	Tests to ensure timeout responses are handled properly.
test_write_error	Tests to ensure that write errors from the PLB are handled properly.

7 Synthesis

Synthesis was performed using the Synopsys Design Compiler tool. The AES Encryption Core was designed in VHDL and mapped to gate level code using logic elements from IBM's SA27E ASIC technology library.

Scripts were written in dcshell to load the VHDL and timing assertions, map the RTL to gates optimizing for area, and write out the design in db and in verilog formats. Reports are generated to produce area and timing information.

The Synopsys Design Compiler and the static timing tool, Einstimer, use different models for timing. To force the models to behave similarly and to meet IBM's softcore timing requirements, a 7.0 ns clock was used for the Synopsys model and a 7.519 ns (133 MHz) clock was used for the Einstimer model.

The 5.3 mm wire load model was used. The operating conditions were set to worst case conditions of 1.65 Volts and 100 degrees Celsius and the maximum fanout was set to 20. Certain gates were specified to not be used for various reasons such as the large cell size or incompatibility with other gates.

The clock inputs were set to ideal and the tool was prevented from repowering the clock trees. For a softcore the customer is required to handle the clock trees. The inputs were given 50% of the clock period to arrive and the output were given 25% of the clock cycle to arrive, leaving the remainder of the clock period to be used by external logic. While this does not match the assertions used for Einstimer, it served as a reasonable approximation.

The VHDL was compiled hierarchically as the design is larger than is recommended for a top down compilation. First, each of the rounds were compiled and optimized for area. The results were written out in db format. Next, the top level for the encryption algorithm was compiled. The round db files were used as inputs, the design was flattened and once again optimized for area. The results were written out in db and

Verilog format. The Verilog was written out because it is one of the formats Einstimer can read. Next, the interface was compiled. After the interface was optimized, the entire design was flattened and optimized again. Results were written out in db and verilog format.

The synthesis results are listed in Table 7.1. The design name indicates the point at which the design was flattened and compiled. The table assumes the conversion: 1cell = 2.96 gates.

Table 7.1 Synthesis Results.

Design name	Cells	Gates	Area (um^2)
AES Encryption Core Buffer size 16	238,547	706,099	2,354,937
AES Encryption Core Buffer size 32	295,429	874,469	2,722,302
AES Algorithm	171,903	508,832	1,811,873
Round 0	3,769	11,156	50,131
Round 1	11,794	34,910	121,766
Round 2	11,827	35,007	124,715
Round 3	11,750	34,780	124,349
Round 4	11,527	34,119	123,561
Round 5	11,763	34,818	124,799
Round 6	11,629	34,421	123,717
Round 7	11,687	34,593	124,512
Round 8	11,792	34,904	124,930
Round 9	11,760	34,809	124,387
Round 10	12,623	37,364	128,115
Round 11	12,680	37,532	129,110
Round 12	11,721	34,694	122,137
Round 13	11,715	34,676	122,016
Round 14	8,646	25,592	88,340

8 Static Timing

Einstimer, an internal IBM tool, was used to evaluate the static timing. The gate level netlists were read into the tool in Verilog format. Timing assertions that met IBM's SoftCore guidelines were then provided for the design. Tests to check for setup time and hold time violations were run on the following paths: inputs to output, input to register, register to output, and register to register. Early and late mode for both the best case and worst case conditions were run. Early mode tests check for hold time violations by tracing each of the shortest paths to ensure that the actual arrival time was greater than the required arrival time. Late mode tests check for setup time violations by tracing each of the longest paths to ensure that the required arrival time is greater than the actual arrival time. Einstimer was also used to ensure that there were no electrical violations. One example is checking to make sure the load capacitances are in the correct range.

Scripts were written in Tool Command Language (TCL) to provide the timing assertions. The worst case temperature is set to 125 degrees Celsius and the best case is set to -40 degrees Celsius. The worst case voltage is set to 1.65 volts and the best case is set to 1.95 volts.

A phase file provides information about the clock. The clock period is 7.519 ns (133 MHz) with a 50% duty cycle. The late mode clock slew is set to 0.3 ns and the early mode clock slew is set to 0.05 ns.

Another file provides information on the slews for the input signals. The late mode slew is set to 0.7 ns and the early mode slew for non-clock inputs is set to 0.02 ns. The early mode slew for the clock input is set to 0.05 ns. The maximum arrival times are set to the three cycle timing guidelines listed in the specifications for the PLB and DCR busses. The maximum arrival times for all other inputs are set to 50% of the clock cycle, shown in Table 8.1. The minimum arrival times are set to 0.5 ns.

Table 8.1 Required Arrival Times for Inputs.

Signal Name	Required Arrival Time (% of clock cycle from rising edge of SYS_PLBCLK)
SYS_PLBRESET	50%
SYS_CPURESET	50%
CPU_DCRREAD	18%
CPU_DCRWRITE	18%
CPU_DCRABUS	18%
CPU_DCRDBUSOUT	18%
DCR_UPPER_ADDR	50%
PLB_MADDRACK	50%
PLB_MTIMEOUT	15%
PLB_MBUSY	30%
PLB_MRDERR	30%
PLB_MWRERR	30%
PLB_MIRQ	30%
PLB_WRDACK	50%
PLB_MWRBTERM	50%
PLB_MRDDACK	50%
PLB_MRDBTERM	50%
PLB_MSSIZE	50%
PLB_MRDDBUS	50%

A file for the output signals specifies that the maximum arrival times meet the three cycle timing guidelines listed in the specifications for the PLB and DCR buses. All maximum arrival times for the other outputs are set to 30% of the clock cycle, shown in Table 8.2. The minimum arrival times are set to 0.0 ns.

Table 8.2 Required Arrival Times for Outputs.

Signal Name	Required Arrival Time (% of clock cycle from rising edge of SYS_PLBCLK)
DCR_CPU_INTERRUPT	30%
DCR_CPUACK	68%
DCR_CPUDBUSIN	68%
M_REQUEST	15%
M_RNW	15%
M_ABORT	15%
M_WRBURST	15%
M_RDBURST	15%
M_PRIORITY	15%
M_BE	15%
M_SIZE	15%
M_ABUS	15%
M_WRDBUS	15%

One file was created to specify maximum capacitances and driving resistances for the inputs. The maximum capacitance for the clock input is 9999.0 pf, which functions as an infinite capacitance. The driving resistance is set to 0.0 ohms. The capacitances on the Core Connect inputs are 0.1 pf with a driving resistance of 1.0 K ohms. The capacitances on other inputs are 0.2 pf with a driving resistance of 1.0 K ohms.

Another file was created to specify the minimum and maximum capacitances for the outputs. The maximum capacitances on the Core Connect outputs are 0.20 pf and the minimum capacitances are 0.01 pf. The other signals had a maximum capacitance of 0.30 pf and a minimum capacitance of 0.01 pf.

These assertions were applied to both the 16 block and the 32 block implementations of the AES algorithm. The first time there were setup violations. These

violations were fixed by setting the clock to a higher frequency during synthesis. This forced the Synopsys Design Compiler to work harder and reduce the number of gates in the longer paths. Reducing the clock to 7.0 ns had the desired effect of eliminating the setup violations in Einstimer. Another problem that occurred is that the maximum load capacitances were exceeded. To fix this problem the power level on some of the registers were adjusted. This eliminated all of the errors produced by Einstimer.

9 Future Work

There are several enhancements that can be made at all levels for this system. Several architectural changes could significantly reduce the area or improve the latency for a transaction. First, the GetDecKey block and the key scheduler implementation should be investigated. The GetDecKey block could be modified to find all round keys for a particular key during key setup. Then these keys could be sent to all rounds of the design. This would eliminate the Key Scheduler from each of the round implementations which would significantly reduce the area of the design. This is legal because only one transaction is allowed to be in the pipeline at one time.

A second architectural change would be to allow multiple transactions in the pipeline at one time. Currently, only one transaction is allowed to be in the pipeline to simplify error recovery.

A third architectural change is to split the encryption and decryption operations into two cores. This would allow both encryption and decryption to occur simultaneously and would allow the throughput to be optimized for encryption and decryption. Currently, the encryption operation does not use all of the 16 cycles allocated to it.

In addition, more verification should be done. The confidence is high that the encryption and decryption algorithms are working because many thousand values have been encrypted and decrypted correctly. However, the remainder of the KAT's should be sent through the core. As this is about twelve million sets of values, the confidence that the algorithm implementation works will improve. The PLB and DCR interfaces should also undergo more extensive verification. The tests could be further improved by inserting more randomness into the tests.

The synthesis scripts could be optimized. The timing assertions can be made so that they are more exact. In addition, more iterations of optimizations could be run.

While this will have less effect than the architecture changes, the script changes will be much less effort.

There are few changes that can be made with static timing. Most of the values in the timing assertions are set by IBM softcore guidelines or by the PLB and DCR specifications.

Before the AES Encryption Core can be used as a softcore, it must pass Design-For-Test Compliance and the CMOS checks, both of which help ensure testability. To use the core as a hardcore, physical design will be necessary. Implementing as a hardcore could improve the clock cycle time and the area, but would be very labor intensive.

Appendix A: Sample Key Expansion Transformation

Table A.1 outlines the key expansion for a 128-bit key. The resulting round keys are found one word at a time and are displayed in the column w[i]. The starting key used in table A.1 is 2b7e151628aed2a6abf7158809cf4f3c, indicated by w[0] through w[3].

Table A.1 Key Expansion of a 128-bit key.[19]

i (dec)	temp	After RotWord	After SubWord	Rcon[i/ Nk]	After xor with Rcon	w[i-Nk]	w[i]
0							2b7e1516
1							28aed2a6
2							abf71588
3							09cf4f3c
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	a0fafe17					28aed2a6	88542cb1
6	88542cb1					abf71588	23a33939
7	23a33939					09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafe17	f2c295f2
9	f2c295f2					88542cb1	7a96b943
10	7a96b943					23a33939	5935807a
11	5935807a					2a6c7605	7359f67f
12	7359f67f	59f67f73	cb42d28f	04000000	cf42d28f	f2c295f2	3d80477d
				...			
40	575c006e	5c006e57	4a639f5b	36000000	7c639f5b	ac7766f3	d014f9a8
41	d014f9a8					19fadc21	c9ee2589
42	c9ee2589					28d12941	e13f0cc8
43	e13f0cc8					575c006e	b6630ca6

Table A.2 outlines the key expansion for a 192-bit key. The resulting round keys are found one word at a time and are displayed in the column w[i]. The starting key used in table A.2 is 8e73b0f7da0e6452c810f32b809079e562f8ead2522c6b7b, indicated by w[0] through w[5].

Table A.2 Key Expansion of a 192-bit key.[19]

i (dec)	temp	After RotWord	After SubWord	Rcon[i/ Nk]	After xor with Rcon	w[i-Nk]	w[i]
0							8e73b0f7
1							da0e6452
2							c810f32b
3							809079e5
4							62f8ead2
5							522c6b7b
6	522c6b7b	2c6b7b52	717f2100	01000000	707f2100	8e73b0f7	fe0c91f7
7	fe0c91f7					da0e6452	2402f5a5
8	2402f5a5					c810f32b	ec12068e
9	ec12068e					809079e5	6c827f6b
10	6c827f6b					62f8ead2	0e7a95b9
11	0e7a95b9					522c6b7b	5c56fec2
12	5c56fec2	56fec25c	b1bb254a	02000000	b3bb254a	fe0c91f7	4db7b4bd
13	4db7b4bd					2402f5a5	69b54118
14	69b54118					ec12068e	85a74796
15	85a74796					6c827f6b	e92538fd
16	e92538fd					0e7a95b9	e75fad44
17	e75fad44					5c56fec2	bb09386
18	bb095386	095386bb	01ed44ea	04000000	05ed44ea	4db7b4bd	485af057
...							
46	8fcc5006					a7e1466c	282d166a
47	282d166a					9411f1df	bc3ce7b5
48	bc3ce7b5	3ce7b5bc	eb94d565	80000000	6b94d565	821f750a	e98ba06f
49	e98ba06f					ad07d753	448c773c
50	448c773c					ca400538	8ecc7204
51	8ecc7204					8fcc5006	01002202

Table A.3 outlines the key expansion for a 256-bit key. The resulting round keys are found one word at a time and are displayed in the column w[i]. The starting key used in table A.3 is 603deb1015ca71be2b73aef0857d77811f352c073b6108d72d9810a30914dff4, indicated by w[0] through w[7].

Table A.3 Key Expansion of a 256-bit key.

i (dec)	temp	After RotWord	After SubWord	Rcon[i/Nk]	After xor with Rcon	w[i-Nk]	w[i]
0							603deb10
1							15ca71be
2							2b73aef0
3							857d7781
4							1f352c07
5							3b6108d7
6							2d9810a3
7							0914dff4
8	0914dff4	14dff409	fa9ebf01	01000000	fb9ebf01	603deb10	9ba35411
9	9ba35411					15ca71be	8e6925af
10	8e6925af					2b73aef0	a51a8b5f
11	a51a8b5f					857d7781	2067fcde
12	2067fcde		b785b01d			1f352c07	a8b09c1a
13	a8b09c1a					3b6108d7	93d194cd
14	93d194cd					2d9810a3	be49846e
15	be49846e					0914dff4	b75d5b9a
16	b75d5b9a	5d5b9ab7	4c39b8a9	02000000	4e39b8a9	9ba35411	d59aecb8
17	d59aecb8					8e6925af	5bf3c917
18	5bf3c917					a51a8b5f	fee94248
19	fee94248					2067fcde	de8ebe96
20	de8ebe96		1d19ae90			a8b09c1a	b5a9328a
21	b5a9328a					93d194cd	2678a647
22	2678a647					be49846e	98312229
23	98312229					b75d5b9a	2f6c79b3
24	2f6c79b3	6c79b32f	50b66d15	04000000	54b66d15	d59aecb8	812c81ad

Table A.3 Key Expansion of a 256-bit key.

i (dec)	temp	After RotWord	After SubWord	Rcon[i/ Nk]	After xor with Rcon	w[i-Nk]	w[i]
...							
52	7401905a		927c60be			5886ca5d	cafaaae3
53	cafaaae3					2e2f31d7	e4d59b34
54	e4d59b34					7e0af1fa	9adf6ace
55	9adf6ace					27cf73c3	bd10190d
56	bd10190d	10190dbd	cad4d77a	40000000	8ad4d77a	749c47ab	fe4890d1
57	fe4890d1					18501dda	e6188d0b
58	e6188d0b					e2757e4f	046df344
59	046df344					7401905a	706c631e

Appendix B: Sample SubBytes Transformation

This appendix outlines an example of how to compute the values for the SubByte transformation table found in section 2.3.3. This example finds the SubByte transformation for b(x)=x={02}. Table B.1 illustrates the steps used to apply the Extended Euclidean Algorithm. Equation B.1 shows how the multiplicative inverse is found and Equation B.2 finds the SubByte transformation for {02}.

Table B.1 Extended Euclidean Algorithm.

	q(x)	r(x)	a(x)	c(x)	b(x)	m(x)	$a_2(x)$	$a_1(x)$	$c_2(x)$	$c_1(x)$
0					x	$x^8+x^4+x^3+x+1$	1	0	0	1
1	0	x	1	0	$x^8+x^4+x^3+x+1$	x	0	1	1	0
2	$x^7+x^3+x^2+1$	1	$x^7+x^3+x^2+1$	1	x	1	1	$x^7+x^3+x^2+1$	0	1
3	x	0	$x^8+x^4+x^3+x+1$	x	1	0	$x^7+x^3+x^2+1$	$x^8+x^4+x^3+x+1$	1	x
4			$x^7+x^3+x^2+1$	1						

$$b^{-1}(x) = a(x) \bmod m(x) = x^7+x^3+x^2+1 \bmod x^8+x^4+x^3+x+1 = x^7+x^3+x^2+1 = \{1d\} \quad \textbf{(Equation B.1)}$$

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \textbf{(Equation B.2)}$$

Appendix C: Encryption Example

Table C.1 is an 128-bit key encryption example, showing the state after each transformation. The start of round column shows the state after the AddRoundKey transformation.

In this example the input state is 00112233445566778899aabbccddeeff

and the key is 000102030405060708090a0b0c0d0e0f.

Table C.1 Encryption Example. [19]

Round Number	Start of Round	After SubBytes	After ShiftRows	After MixColumns	Round Key Value
input	00 44 88 cc				00 04 08 0c
	11 55 99 dd				01 05 09 0d
	22 66 aa ee				02 06 0a 0e
	33 77 bb ff				03 07 0b 0f
1	00 40 80 c0	63 09 cd ba	63 09 cd ba	5f 57 f7 1d	d6 d2 da d6
	10 50 90 d0	ca 53 60 70	53 60 70 ca	72 f5 be b9	aa af a6 ab
	20 60 a0 e0	b7 d0 e0 e1	e0 e1 b7 d0	64 bc 3b f9	74 72 78 76
	30 70 b0 f0	04 51 e7 8c	8c 04 51 e7	15 92 29 1a	fd fa f1 fe
2	89 85 2d cb	a7 97 d8 1f	a7 97 d8 1f	ff 31 64 77	b6 64 be 68
	d8 5a 18 12	61 be ad c9	be ad c9 61	87 d8 51 3a	92 3d 9b 30
	10 ce 43 8f	ca 8b 1a 73	1a 73 ca 8b	96 6a 51 d0	cf bd c5 b3
	e8 68 d8 e4	9b 45 61 69	69 9b 45 61	84 51 fa 09	0b f1 00 fe
...					
10	bd f2 0b 8b	7a 89 2b 3d	7a 89 2b 3d		13 e3 f3 4d
	6e b5 61 10	9f d5 ef ca	d5 ef ca 9f		11 94 07 2b
	7c 77 21 b6	10 f5 fd 4e	fd 4e 10 f5		1d 4a a7 30
	3d 9e 6e 89	27 0b 9f a7	a7 27 0b 9f		7f 17 8b c5
output	69 6a d8 70				
	c4 7b cd b4				
	e0 04 b7 c5				
	d8 30 80 5a				

Appendix D: Decryption Example

Table D.1 is an 128-bit key decryption example, showing the state after each transformation. The start of round column shows the state after the InvMixColumns transformation. In the cases where the numbers from the "After AddRoundKey" column and the following "Start of Round" column are the same, the InvMixColumns transformation is not applied.

In this example, the input state is 69c4e0d86a7b0430d8cdb78070b4c55a

and the key is 000102030405060708090a0b0c0d0e0f.

Table D.1 Decryption Example. [19]

Round Number	Start of Round	After InvShiftRows	After InvSubBytes	Round Key Value	After AddRoundKey
input	69 6a d8 70			13 e3 f3 4d	7a 89 2b 3d
	c4 7b cd b4			11 94 07 2b	d5 ef ca 9f
	e0 04 b7 c5			1d 4a a7 30	fd 4e 10 f5
	d8 30 80 5a			7f 17 8b c5	a7 27 0b 9f
1	7a 89 2b 3d	7a 89 2b 3d	bd f2 0b 8b	54 f0 10 be	e9 02 1b 35
	d5 ef ca 9f	9f d5 ef ca	6e b5 61 10	99 85 93 2c	f7 30 f2 3c
	fd 4e 10 f5	10 f5 fd 4e	7c 77 21 b6	32 57 ed 97	4e 20 cc 21
	a7 27 0b 9f	27 0b 9f a7	3d 9e 6e 89	d1 68 9c 4e	ec f6 f2 c7
2	54 6b 96 a1	54 6b 96 a1	fd 05 35 f1	47 a4 e0 ae	ba a1 d5 5f
	d9 a0 bb 11	11 d9 a0 bb	e3 e5 47 fe	43 1c 16 bf	a0 f9 51 41
	90 9a f4 70	f4 70 90 9a	ba d0 96 37	87 65 ba 7a	3d b5 2c 4d
	a1 b5 0e 2f	b5 0e 2f a1	d2 d7 4e f1	35 b9 f4 d2	e7 6e ba 23
			...		
10	63 09 cd ba	63 09 cd ba	00 40 80 c0	00 04 08 0c	00 44 88 cc
	53 60 70 ca	ca 53 60 70	10 50 90 d0	01 05 09 0d	11 55 99 dd
	e0 e1 b7 d0	b7 d0 e0 e1	20 60 a0 e0	02 06 0a 0e	22 66 aa ee
	8c 04 51 e7	04 51 e7 8c	30 70 b0 f0	03 07 0b 0f	33 77 bb ff
output	00 44 88 cc				
	11 55 99 dd				
	22 66 aa ee				
	33 77 bb ff				

References

[1] A. Menezes, P. van Oorschot, and S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996.

[2] B. Gladman, Implementation Experience with AES Candidate Algorithms, in The Second AES Conference, February 28, 1999.

[3] B. Gladman, Input and Output Block Conventions for AES Encryption Algorithms, AES Round 2 public comment, April 9, 2000.

[4] B. Gladman, The Need for Multiple AES Winners, AES Round 1 public comment, April 7, 1999.

[5] B. Weeks, et al., Hardware Performance Simulations of Round 2 Advanced Encryptions Standard Algorithms, National Security Agency white paper, May 15, 2000.

[6] J. Daemen and V. Rijmen, AES Proposal: Rijndael, AES algorithm submission, September 3, 1999.

[7] J. Daemen and V. Rijmen, The AES second round Comments of the Rijndael, AES Round 2 public comment, May 12, 2000.

[8] J. Daemen and V. Rijmen, Resistance Against Implementation Attacks: A Comparative Study of the AES Proposals, in The Second AES Candidate Conference, printed by the National Institute of Standards and Technology, Gaithersburg, MD, March 22-23, 1999.

[9] J. Nechvatal, et al., Report on the Development of the Advanced Encryption Standard (AES), printed by the National Institute of Standards and Technology, Gaithersburg, MD, October 2, 2000.

[10] L. Gao and G. Sobelman, Improved VLSI Designs for Multiplication and Inversion in GF(2^M) Over Normal Bases, Proceedings of the 13th Annual IEEE International ASIC/SOC Conference, Arlington, VA, September 13-16, 2000, pp. 97-101.

[11] B. Blaner, D Czenkusch, R. Devins, and S. Stever, An Embedded PowerPC SOC for Test and Measurement Applications, Proceedings of the 13th Annual IEEE International ASIC/SOC Conference, Arlington, VA, September 13-16, 2000, pp. 204-208.

[12] T. Ichikawa, T. Kasuya, and M. Matsui, Hardware Evaluation of the AES Finalists, in The Third AES Candidate Conference, printed by the National Institute of Standards and Technology, Gaithersburg, MD, April 13-14, 2000.

[13] AES discussion forum: http://aes.nist.gov/aes

[14] AES home page: http://www.nist.gov/aes

[15] Core Connect Bus Architecture: A 32-, 64-, 128-bit core on-chip bus standard, IBM Corporation, 1999.

[16] The Core Connect Bus Architecture, IBM Corporation, 1999.

[17] Device Control Register Bus: Architecture Specifications Version 2.8, IBM Corporation, 1998.

[18] DCR Bus Functional Model Toolkit: User Manual Version 2.6, IBM Corporation, 1998.

[19] Federal Information Processing Standards Publication 197, Advanced Encryption Standard, U.S. DoC/NIST, November 26, 2001.

[20] Initial plans for estimating the hardware performance of AES Submissions, AES Round 2 Analysis.

[21] 128-bit Processor Local Bus: Architecture Specifications Version 4.3, IBM Corporation, 2000.

[22] PLB Functional Model Toolkit: User manual Version 3.4, IBM Corporation, 1998.