VHDL design of a DES encryption cracking system

Thomas Oelke

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
VHDL Design of a DES Encryption

Cracking System

by

Thomas Oelke

A Thesis Submitted
in
Partial Fulfillment of the
Requirements of the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:

Roy S. Czernikowski, Professor

Muhammad E. Shaaban, Assistant Professor

Kenneth W. Hsu, Professor

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York
June, 1997
Release Permission Form

Rochester Institute of Technology

VHDL Design of a DES Encryption

Cracking System

I, Thomas Oelke, hereby grant permission to any United States citizen or permanent resident or Canadian citizen to reproduce this thesis in whole or in part for non-commercial and non-profit purposes only, providing they agree to not export the contents and agree not to otherwise violate United States Export Administration Regulations or other applicable export laws.

________________________________________
Thomas Oelke

June 16, 1997
Date
Abstract

This thesis illustrates the design of a chip to crack a message encrypted with Digital Encryption Standard (DES). VHSIC Hardware Description Language (VHDL) is used to describe the system. Part of the design criteria of the system is to provide a scalable and reconfigurable set of DES building blocks in VHDL. In order to provide this, a modular design with a pipeline architecture is employed. This system could be synthesized to produce actual hardware in either an ASIC or FPGA part. Simulations using Synopsys with Actel's 3200DX FPGA library demonstrate that the design could be run at over 16Mhz. Because a pipelined architecture is employed which retires one key every clock cycle the chip would be able to test over 16 million keys per second. This is a vast improvement over current software-only based approaches that achieve speeds of 1 to 2 million keys per second on expensive high-end micro-processors.
## Table of Contents

Abstract......................................................................................................................... iii

Table of Contents................................................................................................. iv

List of Figures ........................................................................................................ vii

List of Tables ........................................................................................................ viii

Glossary ................................................................................................................... ix

1 Introduction .......................................................................................................... 1

2 Description of DES ............................................................................................. 4

   2.1 Outline of Algorithm .................................................................................... 4

   2.2 Key Transformation ....................................................................................... 6

   2.3 Initial Permutation ......................................................................................... 7

   2.4 Round operation ............................................................................................. 8

   2.5 Key flow ......................................................................................................... 10

   2.5.1 Rotate ....................................................................................................... 10

   2.5.2 Compression Permutation ......................................................................... 10

   2.6 Data flow ........................................................................................................ 11

   2.6.1 Expansion Permutation ........................................................................... 11

   2.6.2 S-Box Substitution ................................................................................... 12

   2.6.3 P-Box Permutation .................................................................................. 13

3 DES cracking strategies ....................................................................................... 14

   3.1 DES Weak Keys ............................................................................................ 14
3.2 Known plaintext ................................................................. 14
  3.2.1 Brute force ................................................................. 15
3.3 Chosen plaintext ................................................................. 15
  3.3.1 Linear cryptanalysis ...................................................... 16
  3.3.2 Differential cryptanalysis .............................................. 16
  3.3.3 Differential-Linear cryptanalysis ..................................... 17

4 Design of key tester .............................................................. 18
  4.1 Architecture ................................................................. 18
  4.2 Interface ........................................................................ 19
    4.2.1 Description of Interface Signals ................................... 20
  4.3 Encryption Core ............................................................... 21
    4.3.1 Encryption Core Signals ............................................ 21
    4.3.2 Encryption Core Implementation .................................. 22
  4.4 Backend ........................................................................ 25
    4.4.1 Back End Signals ..................................................... 25
    4.4.2 Back End Implementation .......................................... 26
  4.5 Front End ...................................................................... 27
    4.5.1 Front End Signals .................................................... 27
    4.5.2 Front End Implementation .......................................... 28
  4.6 Key Generator ................................................................. 30
    4.6.1 Key Generator Signals .............................................. 30
    4.6.2 Key Generator Implementation ................................... 31
4.7 Design Optimizations ................................................................. 33

5 VHDL and Synopsys ........................................................................ 35
  5.1 VHDL ...................................................................................... 35
  5.2 Synopsys .................................................................................. 36

6 Testing and Verification .................................................................. 37
  6.1 VHDL Testing ........................................................................... 37

7 Results and Conclusions ................................................................. 39
  7.1 Performance ............................................................................. 39
  7.2 Future areas ............................................................................. 40

Bibliography ....................................................................................... 42

Appendix A: S-Box Tables ................................................................. 44

Appendix B: VHDL Code ................................................................. 46

Appendix C: Synopsys Generated Schematics ................................. 98

Appendix D: Simulation Results for Encryption Core ..................... 109
List of Figures

Figure 2-1 DES Encryption Algorithm [1] .................................................................5
Figure 2-2 One Round of DES [1] .............................................................................9
Figure 2-3 Expansion Permutation [1] [3] ...............................................................12
Figure 2-4 S-Box Substitution .................................................................................13
Figure 4-1 Key tester architecture .........................................................................18
Figure 4-2 Encryption Core ......................................................................................23
Figure 4-3: Encryption Core - Dual Pipeline .........................................................24
Figure 4-4 Back End .................................................................................................26
Figure 4-5 Front End ...............................................................................................29
Figure 4-6 Key Generator .......................................................................................32
Figure C-1 Full Chip ...............................................................................................99
Figure C-2 Front end before optimization .........................................................100
Figure C-3 Front end after optimization ..........................................................101
Figure C-4 Key generator before optimization ................................................102
Figure C-5 Key generator after optimization .....................................................103
Figure C-6 Encryption core before optimization .............................................104
Figure C-7 Encryption core after optimization ...............................................105
Figure C-8 One round of encryption after optimization ...................................106
Figure C-9 Back end before optimization ........................................................107
Figure C-10 Back end after optimization ..........................................................108
List of Tables

Table 2-1. Key permuted choice - selects and permutes 56 bits from 64 bit input[1][3].....6
Table 2-2 Data Initial Permutation.................................................................8
Table 2-3 Number of Key Bits Rotated Left Per Round [1][3]..........................10
Table 2-4 Key Compression Permutation [1][3] .............................................11
Table 2-5 P-box Permutation [1][3] ...............................................................13
Table 4-1 Interface Signals.............................................................................19
Table 4-2 Encryption Core Signals.................................................................21
Table 4-3 Back End Signals.............................................................................25
Table 4-4 Front End Signals.............................................................................27
Table 4-5 Key Generator Signals....................................................................30
Glossary

ASIC  Application Specific Integrated Circuit - an integrated circuit (microchip) made for a specific use.

DES  Data Encryption Standard - a standardized block encryption algorithm that utilizes 64 bit blocks and a 56 bit key. (See Chapter 2 for in depth discussion of DES)

FPGA  Field programmable Gate Array - chip which contains numerous logic gates that can be connected and reconnected in different configurations to provide different functionality.

Round  Operation performed in DES that is repeated 16 times. This is a fairly complex function involving both S-Boxes, permutations, and exclusive-or-ing. (See Chapter 2 for full details)

S-Box  Non-linear function that provides DES with much of its security. Takes a 6 bit input and produces a 4 bit output. (See Chapter 2 for full details)

VHDL  VHSIC Hardware Description Language. It is an IEEE and ANSI standard for description of digital designs. Was originally defined for use by U.S. Government.

VHSIC  Very High Speed Integrated Circuit.
1 Introduction

The purpose of this thesis is to demonstrate the use of VHDL to create a system that could be utilized for cracking a message encrypted using DES.

DES stands for Digital Encryption Standard, and was developed in the 1970's and adopted in 1981 by the U.S. Government.[1][3][11] It is regarded one of the more secure encryption systems available in widespread use. It is a 64 bit block cipher, with a 56 bit key size. This means that the input and output blocks are 64 bits long, and the key used for encryption and decryption is 56 bits long.

Currently, the most practical way to attack DES is through brute force. Since the system relies on a 56-bit key, there are $2^{56}$ possible keys to check. This converts to 72,057,594,037,927,936 (decimal) possible keys which reads as Seventy-Two Quadrillion, Fifty-Seven Trillion, Five-Hundred Ninety-Four Billion, Thirty-Seven Million, Nine-Hundred Twenty-Seven Thousand, Nine-Hundred Thirty-Six possible keys. [11]

On average however, the key will be found half way through an exhaustive search of the keyspace. This means on the average only thirty-six quadrillion keys will have to be searched before finding the correct key.
DES is widely used commercially, including use by financial institutions. However because of export restrictions on cryptography it is only possible to export DES from the U.S. under strict guidelines. Essentially the only time DES is allowed to be exported is for use by financial institutions. Otherwise it is normally not allowed to be exported as its 56 bit key length is far greater than the 40 bit key length that is currently allowed to be freely exported from the U.S.

There are a number of reasons that an encryption engine described in VHDL such as this would be useful. One is that use of it to crack DES at a reasonably low cost and time would encourage the government to expand what it allows to be exported. In addition, this could be potentially used commercially to perform cracking of DES messages. One possible use would be to provide access to material that had been encrypted with DES, but that the key utilized had been forgotten.

Also the VHDL encryption core could be utilized as the basis of hardware specifically to perform DES encryption or decryption. This could be embedded as part of a computer system to provide dedicated hardware encryption and or decryption utilizing DES. If FPGAs were utilized as the basis for the hardware this encryption / decryption method could be upgraded or changed to a different methodology without having to physically change components.
Because VHDL is a well established standard, it is ideal for this type of application. The VHDL code can be utilized by a number of different synthesizers, including Synopsys, and targeted for numerous different architectures, from full custom ASICs to FPGA's from numerous possible different vendors, including Xilinx, Actel, and Altera.
2 Description of DES

2.1 Outline of Algorithm

DES is a 64 bit block encryption algorithm.[1][3] The general algorithm is illustrated in Figure 2-1. DES is a symmetric algorithm, so the same algorithm and key are used for both the encryption and decryption with some small differences in the key schedule.

To perform the algorithm, the block of data to be encrypted is first permuted in an initial permutation. It is then broken into a right and left half. After this, it enters into what is the main computational core of the algorithm. These main computations are repeated 16 times, and referred to as rounds. In a round, the right half is moved to the left half. After this, the former left half is exclusive-or-ed with the results of a function from the former right half and a subkey, and the result becomes the new right half. Description of the operation of each round is described in further detail in following sections. After the 16 rounds are completed, the data is put through another final permutation which results in the ciphertext.[1][3]
Figure 2-1 DES Encryption Algorithm [1]

Plaintext

IP

$L_0$ - (Left half)  $R_0$ - (Right half)

$L_1 = R_0$  $R_1 = L_0 \oplus f(R_0, K_1)$

$L_2 = R_1$  $R_2 = L_1 \oplus f(R_1, K_2)$

$L_{15} = R_{14}$  $R_{15} = L_{14} \oplus f(R_{14}, K_{15})$

$R_{15} = L_{14} \oplus f(R_{14}, K_{15})$  $L_{15} = R_{14}$

FP

Ciphertext

f() described in Figure 2-2 - 1 round of encryption
2.2 Key Transformation

The key length is 56 bits. It is normally expressed as a 64-bit number with every eighth bit being used for parity. A permutation is also done to alter the order of the bits that make up the 56 bits used for the key. This permutation is described in table 2-1, and uses as its input the 64 bit expression of the key, and has as its output the 56 bits that will actually be used for generating the subkeys. From these 56 bits, 16 subkeys are extracted, each 48 bits long. Each one of these is used for a round in the algorithm. To create the subkeys the key is split into two halves, and these are rotated left 1 or 2 bits per round, dependent on which round, to create the input to generate the key. The resulting halves are then put through a operation called a compression permutation. This compression permutation both permutes the bits, that is it changes the order of them, and it also selects a subset of 48 of the 56 bits. The result of this provides the 48 bit subkey for that round. There isn’t any significance to this algorithm other than to eliminate the parity bits and scramble the key bits.

Table 2-1. Key permuted choice - selects and permutes 56 bits from 64 bit

input[1][3]

<table>
<thead>
<tr>
<th></th>
<th>57</th>
<th>49</th>
<th>41</th>
<th>33</th>
<th>25</th>
<th>17</th>
<th>9</th>
<th>1</th>
<th>58</th>
<th>50</th>
<th>42</th>
<th>34</th>
<th>26</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>2</td>
<td>59</td>
<td>51</td>
<td>43</td>
<td>35</td>
<td>27</td>
<td>19</td>
<td>11</td>
<td>3</td>
<td>60</td>
<td>52</td>
<td>44</td>
<td>36</td>
</tr>
<tr>
<td>63</td>
<td>55</td>
<td>47</td>
<td>39</td>
<td>31</td>
<td>23</td>
<td>15</td>
<td>7</td>
<td>62</td>
<td>54</td>
<td>46</td>
<td>38</td>
<td>30</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>6</td>
<td>61</td>
<td>53</td>
<td>45</td>
<td>37</td>
<td>29</td>
<td>21</td>
<td>13</td>
<td>5</td>
<td>28</td>
<td>20</td>
<td>12</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

6
2.3 Initial Permutation

The initial permutation has no significant affect on the security of the DES algorithm. In fact it is usually done once in software implementations that are designed to test multiple keys trying to find the original key used for a known plaintext and known ciphertext. It is done this way because the bitwise permutations necessary are more difficult to do in software than they are in hardware. It does provide some confusion of the bits that may cause problems for someone casually attempting to cryptanalyze DES. However it provides no true additional security. It is however, a part of the standard, so should be part of any description of the standard.

The permutation is described in the following table, Table 2-2. The location represents the new bit location and the number represents the bit that will be placed in that location. As can be seen each number between 1 and 64 is used once and only once. This table should be read from left to right, top to bottom. So we can see from the table that bit 58 moves to bit 1, bit 50 moves to bit 2, and so on, up to bit 7 which moves to bit 64.
Table 2-2 Data Initial Permutation

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>50</td>
<td>42</td>
<td>34</td>
<td>26</td>
<td>18</td>
<td>10</td>
<td>2</td>
<td>60</td>
<td>52</td>
<td>44</td>
<td>36</td>
</tr>
<tr>
<td>62</td>
<td>54</td>
<td>46</td>
<td>38</td>
<td>30</td>
<td>22</td>
<td>14</td>
<td>6</td>
<td>64</td>
<td>56</td>
<td>48</td>
<td>40</td>
</tr>
<tr>
<td>57</td>
<td>49</td>
<td>41</td>
<td>33</td>
<td>25</td>
<td>17</td>
<td>9</td>
<td>1</td>
<td>59</td>
<td>51</td>
<td>43</td>
<td>35</td>
</tr>
<tr>
<td>61</td>
<td>53</td>
<td>45</td>
<td>37</td>
<td>29</td>
<td>21</td>
<td>13</td>
<td>5</td>
<td>63</td>
<td>55</td>
<td>47</td>
<td>39</td>
</tr>
</tbody>
</table>

2.4 Round operation

The round operation block diagram is shown in figure 2-2. There are two main flows involved in this diagram. The first is the key flow, which consists of simply the rotates and the compression permutation that generates the subkey. Then there is the data flow, which has the main computational elements and the data being worked on.
Figure 2-2 One Round of DES [1]
2.5 Key flow

2.5.1 Rotate

The number of bits rotated left is determined by the round for which the key is being generated. Table 2-3 has the number of bits rotated per round. These rotates are done independently on each half of the 56 bit key.

Table 2-3 Number of Key Bits Rotated Left Per Round [1][3]

<table>
<thead>
<tr>
<th>Round</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

2.5.2 Compression Permutation

The bits that are utilized and the order that they are presented to the data flow is determined by this algorithm. Because of the rotates each of the bits is occasionally not represented. On average each bit is part of the subkey 14 times out of 16.[1] Table 2-4 shows which bits become part of the 48 bit subkey and what places they appear in the subkey.


Table 2-4 Key Compression Permutation [1][3]

<table>
<thead>
<tr>
<th>14</th>
<th>17</th>
<th>11</th>
<th>24</th>
<th>1</th>
<th>5</th>
<th>3</th>
<th>28</th>
<th>15</th>
<th>6</th>
<th>21</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>19</td>
<td>12</td>
<td>4</td>
<td>26</td>
<td>8</td>
<td>16</td>
<td>7</td>
<td>27</td>
<td>20</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>41</td>
<td>52</td>
<td>31</td>
<td>37</td>
<td>47</td>
<td>55</td>
<td>30</td>
<td>40</td>
<td>51</td>
<td>45</td>
<td>33</td>
<td>48</td>
</tr>
<tr>
<td>44</td>
<td>49</td>
<td>39</td>
<td>56</td>
<td>34</td>
<td>53</td>
<td>46</td>
<td>42</td>
<td>50</td>
<td>36</td>
<td>29</td>
<td>32</td>
</tr>
</tbody>
</table>

2.6 Data flow

2.6.1 Expansion Permutation

This function takes the input 32 bits and expands them to 48 bits. In order to do this, some bits appear twice. As can be seen in Figure 2-3 it is a repeating function with 4 bit input and 6 bit output. So that a single bit doesn’t affect two adjacent bits, the function is such that the edge bits, the first and fourth of the input block, are duplicated but the duplicates are sent to the adjacent blocks. The edge bits on the output, the first and sixth bits, are from the adjacent blocks. The center 4 bits of the output however are the same as the input bits and are in the same order.
2.6.2 S-Box Substitution

After xoring the bits from the expansion permutation with the subkey bits from the compression permutation, the 48 bit result is used as a key to 8 independent S-boxes, as can be seen in figure 2-4. These S-boxes take 6 bits as an input and produce a 4 bit output. As the other elements of the algorithm are linear and therefore easy to analyze. The non-linear S-boxes give DES its security.[1] The S-boxes are defined by the DES algorithm and are essentially a set of lookup tables. However they may be implemented as either a memory element, with some bits to designate the row and others for the column to access the bit, or as a set of discrete logic gates as suggested in [2]. A full table of the S-box functions is given in Appendix A.
2.6.3 P-Box Permutation

The results of the S-boxes are concatenated together to form a 32 bit wide bus. This is then permuted as shown in table 2-5. After the permuted result is exclusive-or-ed with the old left side it becomes the new right side.

Table 2-5 P-box Permutation [1][3]

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>7</td>
<td>20</td>
<td>21</td>
<td>29</td>
<td>12</td>
<td>28</td>
<td>17</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>23</td>
<td>26</td>
<td>5</td>
<td>18</td>
<td>31</td>
<td>10</td>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

13
3 DES cracking strategies

3.1 DES Weak Keys

There are a small set of keys within DES that do not provide as much protection for data encrypted with them.[4][5] One of the ways that these may be weak is if there is a limited number of subkeys generated. If each half of the keys is all ones or all zeros the same subkey will be generated for all the iterations. In addition there are some keys that will generate only 2 or only 4 subkeys. There are only a total of 64 keys that have these flaws, so it is highly unlikely that one of these keys will be used. In addition, because these keys are known to be weaker, generally they are eliminated before they might be used to encrypt anything.

3.2 Known plaintext

A ‘known plaintext’ attack is when the plaintext (or at least a portion of it) being encrypted is known by the person trying to break the encryption. Most attacks fit into this category. In real applications known plaintext attacks are usually most applicable. In most cases, at least some section of the plaintext being encrypted is known. For example if an email message is encrypted, it is known that the first line starts with ‘From ‘. In addition, other parts of the message can either be guessed at or when one
section is decrypted it can be examined for whether it matches a part of the message that is expected. Many other types of data have similar headers or footers that can be exploited if the type of data being sent is known, a few examples include GIF files, ZIP files, and JPEG files.

3.2.1 Brute force

DES, like most encryption schemes, can be broken by brute force using a known plaintext and a known ciphertext. Essentially each key is tried individually and tested to see if that key will encrypt the plaintext to produce a matching ciphertext result. Most of the attacks against DES are of the brute force type as with known plaintext it is the least computationally intensive method known so far. Brute force attacks can be done with either an encryption of the plaintext and comparison to the ciphertext or with decryption of the ciphertext and comparison to the plaintext. With DES either way requires the same amount of computation. This is because DES utilizes the same key and methodology for both encryption and decryption.

3.3 Chosen plaintext

In some cases the person attempting to break the encryption will have the luxury of being able to encrypt whatever plaintext they choose utilizing the unknown key. If this is the case, linear and/or differential cryptanalysis can be more easily applied.
### 3.3.1 Linear cryptanalysis

Linear cryptanalysis utilizes linear approximations to describe the action of a block cipher, such as DES. It involves exploiting the bias of some bits on the output based on the input. A full description of what is involved is beyond the scope of this paper but can be found in [6][7][8]. In order to find the bias a number of input bits are xor-ed with some number of output bits. This will give a result that with some probability indicates the state of a specific bit or set of bits within the key. Against full DES, an attack based on linear cryptanalysis can recover the key in an average of $2^{43}$ plaintexts.[1]

### 3.3.2 Differential cryptanalysis

This approach examines the difference between two plaintext inputs and the resulting effect on the output, giving indication of the key used to do the encryption. As more and more ciphertext pairs are analyzed, a key will emerge as the most probable to have been used. It is when the plaintext can be chosen that this approach is most feasible. Schneir states “Differential cryptanalysis works against DES and other similar algorithms with constant S-boxes.”[1] He goes on to state that the S-Boxes used by DES are optimized against differential cryptanalysis. With a full 16 round DES this attack is highly unlikely as it requires $2^{47}$ chosen plaintexts or $2^{55}$ known plaintexts. In addition $2^{37}$ DES operations have to be performed. Because this is so large this attack
has remained largely theoretical. In addition for 16 round DES, this attack is slightly less efficient than a simple brute force attack.[1]

### 3.3.3 Differential-Linear cryptanalysis

This is a new approach that may provide a better method for attacking DES.[1] Currently there is only an attack on an 8 round variant of DES that recovers 10 of the bits with an 80 percent probability when 512 chosen plaintexts are used. It doesn’t appear that it extends easily to more rounds, however the work is still very new.[1] It is quite possible that this may become the basis for a new attack on full DES.
4 Design of key tester

4.1 Architecture

The main architecture is shown in Figure 4-1. The crux of the architecture is a pipelined encryption engine that will perform the actual DES encryption. The surrounding blocks provide support for feeding the core and evaluating the results output from the core.

Figure 4-1 Key tester architecture
4.2 Interface

The interface was designed to allow easy loading of keyspace to search. It also facilitates the loading of data to encrypt and ciphertext to be compared. Another design consideration was to provide easy notification of when the key is found and output of the actual key when found. In order to provide this using a bus width that could be utilized by most processors, an eight bit data bus was chosen with a number of control bits and address bits. A simple 68HC11 or other 8 bit processor could easily control this architecture without any problems. The resulting interface lines are as follows:

**Table 4-1 Interface Signals**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Number bits / pins</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>1</td>
<td>input</td>
</tr>
<tr>
<td>Reset</td>
<td>1</td>
<td>input</td>
</tr>
<tr>
<td>Data Input</td>
<td>8</td>
<td>input</td>
</tr>
<tr>
<td>Address</td>
<td>5</td>
<td>input</td>
</tr>
<tr>
<td>Key Output</td>
<td>8</td>
<td>output</td>
</tr>
<tr>
<td>Key matched</td>
<td>1</td>
<td>output</td>
</tr>
<tr>
<td>Section done</td>
<td>1</td>
<td>output</td>
</tr>
</tbody>
</table>
4.2.1 Description of Interface Signals

Clock: Clock used for controlling pipeline used in testing keys. This is what controls the speed at which keys are checked.

Reset: Resets the chip. When used will start searching the next keyspace that was loaded. Should only be utilized once at startup.

Data Input: data bus for inputting the keyspace, the plaintext and the ciphertext to be compared against.

Address: Controls whether the data goes to the keyspace, the plaintext or the ciphertext and which byte within those. Two bits are needed to differentiate whether a section of the plaintext, ciphertext or keyspace is being loaded. The other three bits are used for which byte is being loaded. These bits also are used to determine which byte of the result is output.

Key Output: Output bus for the key that produces a match. Only valid when there has been a matching key found.

Key matched: Signals when a matching key is found.

Section Done: Signals when done with a specific key space.
4.3 Encryption Core

The encryption core is the central part of the key checker. It is what does the actual encryption of data that is being checked.

Table 4-2 Encryption Core Signals

<table>
<thead>
<tr>
<th></th>
<th>Number bits</th>
<th>Input / Output</th>
<th>How often changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key in</td>
<td>56</td>
<td>input</td>
<td>1 / clock cycle</td>
</tr>
<tr>
<td>Plaintext</td>
<td>64</td>
<td>input</td>
<td>startup</td>
</tr>
<tr>
<td>Ciphertext</td>
<td>64</td>
<td>input</td>
<td>1 / clock cycle</td>
</tr>
<tr>
<td>Key out</td>
<td>56</td>
<td>output</td>
<td>1 / clock cycle</td>
</tr>
</tbody>
</table>

4.3.1 Encryption Core Signals

Clock: Clock used for controlling pipeline used in testing keys. This is what controls the speed at which keys are checked.

Key In: Key to be used to encrypt the plaintext being input.

Plaintext: Data input that is to be encrypted.

Result: Result of plaintext encrypted with the key ‘Key Out.’

Key Out: Key used to encrypt the ciphertext that is being output currently.
4.3.2 Encryption Core Implementation

A pipelined architecture will be used in order to provide higher throughput. In addition it will eliminate the need for gates to keep an internal state. This means that all gates will be utilized for actual computation of the encryption algorithm.

In order to provide easy scalability to a dual pipeline, the stages for generating the keys and the stages for processing the data were kept separate. The same VHDL entities could be used to create a dual pipeline architecture that could be used to encrypt a different second plaintext. Because of one of the properties of DES this would provide the ability to test the inverse of the keys. The architecture for a dual pipeline core is shown in Figure 4-3.
Figure 4-2 Encryption Core

The diagram illustrates the encryption process with the following details:

- **Plaintext** input is 64 bits.
- **Key In** input is 56 bits.
- **IP** and **KP** blocks handle the initial and final data transformations.
- **Round (1)** through **Round (4)** each process 64-bit data, followed by **Key round** operations.
- **Rounds 5-14** consist of multiple repeated key rounds.
- **Round (15)** and **Round (16)** with 56-bit sub-keys.
- The **FP** and **KP\(^{-1}\)** blocks complete the encryption process.
- The **Result** output is 64 bits, and the **Key Out** is 56 bits.

The architecture is designed to process data through multiple rounds of encryption, with key rounds alternating between the plaintext and key inputs.
Figure 4-3: Encryption Core - Dual Pipeline

Plaintext

Key In

In Plaintext "P"

56

Data

KP

64

Data

Round (1) /-

Key Round (1) /-

Round (1)

64 48 sub-key

Round (2)

Data -h

64 56 48 sub-key

Data

Round (3) -f

Key Round (3) -/

Round (1)

Data

Round (4)

h-

Key Round (4) h-

Round (1)

Data

I

64 sub-key

^64 56 sub-key

^ Data

^64 48 sub-key

Rounds 5-14

Round (15) /-

Key Round (15) H

Round (1)

64 48 sub-key

Round (16)

Data -/

56 56 48 sub-key

Data

Key Round (16) FP

64 56 48 sub-key

Data

KP-

64 64 key

64 56 48 48 sub-key

FP 56 key

Ciphertext (E(P, k))

Key

Out

Ciphertext (E(P, k'))

24
4.4 Backend

This part of the chip has the purpose of comparing the ciphertext created by the encryption core and that being searched for. If a match is found, the key that was used to create the ciphertext is held to be output to the user.

Table 4-3 Back End Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Number bits</th>
<th>Input / Output</th>
<th>How often changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>56</td>
<td>input</td>
<td>1 / clock cycle</td>
</tr>
<tr>
<td>Ciphertext</td>
<td>64</td>
<td>input</td>
<td>1 / clock cycle</td>
</tr>
<tr>
<td>Compare text</td>
<td>64</td>
<td>input</td>
<td>startup</td>
</tr>
<tr>
<td>Address</td>
<td>3</td>
<td>input</td>
<td></td>
</tr>
<tr>
<td>Reset</td>
<td>1</td>
<td>input</td>
<td></td>
</tr>
<tr>
<td>Key Found</td>
<td>1</td>
<td>output</td>
<td></td>
</tr>
<tr>
<td>Matching Key</td>
<td>8</td>
<td>output</td>
<td></td>
</tr>
</tbody>
</table>

4.4.1 Back End Signals

Key: Key that was used to encrypt the ciphertext currently being input

Ciphertext: Data that was encrypted and is to be compared to.

Compare text: Data to be compared to. (What is hopefully going to be the result of the encryption)

Address: Controls which 8 bits of the key that matched are output.
Reset: Resets the system. When put to ‘1’, the key found signal is cleared until a new match occurs.

Key Found: Goes to ‘1’ when a match occurs. Goes to ‘0’ on reset.

Matching Key: 8 bits of the key that was being input when a match between ciphertext and comparetext occurred.

4.4.2 Back End Implementation

This is primarily a 64 bit comparator, with a register to store the key when a match occurs. Figure 4-4 shows the general architecture of the system.

Figure 4-4 Back End
4.5 Front End

The Front End section provides the interface for inputs from outside the chip. It mainly is a means for demultiplexing the input and latching the input for use later within the chip.

Table 4-4 Front End Signals

<table>
<thead>
<tr>
<th></th>
<th>Number bits</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>8</td>
<td>input</td>
</tr>
<tr>
<td>Address</td>
<td>5</td>
<td>input</td>
</tr>
<tr>
<td>Write</td>
<td>1</td>
<td>input</td>
</tr>
<tr>
<td>Keyspace</td>
<td>24</td>
<td>output</td>
</tr>
<tr>
<td>Compare text</td>
<td>64</td>
<td>output</td>
</tr>
<tr>
<td>Plaintext</td>
<td>64</td>
<td>output</td>
</tr>
</tbody>
</table>

4.5.1 Front End Signals

**Data:** Data to become part of the different outputs.

**Address:** Determines which 8 bit section of which output will be set to what is on ‘data’ when write equals ‘1’.

**Write:** Signal for when to latch data in.

**Keyspace:** Section of the key space to be searched.

**Plaintext:** Data to be encrypted.

**Compare text:** Data to be compared to. (What is hopefully going to be the result of the encryption)
4.5.2 Front End Implementation

This is essentially just a set of latches with the triggers being the write input after passing through a demultiplexor. The address bits control which section of the registers is active, the data is what is to be stored, and the write bit signifies when the data is valid to be stored into the section specified by the address.

Figure 4-5 displays the general architecture of the front end. In order for there to be an easily understandable and viewable diagram, the latches have been simplified to some extent. Rather than display all 19 of the 8 bit latch sections, a representative of the output registers is shown.
Figure 4-5 Front End

Data

Keyspace

Address

Plaintext

Ciphertext

Demux

Write

29
4.6 Key Generator

The Key Generator creates the keys to be tested. It is desirable for it to cycle through all the keys in the key space and provide a signal of when it has done so.

Table 4-5 Key Generator Signals

<table>
<thead>
<tr>
<th></th>
<th>Number bits</th>
<th>Input / Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyspace</td>
<td>24</td>
<td>input</td>
</tr>
<tr>
<td>Clock</td>
<td>1</td>
<td>input</td>
</tr>
<tr>
<td>Reset</td>
<td>1</td>
<td>input</td>
</tr>
<tr>
<td>Key</td>
<td>56</td>
<td>output</td>
</tr>
<tr>
<td>Section Done</td>
<td>1</td>
<td>output</td>
</tr>
</tbody>
</table>

4.6.1 Key Generator Signals

Clock: Clock driving the system.

Keyspace: Next section of the key space to be searched.

Reset: Resets pseudo-random generator to 1’s and loads the next key space.

Key: The key to be tested next.

Section Done: Shows when finished the last section, so new keyspace can be loaded.
4.6.2 Key Generator Implementation

The heart of the design is a pseudo-random pattern generator. It is based on a page from [9] that gives the proper polynomial that can be used and provide a feedback that will exercise all possible combinations of the shift register except all 0’s. In addition because there are only 3 gate levels that must be gone through between clocks, it is more than fast enough for this application. It provides a method that traverses nearly the entire range while costing very few gates. If it were implemented as a straight arithmetic increment between each clock cycle, there would be many more gates involved to create the 32 bit adder, and it would be considerably slower than this method.

Every $2^{32} - 1$ clock cycles the pattern generator will equal all 1’s, and at this point, the new keyspace will be loaded. Any initial point could be utilized except for all zeros, but an initial point of all ones is easily recognizable. Depending on the underlying technology utilized a different pattern might allow fewer gates to be used. When a full reset is done, the pattern generator will be set to all 1’s. This really should only be necessary at the start of a new search.
Figure 4-6 Key Generator

32 bit pseudo-random pattern generator

24 bit latch

Key
4.7 Design Optimizations

If enough gates / area is available a dual pipeline architecture could be utilized that would test a key and its inverse at the same time. A diagram of a single pipeline architecture is shown in figure 4-2. Figure 4-3 shows a dual pipeline architecture. The reason a dual pipeline architecture with only one key schedule is possible is because DES has a symmetrical property. This property is such that if plaintext P is encrypted with key K to produce ciphertext C, then encryption of the complement of P (P') with the complement of K (K') will produce the complement of the ciphertext (C'). So the identity is as follows.

\[ E_K(P) = C \iff E_{K'}(P') = C' \]

If the goal is only to try to find the key to one specific plaintext - ciphertext pair, the front end sections associated with loading the ciphertext and plaintext can be eliminated. In addition the complexity of some of the gates in the first and second round can be reduced or those gates may possibly even be eliminated totally.

An additional measure that could be utilized to reduce the number of gates needed would be to hold certain key bits constant on each processor. If there are 64 processors one s-box could potentially be totally eliminated from the first round, as the bits on the input would be predetermined, so the output would already be known. Six
bits of the key are exclusive-or-ed with the plain text in order to produce the input to the s-box. If the key bits are held constant (which also would denote which processor of the 64) and the plaintext input is held constant, the inputs to the s-box are going to be constant. This will result in the outputs of the s-box being constant, which means the s-box is unnecessary. As the number of processors is increased, the number of keybits that can be held constant is increased, and the amount of gates that can be eliminated is increased.
5 VHDL and Synopsys

5.1 VHDL

VHDL is a hardware description language. It is an IEEE and ANSI standard for description of digital designs. The code written for this thesis was made to be compatible with the 1987 version of the VHDL standard. This was done so that the code could be synthesized utilizing Synopsys.

VHDL allows for easy reuse of code through structural models. The model can be either structural or behavioral based, or a combination of the two. The code for this thesis was written in a structural style at the upper levels, providing the architectural framework. At the lower component levels the code was written in a behavioral style so that it could be easily simulated and also having some flexibility when being synthesized.[10]

For compilation and behavioral testing Mentor Graphics Quick VHDL suite was used, including qvhcom and qvhsim.
5.2 Synopsys

Synopsys is a commercial tool that takes VHDL or Verilog code and produces hardware designs. In this case it was used to synthesize VHDL code to produce results for programming an FPGA. In addition it was used to visually verify that the code written resulted in the architecture desired.
6 Testing and Verification

6.1 VHDL Testing

The VHDL code was tested and verified utilizing test benches also written in VHDL. Each of the major sections of the chip had a test bench written for it. The test benches for the front end and back end were written by hand and verified that those sections behaved as expected. Because these sections were fairly straightforward, it was relatively easy to write test benches for them by hand and verify the results.

The key generator required more extensive testing, as it had greater complexity. In order to confirm that the pseudo-random pattern generator would perform as expected, a C program was written that would provide the same output as the pattern generator. A variation of this C program was also utilized to confirm that the pattern generator did go through $2^{32}-1$ patterns before returning to its starting pattern. In order to provide a VHDL test bench, a Perl script was written that took as input the C program’s results, and utilized that to generate an array of patterns for the pattern generator to be checked against. The array was then utilized as part of a VHDL program to assert that the correct results were being produced by the pseudo random pattern generator.
The encryption core section was a very complex section to test thoroughly. However, utilizing another person’s DES encryption program it was possible to generate a VHDL testbench that could be easily altered to a different plaintext or different key schedule, providing the ability to test numerous different inputs. Although it has not been completely proven that the encryption core is completely correct, it has been thoroughly tested with over 20000 encryptions verified against an independently created DES implementation. Because of the extreme inter-relation from any single input bit to the output, it is very unlikely that the encryption core harbors any unknown errors.
7 Results and Conclusions

7.1 Performance

If this model is synthesized the target performance is that it would be able to run at over 16 MHz. If the chip is assumed to operate at 16 MHz, we can make some measurements of time needed to completely cover the keyspace. As one key should be retired per clock cycle due to the pipeline architecture of the chip a throughput of $2^{26}$ keys per second would be reached. At this rate the entire keyspace could be searched by four processors in only 34 years. As this is a problem that can be worked on easily in parallel, the key space could be divided up to be worked on by multiple processors in parallel. As it scales completely linearly, if $2^{24}$ processors are working on it, the maximum allowed with my design, the whole keyspace can be explored in only 4 minutes. However at that scale detection and replacement of failed parts becomes a much more difficult task. In addition the total cost for a machine of this scale is too high for most possible users. A more likely scenario would be a set of 4096 processors working in parallel to cover the keyspace in only a little over 12 days. If a full custom chip was used, a much higher speed would be possible, and would further reduce the amount of time that would be needed.
7.2 Future areas

There are a number of different areas that this work could branch into. One is the possibility of further optimizing the design for use in an FPGA. One possibility is utilizing some of the VHDL code for programming FPGA's to crack a specific DES message, thereby eliminating most of the “Front End” section of the chip. Another would be optimizing the chip to be run in a parallel situation, setting some of the key bits to be constant on each of the chips.

Another possible avenue that could be explored is utilizing the VHDL code for creating a full custom chip. A full custom layout would allow for faster performance,
but cost of a full custom chip would be greater, as well as the loss of flexibility that is provided with a FPGA.

Another avenue that could be explored is the possible differences in architecture for a single pipeline architecture as I have created here, versus a dual pipeline architecture, versus a non-pipeline architecture. Each would have its pros and cons, including cost considerations. The dual pipeline architecture would have twice the throughput of the single pipeline, however when cost would be factored in, it has not been determined which of the architectures would be best if all were evaluated on a normalized basis, such as in terms of keys per second per dollar.


3. M. Fischer, “How to implement the Data Encryption Standard (DES)”
   ftp://sable.ox.ac.uk/pub/crypto/DES/des-how-to.txt


11. A. Haberlach. “DES Challenge at OSU”

http://www.engr.orst.edu/~haberlad/desch
Appendix A: S-Box tables

(All values in decimal format)

Indexes are done in standard DES style, so that the row is determined by the first and last bit, the column is determined by the second through fifth bits. [1] [3]

<table>
<thead>
<tr>
<th>S-Box 1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>13</td>
<td>1</td>
<td>2</td>
<td>15</td>
<td>11</td>
<td>8</td>
<td>3</td>
<td>10</td>
<td>6</td>
<td>12</td>
<td>5</td>
<td>9</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>15</td>
<td>7</td>
<td>4</td>
<td>14</td>
<td>2</td>
<td>13</td>
<td>1</td>
<td>10</td>
<td>6</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>14</td>
<td>8</td>
<td>13</td>
<td>6</td>
<td>2</td>
<td>11</td>
<td>15</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>12</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>11</td>
<td>3</td>
<td>14</td>
<td>10</td>
<td>0</td>
<td>6</td>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S-Box 2</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>1</td>
<td>8</td>
<td>14</td>
<td>6</td>
<td>11</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>7</td>
<td>2</td>
<td>13</td>
<td>12</td>
<td>0</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>13</td>
<td>4</td>
<td>7</td>
<td>15</td>
<td>2</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>14</td>
<td>7</td>
<td>11</td>
<td>10</td>
<td>4</td>
<td>13</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>12</td>
<td>6</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>8</td>
<td>10</td>
<td>1</td>
<td>3</td>
<td>15</td>
<td>4</td>
<td>2</td>
<td>11</td>
<td>6</td>
<td>7</td>
<td>12</td>
<td>0</td>
<td>5</td>
<td>14</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S-Box 3</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td>9</td>
<td>14</td>
<td>6</td>
<td>3</td>
<td>15</td>
<td>5</td>
<td>1</td>
<td>13</td>
<td>12</td>
<td>7</td>
<td>11</td>
<td>4</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>7</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>14</td>
<td>12</td>
<td>11</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>6</td>
<td>4</td>
<td>9</td>
<td>8</td>
<td>15</td>
<td>3</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>5</td>
<td>10</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>10</td>
<td>13</td>
<td>0</td>
<td>6</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>15</td>
<td>14</td>
<td>3</td>
<td>11</td>
<td>5</td>
<td>2</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S-Box 4</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>13</td>
<td>14</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>11</td>
<td>12</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>8</td>
<td>11</td>
<td>5</td>
<td>6</td>
<td>15</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>12</td>
<td>1</td>
<td>10</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>6</td>
<td>9</td>
<td>0</td>
<td>12</td>
<td>11</td>
<td>7</td>
<td>13</td>
<td>15</td>
<td>1</td>
<td>3</td>
<td>14</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>15</td>
<td>0</td>
<td>6</td>
<td>10</td>
<td>1</td>
<td>13</td>
<td>8</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>11</td>
<td>12</td>
<td>7</td>
<td>2</td>
<td>14</td>
</tr>
</tbody>
</table>

44
### S-Box 5

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>12</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>6</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td>15</td>
<td>13</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>11</td>
<td>2</td>
<td>12</td>
<td>4</td>
<td>7</td>
<td>13</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>15</td>
<td>10</td>
<td>3</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>10</td>
<td>13</td>
<td>7</td>
<td>8</td>
<td>15</td>
<td>9</td>
<td>12</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>8</td>
<td>12</td>
<td>7</td>
<td>1</td>
<td>14</td>
<td>2</td>
<td>13</td>
<td>6</td>
<td>15</td>
<td>0</td>
<td>9</td>
<td>10</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

### S-Box 6

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>1</td>
<td>10</td>
<td>15</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>0</td>
<td>13</td>
<td>3</td>
<td>4</td>
<td>14</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>15</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>12</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>13</td>
<td>14</td>
<td>0</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>14</td>
<td>15</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>10</td>
<td>1</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>12</td>
<td>9</td>
<td>5</td>
<td>15</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

### S-Box 7

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>11</td>
<td>2</td>
<td>14</td>
<td>15</td>
<td>0</td>
<td>8</td>
<td>13</td>
<td>3</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>0</td>
<td>11</td>
<td>7</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>10</td>
<td>14</td>
<td>3</td>
<td>5</td>
<td>12</td>
<td>2</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>11</td>
<td>13</td>
<td>12</td>
<td>3</td>
<td>7</td>
<td>14</td>
<td>10</td>
<td>15</td>
<td>6</td>
<td>8</td>
<td>0</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>11</td>
<td>13</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td>5</td>
<td>0</td>
<td>15</td>
<td>14</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

### S-Box 8

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>6</td>
<td>15</td>
<td>11</td>
<td>1</td>
<td>10</td>
<td>9</td>
<td>3</td>
<td>14</td>
<td>5</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>15</td>
<td>13</td>
<td>8</td>
<td>10</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>0</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>11</td>
<td>4</td>
<td>1</td>
<td>9</td>
<td>12</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>10</td>
<td>13</td>
<td>15</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>14</td>
<td>7</td>
<td>4</td>
<td>10</td>
<td>8</td>
<td>13</td>
<td>15</td>
<td>12</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Appendix B: VHDL Code
entity deskey_check is
  port (clk in bit,
        key_fnd out bit,
        data_in in bit_vector(1 to 8),
        write in bit,
        addr_in in bit_vector(1 to 5),
        key_out out bit_vector(1 to 8),
        reset in bit);
begin
end deskey_check;

architecture deskey_check_struct of deskey_check is
  signal comparetext bit_vector(1 to 64);
  signal ciphertext bit_vector(1 to 64);
  signal plaintext bit_vector(1 to 64);
  signal keyspace bit_vector(1 to 24);
  signal key bit_vector(1 to 56);
  signal key_c_out bit_vector(1 to 56);

component dk_frontend
  port (data_in in bit_vector(1 to 8),
        addr_in in bit_vector(1 to 5),
        write in bit,
        keyspace OUT bit_vector(1 to 24),
        ptext OUT bit_vector(1 to 64),
        ctext OUT bit_vector(1 to 64));
end component;

component dk_core
  port (clk IN bit,
        key_IN bit_vector(1 to 56),
        ptext IN bit_vector(1 to 64),
        ctext OUT bit_vector(1 to 64),
        key_OUT OUT bit_vector(1 to 56));
end component;

component dk_keygen
  port (keyspace IN bit_vector(1 to 24),
        clk IN bit,
        reset IN bit,
        done out bit,
        key OUT bit_vector(1 to 56));
end component;

component dk_backend
  port (ciphertext, comparetext: in bit_vector(1 to 64),
        key_in: in bit_vector(1 to 56),
        addr: in bit_vector(1 to 3),
        reset in bit,
        key_fnd out bit,
        key_out out bit_vector(1 to 8));
end component;

begin
  FRONT dk_frontend
  port map (data_in, addr_in, write, keyspace, plaintext, comparetext);

  CORE : dk_core
  port map (clk, key, plaintext, ciphertext, key_c_out);

  KEYGEN dk_keygen
port map (keyspace, clk, reset, done, key);

BACKEND  dk_backend
  port map (ciphertext, comparetext, key_c_out,
           addr_in(3 to 5), reset, key_fnd, key_out);

end deskey_check_struct;

entity dk_core is
port (clk in bit;
key in bit_vector(1 to 56);
ptext in bit_vector(1 to 64);
ctxt in bit_vector(1 to 64);
key_out out bit_vector(1 to 56));
begin
end dk_core;

architecture dk_core_structure of dk_core is
signal ptext_perm bit_vector(1 to 64);
signal data_out1 bit_vector(1 to 64);
signal data_out2 bit_vector(1 to 64);
signal data_out3 bit_vector(1 to 64);
signal data_out4 bit_vector(1 to 64);
signal data_out5 bit_vector(1 to 64);
signal data_out6 bit_vector(1 to 64);
signal data_out7 bit_vector(1 to 64);
signal data_out8 bit_vector(1 to 64);
signal data_out9 bit_vector(1 to 64);
signal data_out10 bit_vector(1 to 64);
signal data_out11 bit_vector(1 to 64);
signal data_out12 bit_vector(1 to 64);
signal data_out13 bit_vector(1 to 64);
signal data_out14 bit_vector(1 to 64);
signal data_out15 bit_vector(1 to 64);
signal data_out16 bit_vector(1 to 64);

signal key_out0 bit_vector(1 to 56);
signal key_out1 bit_vector(1 to 56);
signal key_out2 bit_vector(1 to 56);
signal key_out3 bit_vector(1 to 56);
signal key_out4 bit_vector(1 to 56);
signal key_out5 bit_vector(1 to 56);
signal key_out6 bit_vector(1 to 56);
signal key_out7 bit_vector(1 to 56);
signal key_out8 bit_vector(1 to 56);
signal key_out9 bit_vector(1 to 56);
signal key_out10 bit_vector(1 to 56);
signal key_out11 bit_vector(1 to 56);
signal key_out12 bit_vector(1 to 56);
signal key_out13 bit_vector(1 to 56);
signal key_out14 bit_vector(1 to 56);
signal key_out15 bit_vector(1 to 56);
signal key_out16 bit_vector(1 to 56);

signal subkey1 bit_vector(1 to 48);
signal subkey2 bit_vector(1 to 48);
signal subkey3 bit_vector(1 to 48);
signal subkey4 bit_vector(1 to 48);
signal subkey5 bit_vector(1 to 48);
signal subkey6 bit_vector(1 to 48);
signal subkey7 bit_vector(1 to 48);
signal subkey8 bit_vector(1 to 48);
signal subkey9 bit_vector(1 to 48);
signal subkey10 bit_vector(1 to 48);
signal subkey11 bit_vector(1 to 48);
signal subkey12 bit_vector(1 to 48);
signal subkey13 bit_vector(1 to 48);
signal subkey14 bit_vector(1 to 48);
signal subkey15 bit_vector(1 to 48);
signal subkey16 bit_vector(1 to 48);
component dk_keyround1
    port (clk in bit;
        key_in : in bit_vector(1 to 56);
        subkey : out bit_vector(1 to 48);
        key_out : out bit_vector(1 to 56));
    end component;

component dk_keyround2
    port (clk in bit;
        key_in in bit_vector(1 to 56);
        subkey : out bit_vector(1 to 48);
        key_out : out bit_vector(1 to 56));
    end component;

component dk_round
    port (clk bit;
        data_in in bit_vector(1 to 64);
        subkey in bit_vector(1 to 48);
        data_out: out bit_vector(1 to 64));
    end component;

component dk_keyfp
    port (input in bit_vector(1 to 56);
        output out bit_vector(1 to 56));
    end component;

component dk_keyip
    port (input in bit_vector(1 to 56);
        output out bit_vector(1 to 56));
    end component;

component dk_datafp
    port (input in bit_vector(1 to 64);
        output out bit_vector(1 to 64));
    end component;

component dk_dataip
    port (input in bit_vector(1 to 64);
        output out bit_vector(1 to 64));
    end component;

begin
    -- each of the rounds for generating the subkeys.
    keyrl : dk_keyround1
        port map (clk, key_out0, subkey1, key_out1);
    keyr2 : dk_keyround1
        port map (clk, key_out1, subkey2, key_out2);
    keyr3 : dk_keyround2
        port map (clk, key_out0, subkey3, key_out3);
    keyr4 : dk_keyround2
        port map (clk, key_out1, subkey4, key_out4);
    keyr5 : dk_keyround2
        port map (clk, key_out2, subkey5, key_out5);
    keyr6 : dk_keyround2
        port map (clk, key_out3, subkey6, key_out6);
    keyr7 : dk_keyround2
        port map (clk, key_out4, subkey7, key_out7);
    keyr8 : dk_keyround2
        port map (clk, key_out5, subkey8, key_out8);
    keyr9 : dk_keyround1
        port map (clk, key_out0, subkey9, key_out9);
    keyrl0 : dk_keyround2
        port map (clk, key_out1, subkey10, key_out10);
    keyrl1 : dk_keyround2
        port map (clk, key_out2, subkey11, key_out11);
    keyrl2 : dk_keyround2
        port map (clk, key_out3, subkey12, key_out12);
    keyrl3 : dk_keyround2
        port map (clk, key_out4, subkey13, key_out13);
    keyrl4 : dk_keyround2
        port map (clk, key_out5, subkey14, key_out14);
keyrl5 : dk_keyround2
    port map (clk, key_out14, subkey15, key_out15);
keyrl6 : dk_keyround1
    port map (clk, key_out15, subkey16, key_out16);

-- each of the rounds for doing the encryption.
round1 : dk_round
    port map (clk, ptext_perm, subkey1, data_out1);
round2 : dk_round
    port map (clk, data_out1, subkey2, data_out2);
round3 : dk_round
    port map (clk, data_out2, subkey3, data_out3);
round4 : dk_round
    port map (clk, data_out3, subkey4, data_out4);
round5 : dk_round
    port map (clk, data_out4, subkey5, data_out5);
round6 : dk_round
    port map (clk, data_out5, subkey6, data_out6);
round7 : dk_round
    port map (clk, data_out6, subkey7, data_out7);
round8 : dk_round
    port map (clk, data_out7, subkey8, data_out8);
round9 : dk_round
    port map (clk, data_out8, subkey9, data_out9);
round10 : dk_round
    port map (clk, data_out9, subkey10, data_out10);
round11 : dk_round
    port map (clk, data_out10, subkey11, data_out11);
round12 : dk_round
    port map (clk, data_out11, subkey12, data_out12);
round13 : dk_round
    port map (clk, data_out12, subkey13, data_out13);
round14 : dk_round
    port map (clk, data_out13, subkey14, data_out14);
round15 : dk_round
    port map (clk, data_out14, subkey15, data_out15);
round16 : dk_round
    port map (clk, data_out15, subkey16, data_out16);

-- initial permutation of data (before any rounds)
initperm : dk_dataip
    port map (ptext, ptext_perm);

-- Final permutation of data (after all rounds)
finalperm : dk_datafp
    port map (data_out16, ctext);

-- initial permutation of key (before any rounds)
keyip : dk_keyip
    port map (key, key_out0);

-- final permutation of key (after all rounds)
keyfp : dk_keyfp
    port map (key_out16, key_out);

end dk_core_struct;
entity dk_backend is
  port (ciphertext in bit_vector(1 to 64);
         comparetext in bit_vector(1 to 64);
         key_in in bit_vector(1 to 56);
         addr in bit_vector(1 to 3);
         reset in bit;
         key_fnd out bit;
         key_out out bit_vector(1 to 8));
begin
  end dk_backend;

architecture dk_backend_beh of dk_backend is
begin

  -- compareresult process (ciphertext, comparetext, reset)
  -- we have a match!
  save_key := key_in;
  key_fnd := '1';
end if;
end process compareresult;

  -- output a section of the saved key
  -- based on the address coming in.
  case addr is
    when '000' => key_out := save_key(1 to 8);
    when '001' => key_out := save_key(9 to 16);
    when '010' => key_out := save_key(17 to 24);
    when '011' => key_out := save_key(25 to 32);
    when '100' => key_out := save_key(33 to 40);
    when '101' => key_out := save_key(41 to 48);
    when '110' => key_out := save_key(49 to 56);
    when others => key_out := save_key(49 to 56);
  end case;
end process outputresult;
end dk_backend_beh;
entity dk_frontend is
  port (
data: in bit_vector(1 to 8);
  addr: in bit_vector(1 to 5);
  write: in bit;
  keyspace: OUT bit_vector(1 to 24);
  ptext: OUT bit_vector(1 to 64);
  ctext: OUT bit_vector(1 to 64));
begin
end dk_frontend;

architecture dk_frontend_beh of dk_frontend is
begin
  latchdata process (write, addr, data)
  begin
    if (write = '1') then
      case addr is
        when "00000" => keyspace (1 to 8) := data;
        when "00001" => keyspace (9 to 16) := data;
        when "00010" => keyspace (17 to 24) := data;
        when "00011" => null;
        when "01000" => ctext (1 to 8) := data;
        when "01001" => ctext (9 to 16) := data;
        when "01010" => ctext (17 to 24) := data;
        when "01011" => ctext (25 to 32) := data;
        when "01100" => ctext (33 to 40) := data;
        when "01101" => ctext (41 to 48) := data;
        when "01110" => ctext (49 to 56) := data;
        when "01111" => ctext (57 to 64) := data;
        when "10000" => ptext (1 to 8) := data;
        when "10001" => ptext (9 to 16) := data;
        when "10010" => ptext (17 to 24) := data;
        when "10011" => ptext (25 to 32) := data;
        when "10100" => ptext (33 to 40) := data;
        when "10101" => ptext (41 to 48) := data;
        when "10110" => ptext (49 to 56) := data;
        when "10111" => ptext (57 to 64) := data;
        when others => null;
      end case;
    end if;
  end process latchdata;
end dk_frontend_beh;
--- Project  Tom Oelke's Thesis
--- File name  dk_keygen.vhd
--- Title  DES key checking program/chip key generator

--- Revisions :  
--- Date  Author  Revision  Comments
--- 4/10/97  T. Oelke  Rev. 1  creation

entity dk_keygen is
  port (keyspace: in bit_vector(1 to 24);
  clk  in bit;
  reset in bit;
  done  out bit;
  key  out bit_vector(1 to 56));
begin
  end dk_keygen;

architecture dk_keygen_beh of dk_keygen is
  signal curkeyspace: bit_vector(1 to 24);
  signal pgen : bit_vector(1 to 32);
begin
  loadkeyspace  process (p gen, keyspace)
  begin
    -- Load the key space into the chip when we come back around to
    -- the beginning.
    -- !! May need to change to ANDs depending on how it synthesizes.
    if (p gen = X'FFFFFFFF') then
      curkeyspace := keyspace;
    end if;
    if (reset = '1') then
      p gen := X'FFFFFFFF';
    end if;
  end process loadkeyspace;

  genkey  process (clk, reset)
  begin
    -- do the pseudo random generation thing.
    if (clk'event and clk = '1') then
      p gen(2 to 32) := p gen(1 to 31);
      p gen(1) := p gen(32) xor p gen(31) xor p gen(30) xor p gen(10);
    end if;
    if (reset = '1') then
      p gen := X'FFFFFFFF';
    end if;
  end process genkey;

  outputkey  process (p gen, curkeyspace)
  begin
    key(1 to 24) := curkeyspace;
    key(25 to 56) := p gen;
  end process outputkey;
end dk_keygen_beh;
entity dk_dataip is
  port (input . in bit_vector(1 to 64);
        output out bit_vector(1 to 64));
begin
end dk_dataip;

architecture dk_dataip_beh of dk_dataip is
begin
  -- Initial Permutation of plaintext
  -- inpu...
output(53) := input(29);
output(54) := input(21);
output(55) := input(13);
output(56) := input(5);
output(57) := input(63);
output(58) := input(55);
output(59) := input(47);
output(60) := input(39);
output(61) := input(31);
output(62) := input(23);
output(63) := input(15);
output(64) := input(7);
end process initpermute;
end dk_dataip_beh;
entity dk_datafp is
  port (input in bit_vector(1 to 64);
        output out bit_vector(1 to 64));
begin
end dk_datafp;

architecture dk_datafp_beh of dk_datafp is
begin
  -- Final permutation of data
  -- This also does the 'uncrossing'
  finalpermute process (input)
  begin
    -- Autogenerated code by datafp_gen.pl
    output(1) := input(8);
    output(2) := input(40);
    output(3) := input(16);
    output(4) := input(48);
    output(5) := input(24);
    output(6) := input(56);
    output(7) := input(32);
    output(8) := input(64);
    output(9) := input(7);
    output(10) := input(39);
    output(11) := input(15);
    output(12) := input(47);
    output(13) := input(23);
    output(14) := input(55);
    output(15) := input(31);
    output(16) := input(63);
    output(17) := input(6);
    output(18) := input(38);
    output(19) := input(14);
    output(20) := input(46);
    output(21) := input(22);
    output(22) := input(54);
    output(23) := input(30);
    output(24) := input(62);
    output(25) := input(5);
    output(26) := input(37);
    output(27) := input(13);
    output(28) := input(45);
    output(29) := input(21);
    output(30) := input(53);
    output(31) := input(29);
    output(32) := input(61);
    output(33) := input(41);
    output(34) := input(36);
    output(35) := input(12);
    output(36) := input(44);
    output(37) := input(20);
    output(38) := input(52);
    output(39) := input(28);
    output(40) := input(60);
    output(41) := input(3);
    output(42) := input(35);
    output(43) := input(11);
    output(44) := input(43);
    output(45) := input(19);
    output(46) := input(51);
    output(47) := input(27);
    output(48) := input(59);
    output(49) := input(2);
    output(50) := input(34);
    output(51) := input(10);
  end process;
end dk_datafp_beh;
output(52) := input(42);
output(53) := input(18);
output(54) := input(50);
output(55) := input(26);
output(56) := input(58);
output(57) := input(1);
output(58) := input(33);
output(59) := input(9);
output(60) := input(41);
output(61) := input(17);
output(62) := input(49);
output(63) := input(25);
output(64) := input(57);
-- ) end autogenerated code
end process finalpermute;
end dk_datafp_beh;
entity dk_keyip is
    port (input  in  bit_vector(1 to 56);
          output out bit_vector(1 to 56));
begin
end dk_keyip;

architecture dk_keyip_beh of dk_keyip is
begin
    -- keyinitpermute does the initial permutation of the key.
    keyinitpermute process (input)
    begin
        -- autogenerated code by keyinitpermute.pl {
        output(1) := input(50);
        output(2) := input(43);
        output(3) := input(36);
        output(4) := input(29);
        output(5) := input(22);
        output(6) := input(15);
        output(7) := input(8);
        output(8) := input(1);
        output(9) := input(51);
        output(10) := input(44);
        output(11) := input(37);
        output(12) := input(30);
        output(13) := input(23);
        output(14) := input(16);
        output(15) := input(9);
        output(16) := input(2);
        output(17) := input(52);
        output(18) := input(45);
        output(19) := input(38);
        output(20) := input(31);
        output(21) := input(24);
        output(22) := input(17);
        output(23) := input(10);
        output(24) := input(3);
        output(25) := input(53);
        output(26) := input(46);
        output(27) := input(39);
        output(28) := input(32);
        output(29) := input(56);
        output(30) := input(49);
        output(31) := input(42);
        output(32) := input(35);
        output(33) := input(28);
        output(34) := input(21);
        output(35) := input(14);
        output(36) := input(7);
        output(37) := input(55);
        output(38) := input(48);
        output(39) := input(41);
        output(40) := input(34);
        output(41) := input(27);
        output(42) := input(20);
        output(43) := input(13);
        output(44) := input(6);
        output(45) := input(54);
        output(46) := input(47);
        output(47) := input(40);
        output(48) := input(33);
        output(49) := input(26);
        output(50) := input(19);
        output(51) := input(12);
        output(52) := input(5);
    end keyinitpermute;
end dk_keyip_beh;
output(53) := input(25);
output(54) := input(18);
output(55) := input(11);
output(56) := input(4);
-- ) end autogenerated code
end process keyinitpermute;
end dk_keyip_beh;
entity dk_keyfp is
  port (input in bit_vector(1 to 56);
       output out bit_vector(1 to 56));
begin
end dk_keyfp;

architecture dk_keyfp_beh of dk_keyfp is
begin
  -- keyfp does the final permutation of the key.
  -- Essentially an undoing of the initial permutation.
  keyfp process (input)
  begin
    -- autogenerated code by keyfp.pl
    output(50) := input(1);
    output(43) := input(2);
    output(36) := input(3);
    output(29) := input(4);
    output(22) := input(5);
    output(15) := input(6);
    output(8) := input(7);
    output(1) := input(8);
    output(51) := input(9);
    output(44) := input(10);
    output(37) := input(11);
    output(30) := input(12);
    output(23) := input(13);
    output(16) := input(14);
    output(9) := input(15);
    output(21) := input(16);
    output(52) := input(17);
    output(45) := input(18);
    output(38) := input(19);
    output(31) := input(20);
    output(24) := input(21);
    output(17) := input(22);
    output(10) := input(23);
    output(3) := input(24);
    output(53) := input(25);
    output(46) := input(26);
    output(39) := input(27);
    output(32) := input(28);
    output(56) := input(29);
    output(49) := input(30);
    output(42) := input(31);
    output(35) := input(32);
    output(28) := input(33);
    output(21) := input(34);
    output(14) := input(35);
    output(7) := input(36);
    output(55) := input(37);
    output(48) := input(38);
    output(41) := input(39);
    output(34) := input(40);
    output(27) := input(41);
    output(20) := input(42);
    output(13) := input(43);
    output(6) := input(44);
    output(54) := input(45);
    output(47) := input(46);
    output(40) := input(47);
    output(33) := input(48);
    output(26) := input(49);
    output(19) := input(50);
    output(12) := input(51);
output(5) := input(52);
output(25) := input(53);
output(18) := input(54);
output(11) := input(55);
output(4)  := input(56);

-- ) end autogenerated code
end process keyfp;
end dk_keyfp_beh;
entity dk_round is
  port (clk in bit;
        data_in in bit_vector(1 to 64);
        subkey in bit_vector(1 to 48);
        data_out out bit_vector(1 to 64));
begin
end dk_round;

architecture dk_round_beh of dk_round is
  component dk_sbox1
    port (i bit_vector(1 to 6);
          o bit_vector(1 to 4));
  end component;
  component dk_sbox2
    port (i bit_vector(1 to 6);
          o bit_vector(1 to 4));
  end component;
  component dk_sbox3
    port (i bit_vector(1 to 6);
          o bit_vector(1 to 4));
  end component;
  component dk_sbox4
    port (i bit_vector(1 to 6);
          o bit_vector(1 to 4));
  end component;
  component dk_sbox5
    port (i bit_vector(1 to 6);
          o bit_vector(1 to 4));
  end component;
  component dk_sbox6
    port (i bit_vector(1 to 6);
          o bit_vector(1 to 4));
  end component;
  component dk_sbox7
    port (i bit_vector(1 to 6);
          o bit_vector(1 to 4));
  end component;
  component dk_sbox8
    port (i bit_vector(1 to 6);
          o bit_vector(1 to 4));
  end component;
signal sbox_in bit_vector(1 to 48);
signal sbox_out bit_vector(1 to 32);
begin
  s1 dk_sbox1
    port map (sbox_in(1 to 6), sbox_out(1 to 4));
  s2 dk_sbox2
    port map (sbox_in(7 to 12), sbox_out(5 to 8));
  s3 dk_sbox3
    port map (sbox_in(13 to 18), sbox_out(9 to 12));
  s4 dk_sbox4
    port map (sbox_in(19 to 24), sbox_out(13 to 16));
  s5 dk_sbox5
    port map (sbox_in(25 to 30), sbox_out(17 to 20));
  s6 dk_sbox6
    port map (sbox_in(31 to 36), sbox_out(21 to 24));
  s7 dk_sbox7
    port map (sbox_in(37 to 42), sbox_out(25 to 28));
  s8 dk_sbox8
    port map (sbox_in(43 to 48), sbox_out(29 to 32));
calc_sbox_in process (subkey, data_in)
begin

-- expansion and xor
-- autogenerated code by expansion.pl
sbox_in(1) = subkey(1) xor data_in(64);
sbox_in(2) = subkey(2) xor data_in(33);
sbox_in(3) = subkey(3) xor data_in(34);
sbox_in(4) = subkey(4) xor data_in(35);
sbox_in(5) = subkey(5) xor data_in(36);
sbox_in(6) = subkey(6) xor data_in(37);
sbox_in(7) = subkey(7) xor data_in(36);
sbox_in(8) = subkey(8) xor data_in(37);
sbox_in(9) = subkey(9) xor data_in(38);
sbox_in(10) = subkey(10) xor data_in(39);
sbox_in(11) = subkey(11) xor data_in(40);
sbox_in(12) = subkey(12) xor data_in(41);
sbox_in(13) = subkey(13) xor data_in(40);
sbox_in(14) = subkey(14) xor data_in(41);
sbox_in(15) = subkey(15) xor data_in(42);
sbox_in(16) = subkey(16) xor data_in(43);
sbox_in(17) = subkey(17) xor data_in(44);
sbox_in(18) = subkey(18) xor data_in(45);
sbox_in(19) = subkey(19) xor data_in(44);
sbox_in(20) = subkey(20) xor data_in(45);
sbox_in(21) = subkey(21) xor data_in(46);
sbox_in(22) = subkey(22) xor data_in(47);
sbox_in(23) = subkey(23) xor data_in(48);
sbox_in(24) = subkey(24) xor data_in(49);
sbox_in(25) = subkey(25) xor data_in(48);
sbox_in(26) = subkey(26) xor data_in(49);
sbox_in(27) = subkey(27) xor data_in(50);
sbox_in(28) = subkey(28) xor data_in(51);
sbox_in(29) = subkey(29) xor data_in(52);
sbox_in(30) = subkey(30) xor data_in(53);
sbox_in(31) = subkey(31) xor data_in(52);
sbox_in(32) = subkey(32) xor data_in(53);
sbox_in(33) = subkey(33) xor data_in(54);
sbox_in(34) = subkey(34) xor data_in(55);
sbox_in(35) = subkey(35) xor data_in(56);
sbox_in(36) = subkey(36) xor data_in(57);
sbox_in(37) = subkey(37) xor data_in(56);
sbox_in(38) = subkey(38) xor data_in(57);
sbox_in(39) = subkey(39) xor data_in(58);
sbox_in(40) = subkey(40) xor data_in(59);
sbox_in(41) = subkey(41) xor data_in(60);
sbox_in(42) = subkey(42) xor data_in(61);
sbox_in(43) = subkey(43) xor data_in(60);
sbox_in(44) = subkey(44) xor data_in(61);
sbox_in(45) = subkey(45) xor data_in(62);
sbox_in(46) = subkey(46) xor data_in(63);
sbox_in(47) = subkey(47) xor data_in(64);
sbox_in(48) = subkey(48) xor data_in(33);
-- ) end autogenerated code
end process calc_sbox_in;
calc_output process (data_in, sbox_out, clk)
begin
if (clk = '1' and clk'event) then
  -- Permutation and xor
  -- autogenerated code by round-outputgen.pl
  data_out(33) = sbox_out(16) xor data_in(1);
data_out(34) = sbox_out(7) xor data_in(2);
data_out(35) = sbox_out(20) xor data_in(3);
data_out(36) = sbox_out(21) xor data_in(4);
data_out(37) = sbox_out(29) xor data_in(5);
data_out(38) = sbox_out(12) xor data_in(6);
data_out(39) = sbox_out(28) xor data_in(7);
data_out(40) = sbox_out(17) xor data_in(8);
data_out(41) = sbox_out(1) xor data_in(9);
data_out(42) = sbox_out(15) xor data_in(10);
data_out(43) = sbox_out(23) xor data_in(11);
data_out(44) = sbox_out(28) xor data_in(12);
data_out(45) = sbox_out(5) xor data_in(13);
data_out(46) = sbox_out(19) xor data_in(14);
data_out(47) = sbox_out(31) xor data_in(15);
data_out(48) = sbox_out(10) xor data_in(16);
data_out(49) = sbox_out(2) xor data_in(17);
end process calc_output;
data_out(50) := sbox_out(8) xor data_in(18);
data_out(51) := sbox_out(24) xor data_in(19);
data_out(52) := sbox_out(14) xor data_in(20);
data_out(53) := sbox_out(32) xor data_in(21);
data_out(54) := sbox_out(27) xor data_in(22);
data_out(55) := sbox_out(1) xor data_in(23);
data_out(56) := sbox_out(9) xor data_in(24);
data_out(57) := sbox_out(19) xor data_in(25);
data_out(58) := sbox_out(13) xor data_in(26);
data_out(59) := sbox_out(30) xor data_in(27);
data_out(60) := sbox_out(6) xor data_in(28);
data_out(61) := sbox_out(22) xor data_in(29);
data_out(62) := sbox_out(11) xor data_in(30);
data_out(63) := sbox_out(4) xor data_in(31);
data_out(64) := sbox_out(25) xor data_in(32);
end if;
end process calc_output;

-- new left side (bits 1-32) becomes old right hand side (bits 33-64)
left_right_trans process (clk)
begin
if (clk = '1' and clk'event) then
    data_out(1 to 32) := data_in(33 to 64);
end if;
end process left_right_trans;
end dk_round_beh;
entity dk_keyround1 is
  port (clk : in bit;
        key_in in bit_vector(1 to 56);
        subkey out bit_vector(1 to 48);
        key_out out bit_vector(1 to 56));
begin
  subkeygen
  begin
    -- select and permute the bits to be used for the subkey
    -- autogenerated by subkeygen.pl
    subkey(1) := key_out_tmp(14);
    subkey(2) := key_out_tmp(17);
    subkey(3) := key_out_tmp(11);
    subkey(4) := key_out_tmp(24);
    subkey(5) := key_out_tmp(1);
    subkey(6) := key_out_tmp(5);
    subkey(7) := key_out_tmp(3);
    subkey(8) := key_out_tmp(28);
    subkey(9) := key_out_tmp(15);
    subkey(10) := key_out_tmp(6);
    subkey(11) := key_out_tmp(21);
    subkey(12) := key_out_tmp(10);
    subkey(13) := key_out_tmp(23);
    subkey(14) := key_out_tmp(19);
    subkey(15) := key_out_tmp(12);
    subkey(16) := key_out_tmp(4);
    subkey(17) := key_out_tmp(26);
    subkey(18) := key_out_tmp(8);
    subkey(19) := key_out_tmp(16);
    subkey(20) := key_out_tmp(7);
    subkey(21) := key_out_tmp(27);
    subkey(22) := key_out_tmp(20);
    subkey(23) := key_out_tmp(13);
    subkey(24) := key_out_tmp(12);
    subkey(25) := key_out_tmp(41);
    subkey(26) := key_out_tmp(52);
    subkey(27) := key_out_tmp(31);
    subkey(28) := key_out_tmp(37);
    subkey(29) := key_out_tmp(47);
    subkey(30) := key_out_tmp(55);
    subkey(31) := key_out_tmp(30);
    subkey(32) := key_out_tmp(40);
    subkey(33) := key_out_tmp(51);
    subkey(34) := key_out_tmp(45);
    subkey(35) := key_out_tmp(33);
    subkey(36) := key_out_tmp(49);
    subkey(37) := key_out_tmp(44);
    subkey(38) := key_out_tmp(49);
    subkey(39) := key_out_tmp(39);
    subkey(40) := key_out_tmp(56);
    subkey(41) := key_out_tmp(34);
    subkey(42) := key_out_tmp(53);
    subkey(43) := key_out_tmp(46);
    subkey(44) := key_out_tmp(42);
    subkey(45) := key_out_tmp(50);
    subkey(46) := key_out_tmp(36);
    subkey(47) := key_out_tmp(29);
  end process subkeygen;
end dk_keyround1;
subkey(48) := key_out_tmp(32);
-- ) end autogenerated code
end process subkeygen;

producekey process (key_in)
begin
-- rotate 1 left of each of the halves (28 bits) of the key.
key_out_tmp(1 to 27) := key_in (2 to 28);
key_out_tmp(28) := key_in (1);

key_out_tmp(29 to 55) := key_in (30 to 56);
key_out_tmp(56) := key_in (29);
end process producekey;

latchkey process (clk, key_out_tmp)
begin
-- latch the new key on the output.
if (clk = '1' and clk'event) then
key_out := key_out_tmp;
end if;
end process latchkey;

end dk_keyround1_beh;
entity dk_keyround2 is
  port (clk in bit;
        key_in in bit_vector(1 to 56);
        subkey out bit_vector(1 to 48);
        key_out out bit_vector(1 to 56));
begin
  end dk_keyround2;

architecture dk_keyround2_beh of dk_keyround2 is
  signal key_out_tmp bit_vector(1 to 56);
begin
  subkeygen process (key_out_tmp)
  begin
    -- select and permute the bits to be used for the subkey
    -- autogenerated by subkeygen.pl 
    subkey(1) := key_out_tmp(14);
    subkey(2) := key_out_tmp(17);
    subkey(3) := key_out_tmp(11);
    subkey(4) := key_out_tmp(24);
    subkey(5) := key_out_tmp(1);
    subkey(6) := key_out_tmp(5);
    subkey(7) := key_out_tmp(3);
    subkey(8) := key_out_tmp(28);
    subkey(9) := key_out_tmp(15);
    subkey(10) := key_out_tmp(6);
    subkey(11) := key_out_tmp(21);
    subkey(12) := key_out_tmp(10);
    subkey(13) := key_out_tmp(23);
    subkey(14) := key_out_tmp(19);
    subkey(15) := key_out_tmp(12);
    subkey(16) := key_out_tmp(4);
    subkey(17) := key_out_tmp(26);
    subkey(18) := key_out_tmp(9);
    subkey(19) := key_out_tmp(16);
    subkey(20) := key_out_tmp(7);
    subkey(21) := key_out_tmp(27);
    subkey(22) := key_out_tmp(20);
    subkey(23) := key_out_tmp(13);
    subkey(24) := key_out_tmp(12);
    subkey(25) := key_out_tmp(41);
    subkey(26) := key_out_tmp(52);
    subkey(27) := key_out_tmp(31);
    subkey(28) := key_out_tmp(37);
    subkey(29) := key_out_tmp(47);
    subkey(30) := key_out_tmp(55);
    subkey(31) := key_out_tmp(30);
    subkey(32) := key_out_tmp(40);
    subkey(33) := key_out_tmp(51);
    subkey(34) := key_out_tmp(45);
    subkey(35) := key_out_tmp(33);
    subkey(36) := key_out_tmp(48);
    subkey(37) := key_out_tmp(44);
    subkey(38) := key_out_tmp(49);
    subkey(39) := key_out_tmp(39);
    subkey(40) := key_out_tmp(56);
    subkey(41) := key_out_tmp(34);
    subkey(42) := key_out_tmp(53);
    subkey(43) := key_out_tmp(46);
    subkey(44) := key_out_tmp(42);
    subkey(45) := key_out_tmp(50);
    subkey(46) := key_out_tmp(36);
    subkey(47) := key_out_tmp(29);
  end process subkeygen;
end dk_keyround2_beh;
subkey(48) := key_out_tmp(32);
-- ) end autogenerated code
end process subkeygen;

producekey . process (key_in)
begin
-- rotate 2 left of each of the halves (28 bits) of the key.
key_out_tmp(1 to 26) := key_in (3 to 28);
key_out_tmp(27 to 28) := key_in (1 to 2);
key_out_tmp(29 to 54) := key_in (31 to 56);
key_out_tmp(55 to 56) := key_in (29 to 30);
end process producekey;
latchkey process (clk, key_out_tmp)
begin
-- latch the new key on the output.
if (clk = '1' and clk'event) then
  key_out := key_out_tmp;
end if;
end process latchkey;
end dk_keyround2_beh;
entity dk_sboxl is
  port (input in bit_vector(1 to 6);
        output out bit_vector(1 to 4));
begin
end dk_sboxl;

architecture dk_sboxl_beh of dk_sboxl is
begin
  sbox process (input)
  begin
    case input is
      when '000000' => output := "1110";
      when '000001' => output := "0000";
      when '000010' => output := "0100";
      when '000011' => output := "1111";
      when '000100' => output := "1101";
      when '000101' => output := "0111";
      when '000110' => output := "0011";
      when '000111' => output := "0001";
      when '001000' => output := "0100";
      when '001001' => output := "1100";
      when '001010' => output := "1100";
      when '001011' => output := "0001";
      when '001100' => output := "1101";
      when '001101' => output := "1101";
      when '001110' => output := "0010";
      when '001111' => output := "0011";
      when '010000' => output := "0011";
      when '010001' => output := "1010";
      when '010010' => output := "1010";
      when '010011' => output := "0110";
      when '010100' => output := "0110";
      when '010101' => output := "1000";
      when '010110' => output := "0000";
      when '010111' => output := "0001";
      when '011000' => output := "1010";
      when '011001' => output := "1010";
      when '011010' => output := "0100";
      when '011011' => output := "1100";
      when '011100' => output := "0100";
      when '011101' => output := "0100";
      when '011110' => output := "0000";
      when '011111' => output := "0001";
      when '100000' => output := "1111";
      when '100001' => output := "1111";
      when '100010' => output := "0001";
      when '100011' => output := "1100";
      when '100100' => output := "1100";
      when '100101' => output := "1000";
      when '100110' => output := "1000";
      when '100111' => output := "0010";
      when '101000' => output := "0010";
      when '101001' => output := "0010";
      when '101010' => output := "0000";
      when '101011' => output := "0001";
      when '101100' => output := "0101";
      when '101101' => output := "0101";
      when '101110' => output := "0101";
      when '101111' => output := "0011";
      when '110000' => output := "0011";
      when '110001' => output := "0011";
      when '110010' => output := "1100";
      when '110011' => output := "1100";
      when '110100' => output := "1100";
      when '110101' => output := "1100";
      when '110110' => output := "1100";
      when '110111' => output := "1100";
      when '111000' => output := "0011";
      when '111001' => output := "0011";
      when '111010' => output := "1011";
      when '111011' => output := "1011";
      when '111100' => output := "1011";
      when '111101' => output := "1011";
      when '111110' => output := "1011";
      when '111111' => output := "1011";
    end case;
  end process sbox;
when '110100' => output := '1001';
when '110101' => output := '0011';
when '110110' => output := '0111';
when '110111' => output := '1110';
when '111000' => output := '0011';
when '111001' => output := '1010';
when '111010' => output := '1010';
when '111011' => output := '0000';
when '111100' => output := '0101';
when '111101' => output := '0110';
when '111110' => output := '0000';
when '111111' => output := '1101';
when others => null;
end case;
end process;
end dk_sbox1_beh;
entity dk_sbox2 is
  port (input in bit_vector(1 to 6);
       output out bit_vector(1 to 4));
begin
end dk_sbox2;

architecture dk_sbox2_beh of dk_sbox2 is
begin
  sbox process (input)
  begin
    -- sbox2 case input is
    when '000000' => output := "1111";
    when '000001' => output := "0011";
    when '000010' => output := "1101";
    when '000011' => output := "1110";
    when '000100' => output := "0001";
    when '000101' => output := "0100";
    when '000110' => output := "1110";
    when '000111' => output := "0111";
    when '001000' => output := "0110";
    when '001001' => output := "1111";
    when '001010' => output := "1011";
    when '001011' => output := "0010";
    when '001100' => output := "0011";
    when '001101' => output := "1000";
    when '001110' => output := "0100";
    when '001111' => output := "1101";
    when '010000' => output := "1001";
    when '010001' => output := "1100";
    when '010010' => output := "0111";
    when '010011' => output := "1010";
    when '010100' => output := "1100";
    when '010101' => output := "0110";
    when '010110' => output := "0000";
    when '010111' => output := "1011";
    when '011000' => output := "1010";
    when '011001' => output := "0101";
    when '011010' => output := "0001";
    when '011011' => output := "1001";
    when '011100' => output := "1111";
    when '011101' => output := "0100";
    when '011110' => output := "0010";
    when '011111' => output := "0101";
    when '100000' => output := "0000";
    when '100001' => output := "1101";
    when '100010' => output := "1110";
    when '100011' => output := "0100";
    when '100100' => output := "1011";
    when '100101' => output := "1100";
    when '100110' => output := "0000";
    when '100111' => output := "0011";
    when '101000' => output := "1100";
    when '101001' => output := "0110";
    when '101010' => output := "0001";
    when '101011' => output := "0010";
    when '101100' => output := "1001";
    when '101101' => output := "1101";
    when '101110' => output := "1010";
    when '101111' => output := "0110";
    when '110000' => output := "0101";
    when '110001' => output := "1011";
    when '110010' => output := "1000";
    when '110011' => output := "0110";
    when '110100' => output := "0001";
    when '110101' => output := "0101";
    when '110110' => output := "1000";
    when '110111' => output := "0010";
    when '111000' => output := "0011";
    when '111001' => output := "1111";
    when '111010' => output := "1110";
    when '111011' => output := "0100";
    when '111100' => output := "1010";
    when '111101' => output := "0111";
    when '111110' => output := "1111";
  end case;
end sbox;
when '110100' = output := '1100';
when '110101' = output := '0111';
when '110110' = output := '0110';
when '110111' = output := '1100';
when '111000' = output := '1001';
when '111001' = output := '0000';
when '111010' = output := '0011';
when '111011' = output := '0010';
when '111100' = output := '0010';
when '111101' = output := '1110';
when '111110' = output := '1111';
when '111111' = output := '1001';
when others = null;
end case;
end process;
end dk_sbox2_beh;
entity dk_sbox3 is
  port (input in bit_vector(1 to 6);
       output out bit_vector(1 to 4));
begin
end dk_sbox3;

architecture dk_sbox3_beh of dk_sbox3 is
begin
  sbox process (input)
  begin
    case input is
      when '000000' => output := "1010";
      when '000001' => output := "1101";
      when '000010' => output := "0000";
      when '000011' => output := "0111";
      when '000100' => output := "1001";
      when '000101' => output := "0000";
      when '000110' => output := "1110";
      when '000111' => output := "1001";
      when '001000' => output := "0110";
      when '001001' => output := "0011";
      when '001010' => output := "0011";
      when '001011' => output := "0100";
      when '001100' => output := "0110";
      when '001101' => output := "0110";
      when '001110' => output := "0110";
      when '001111' => output := "0110";
      when '010000' => output := "0001";
      when '010001' => output := "0010";
      when '010010' => output := "1101";
      when '010011' => output := "1000";
      when '010100' => output := "0100";
      when '010101' => output := "0100";
      when '010110' => output := "0110";
      when '010111' => output := "1100";
      when '011000' => output := "1111";
      when '011001' => output := "1111";
      when '011010' => output := "1111";
      when '011011' => output := "1111";
      when '011100' => output := "1111";
      when '011101' => output := "1111";
      when '011110' => output := "1111";
      when '011111' => output := "1111";
      when '100000' => output := "1010";
      when '100001' => output := "0001";
      when '100010' => output := "0110";
      when '100011' => output := "0100";
      when '100100' => output := "1010";
      when '100101' => output := "1101";
      when '100110' => output := "1101";
      when '100111' => output := "1101";
      when '101000' => output := "0000";
      when '101001' => output := "0011";
      when '101010' => output := "0111";
      when '101011' => output := "1001";
      when '101100' => output := "0011";
      when '101101' => output := "1000";
      when '101110' => output := "1000";
      when '101111' => output := "1000";
      when '110000' => output := "1011";
      when '110001' => output := "1100";
      when '110010' => output := "0101";
      when '110011' => output := "0001";
      when '110100' => output := "0001";
      when '110101' => output := "0101";
      when '110110' => output := "0101";
      when '110111' => output := "0101";
      when '111000' => output := "1011";
      when '111001' => output := "1101";
      when '111010' => output := "1101";
      when '111011' => output := "1101";
      when '111100' => output := "1111";
      when '111101' => output := "1111";
      when '111110' => output := "1111";
      when '111111' => output := "1111";
    end case;
  end sbox;
when '110100' = output := '0010';
when '110101' = output := '1110';
when '110110' = output := '1100';
when '110111' = output := '0011';
when '111000' = output := '0101';
when '111001' = output := '1011';
when '111010' = output := '1010';
when '111011' = output := '0101';
when '111100' = output := '1110';
when '111101' = output := '0010';
when '111110' = output := '0011';
when '111111' = output := '1100';
when others = null;
end case;
end process;
end dk_sbox3_beh;
entity dk_sbox4 is
  port (input in bit_vector(1 to 6);
        output out bit_vector(1 to 4));
begin
end dk_sbox4;

architecture dk_sbox4_beh of dk_sbox4 is
begin
  sbox process (input)
  begin
    -- sbox4
    case input is
      when "000000" => output = "0111";
      when "000001" => output = "1101";
      when "000010" => output = "1101";
      when "000011" => output = "1000";
      when "000100" => output = "1110";
      when "000101" => output = "1011";
      when "000110" => output = "0011";
      when "000111" => output = "0101";
      when "001000" => output = "0000";
      when "001001" => output = "0110";
      when "001010" => output = "0110";
      when "001011" => output = "1111";
      when "001100" => output = "1001";
      when "001101" => output = "0000";
      when "001110" => output = "1010";
      when "001111" => output = "0011";
      when "010000" => output = "0001";
      when "010001" => output = "0100";
      when "010010" => output = "0100";
      when "010011" => output = "1111"
      when "010100" => output = "1000";
      when "010101" => output = "0010";
      when "010110" => output = "0110";
      when "010111" => output = "1111";
      when "011000" => output = "0110";
      when "011001" => output = "1101";
      when "011010" => output = "1101";
      when "011011" => output = "0110";
      when "011100" => output = "1110";
      when "011101" => output = "0111";
      when "011110" => output = "1111";
      when "011111" => output = "1111";
      when "100000" => output = "1010";
      when "100001" => output = "0011";
      when "100010" => output = "0110";
      when "100011" => output = "1111";
      when "100100" => output = "1101";
      when "100101" => output = "0000";
      when "100110" => output = "0000";
      when "100111" => output = "0110";
      when "101000" => output = "1100";
      when "101001" => output = "0001";
      when "101010" => output = "1100";
      when "101011" => output = "0001";
      when "101100" => output = "1110";
      when "101101" => output = "0111";
      when "101110" => output = "1111";
      when "101111" => output = "0111";
      when "110000" => output = "1101";
      when "110001" => output = "1101";
      when "110010" => output = "1101";
      when "110011" => output = "1101";
      when "110100" => output = "0001";
      when "110101" => output = "0001";
      when "110110" => output = "0001";
      when "110111" => output = "0001";
      when "111000" => output = "0100";
      when "111001" => output = "0100";
      when "111010" => output = "0100";
      when "111011" => output = "0100";
      when "111100" => output = "0100";
      when "111101" => output = "0100";
      when "111110" => output = "0100";
      when "111111" => output = "0100";
    end case;
  end process;
end dk_sbox4_beh;
when "110100" = output := "0011";
when "110101" = output := "0101";
when "110110" = output := "1110";
when "110111" = output := "1011";
when "111000" = output := "0101";
when "111001" = output := "1100";
when "111010" = output := "0010";
when "111011" = output := "0111";
when "111100" = output := "1000";
when "111101" = output := "0010";
when "111110" = output := "0100";
when "111111" = output := "1110";
when others := null;
end case;
end process;
end dk_sbox4_beh;
entity dk_sbox5 is
    port (input: in bit_vector(1 to 6);
          output: out bit_vector(1 to 4));
begin
end dk_sbox5;

architecture dk_sbox5_beh of dk_sbox5 is
begin
    sbox process (input)
    begin
        case input is
            when "000000" => output := "0010010111";
            when "000001" => output := "0101001000";
            when "000010" => output := "0111011100";
            when "000011" => output := "1001000001";
            when "000100" => output := "1010011011";
            when "000101" => output := "0001011101";
            when "000110" => output := "0001110100";
            when "000111" => output := "1000001000";
            when "001000" => output := "1010011001";
            when "001001" => output := "0100100001";
            when "001010" => output := "0111101000";
            when "001011" => output := "0000100011";
            when "001100" => output := "1001110010";
            when "001101" => output := "0110100000";
            when "001110" => output := "0111010001";
            when "001111" => output := "0000010100";
            when "010000" => output := "0111010000";
            when "010001" => output := "1001001101";
            when "010010" => output := "1001011100";
            when "010011" => output := "1001111001";
            when "010100" => output := "1100100010";
            when "010101" => output := "1100111000";
            when "010110" => output := "1100100100";
            when "010111" => output := "1100101001";
            when "011000" => output := "1100110001";
            when "011001" => output := "1100111000";
            when "011010" => output := "1100100100";
            when "011011" => output := "1100101001";
            when "011100" => output := "1100110000";
            when "011101" => output := "1100111000";
            when "011110" => output := "1100100100";
            when "011111" => output := "1100101001";
            when "100000" => output := "1100110000";
            when "100001" => output := "1100111000";
            when "100010" => output := "1100100100";
            when "100011" => output := "1100101001";
            when "100100" => output := "1100110000";
            when "100101" => output := "1100111000";
            when "100110" => output := "1100100100";
            when "100111" => output := "1100101001";
            when "101000" => output := "1100110000";
            when "101001" => output := "1100111000";
            when "101010" => output := "1100100100";
            when "101011" => output := "1100101001";
            when "101100" => output := "1100110000";
            when "101101" => output := "1100111000";
            when "101110" => output := "1100100100";
            when "101111" => output := "1100101001";
            when "110000" => output := "1100110000";
            when "110001" => output := "1100111000";
            when "110010" => output := "1100100100";
            when "110011" => output := "1100101001";
            when "110100" => output := "1100110000";
            when "110101" => output := "1100111000";
            when "110110" => output := "1100100100";
            when "110111" => output := "1100101001";
            when "111000" => output := "1100110000";
            when "111001" => output := "1100111000";
            when "111010" => output := "1100100100";
            when "111011" => output := "1100101001";
            when "111100" => output := "1100110000";
            when "111101" => output := "1100111000";
            when "111110" => output := "1100100100";
            when "111111" => output := "1100101001";
        end case
    end sbox;
when '110100' = output := '1100';
when '110101' = output := '0000';
when '110110' = output := '0110';
when '110111' = output := '1010';
when '111000' = output := '0100';
when '111001' = output := '1001';
when '111010' = output := '0101';
when '111011' = output := '0011';
when '111100' = output := '0011';
when '111101' = output := '0101';
when '111110' = output := '0011';
when others = null;
end case;
end process;
end dk_oBox5_beh;
entity dk_sbox6 is
    port (input: bit_vector(1 to 6);
        output: out bit_vector(1 to 4));
begin
end dk_sbox6;

architecture dk_sbox6_beh of dk_sbox6 is
begin
    sbox process (input)
    begin
        case input is
            when "000000" => output := "1100";
            when "000001" => output := "1010";
            when "000010" => output := "0001";
            when "000011" => output := "1111";
            when "000100" => output := "1010";
            when "000101" => output := "0100";
            when "000110" => output := "1111";
            when "000111" => output := "0010";
            when "001000" => output := "1001";
            when "001001" => output := "0111";
            when "001010" => output := "0010";
            when "001011" => output := "1100";
            when "001100" => output := "0110";
            when "001101" => output := "1001";
            when "001110" => output := "0101";
            when "001111" => output := "1011";
            when "010000" => output := "0000";
            when "010001" => output := "0101";
            when "010010" => output := "1011";
            when "010011" => output := "1110";
            when "010100" => output := "1101";
            when "010101" => output := "0011";
            when "010110" => output := "1011";
            when "010111" => output := "0111";
            when "011000" => output := "0110";
            when "011001" => output := "1010";
            when "011010" => output := "0111";
            when "011011" => output := "1110";
            when "011100" => output := "0100";
            when "011101" => output := "1100";
            when "011110" => output := "0000";
            when "011111" => output := "1000";
            when "100000" => output := "1000";
            when "100001" => output := "0001";
            when "100010" => output := "1110";
            when "100011" => output := "0111";
            when "100100" => output := "1111";
            when "100101" => output := "0010";
            when "100110" => output := "1001";
            when "100111" => output := "0100";
            when "101000" => output := "1101";
            when "101001" => output := "0101";
            when "101010" => output := "1010";
            when "101011" => output := "0110";
            when "101100" => output := "1011";
            when "101101" => output := "0111";
            when "101110" => output := "1111";
            when "101111" => output := "0000";
            when "110000" => output := "0000";
            when "110001" => output := "1010";
            when "110010" => output := "0011";
            when "110011" => output := "1110";
            when "110100" => output := "1000";
            when "110101" => output := "0100";
            when "110110" => output := "1100";
            when "110111" => output := "0101";
            when "111000" => output := "0110";
            when "111001" => output := "1101";
            when "111010" => output := "0111";
            when "111011" => output := "1110";
        end case;
    end process;
end architecture dk_sbox6_beh;
when '110100' = output := '0100';
when '110101' = output := '0001';
when '110110' = output := '1010';
when '110111' = output := '0111';
when '111000' = output := '0001';
when '111001' = output := '0110';
when '111010' = output := '1001';
when '111011' = output := '1101';
when '111100' = output := '0000';
when '111101' = output := '0111';
when '111110' = output := '1000';
when '111111' = output := '1101';
when others = null;
end case;
end process;
end dk_sbox0_beh;
entity dk_sbox7 is
  port (input in bit_vector(1 to 6);
        output out bit_vector(1 to 4));
begin
  end dk_sbox7;

architecture dk_sbox7_beh of dk_sbox7 is
begin
  sbox process (input)
  begin
    case input is
    when "000000" => output := "0100";
    when "000001" => output := "1111";
    when "000010" => output := "1011";
    when "000011" => output := "0000";
    when "000100" => output := "0010";
    when "000101" => output := "1011";
    when "000110" => output := "1110";
    when "000111" => output := "0111";
    when "001000" => output := "1111";
    when "001001" => output := "0001";
    when "001010" => output := "1001";
    when "001011" => output := "0101";
    when "001100" => output := "1000";
    when "001101" => output := "0010";
    when "001110" => output := "0101";
    when "001111" => output := "1101";
    when "010000" => output := "0011";
    when "010001" => output := "1110";
    when "010010" => output := "0100";
    when "010011" => output := "1010";
    when "010100" => output := "0001";
    when "010101" => output := "1001";
    when "010110" => output := "0111";
    when "010111" => output := "1111";
    when "011000" => output := "1001";
    when "011001" => output := "0010";
    when "011010" => output := "1100";
    when "011011" => output := "0110";
    when "011100" => output := "1100";
    when "011101" => output := "0111";
    when "011110" => output := "0111";
    when "011111" => output := "1010";
    when "100000" => output := "0001";
    when "100001" => output := "0100";
    when "100010" => output := "1000";
    when "100011" => output := "1100";
    when "100100" => output := "1100";
    when "100101" => output := "0001";
    when "100110" => output := "0100";
    when "100111" => output := "1000";
    when "101000" => output := "1010";
    when "101001" => output := "1110";
    when "101010" => output := "0001";
    when "101011" => output := "1010";
    when "101100" => output := "0001";
    when "101101" => output := "1010";
    when "101110" => output := "1010";
    when "101111" => output := "1111";
    end case;
  end sbox;
end dk_sbox7_beh;
when '110100' = output := '0110';
when '110101' = output := '0000';
when '110110' = output := '1000';
when '110111' = output := '1111';
when '111000' = output := '0000';
when '111001' = output := '1110';
when '111010' = output := '0101';
when '111011' = output := '0010';
when '111100' = output := '1001';
when '111101' = output := '0011';
when '111110' = output := '0010';
when '111111' = output := '1100';
when others = null;
end case;
end process;
end dk_sbox7_beh;
entity dk_sbox8 is
  port (input in bit_vector(1 to 6);
        output out bit_vector(1 to 4));
begin
  begin
     -- sbox
     case input is
     when "000000" => output := "1101";
     when "000001" => output := "0001";
     when "000010" => output := "1100";
     when "000011" => output := "0000";
     when "000100" => output := "1101";
     when "000101" => output := "0001";
     when "000110" => output := "0010";
     when "000111" => output := "0111";
     when "001000" => output := "0011";
     when "001001" => output := "0100";
     when "001010" => output := "1110";
     when "001011" => output := "1001";
     when "001100" => output := "1001";
     when "001101" => output := "0010";
     when "001110" => output := "1100";
     when "001111" => output := "0001";
     when "010000" => output := "1110";
     when "010001" => output := "0010";
     when "010010" => output := "1011";
     when "010011" => output := "0100";
     when "010100" => output := "0111";
     when "010101" => output := "0101";
     when "010110" => output := "1010";
     when "010111" => output := "0010";
     when "011000" => output := "0111";
     when "011001" => output := "1010";
     when "011010" => output := "1010";
     when "011011" => output := "0100";
     when "011100" => output := "0111";
     when "011101" => output := "1001";
     when "011110" => output := "0111";
     when "011111" => output := "1100";
     when "100000" => output := "0010";
     when "100001" => output := "1011";
     when "100010" => output := "0001";
     when "100011" => output := "0100";
     when "100100" => output := "0011";
     when "100101" => output := "0001";
     when "100110" => output := "1110";
     when "100111" => output := "0010";
     when "101000" => output := "0111";
     when "101001" => output := "0100";
     when "101010" => output := "1110";
     when "101011" => output := "1010";
     when "101100" => output := "1010";
     when "101101" => output := "0001";
     when "101110" => output := "0011";
     when "101111" => output := "0111";
     when "110000" => output := "1110";
     when "110001" => output := "1001";
     when "110010" => output := "1001";
     when "110011" => output := "1100";
     when "110100" => output := "1100";
     when "110101" => output := "1110";
     when "110110" => output := "1110";
     when "110111" => output := "1110";
     when "111000" => output := "1110";
     when "111001" => output := "1110";
     when "111010" => output := "1110";
     when "111011" => output := "1110";
     when "111100" => output := "1110";
     when "111101" => output := "1110";
     when "111110" => output := "1110";
     when "111111" => output := "1110";
     end_case
end dk_sbox8;
when "110100" => output := "1010";
when "110101" => output := "1001";
when "110110" => output := "1101";
when "110111" => output := "0000";
when "111000" => output := "1111";
when "111001" => output := "0011";
when "111010" => output := "0011";
when "111011" => output := "1001";
when "111100" => output := "1001";
when "111101" => output := "1011";
when "111110" => output := "1011";
when others => null;
end case;
end process;
end dk_sbox8_beh;
Title: test_keycheck
Purpose: test the VHDL model of the DES cracker
Filename: test_keycheck.vhd

Revisions
Date  Author  Version  Comments
5/20/97  T. Oelke  Rev. 1  creation

entity test_keycheck is
  generic (q_size integer := 8;
            d_size integer := 4);
end test_keycheck;

architecture test_kc_beh of test_keycheck is
  signal clk bit;
  signal reset bit;
  signal key_fnd bit;
  signal done bit;
  signal data_in bit_vector(1 to 8);
  signal write bit;
  signal addr_in bit_vector(1 to 5);
  signal key_out bit_vector(1 to 28);

component deskey_check
  port (clk in bit;
        key_fnd out bit;
        done out bit;
        data_in in bit_vector(1 to 8);
        write in bit;
        addr_in in bit_vector(1 to 5);
        key_out out bit_vector(1 to 28);
        reset in bit);
end component;

begin
  KEY_CHECK: deskey_check
    port map (clk, key_fnd, done, data_in,
             write, addr_in, key_out, reset);

clock process
begin
  clk = '0';
  wait for 5 ns;
  clk = '1';
  wait for 5 ns;
end process clock;

loader process
begin
  addr_in = "00000"; -- keyspace (1 to 8) = data;
  data_in = "00000111";
  write = '1';
  wait for 2 ns;
  write = '0';
  wait for 2 ns;

  addr_in = "00001"; -- keyspace (9 to 16) = data;
  data_in = "00010111";
  write = '1';
  wait for 2 ns;
  write = '0';
  wait for 2 ns;

  addr_in = "00010"; -- keyspace (17 to 24) = data;
  data_in = "00010111";
  write = '1';
  wait for 2 ns;
  write = '0';
  wait for 2 ns;
end process loader;

86
addr_in := '010000'; -- ctext (1 to 8) := data;
data_in := '01000111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '01001'; -- ctext (9 to 16) := data;
data_in := '01001111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '01010'; -- ctext (17 to 24) := data;
data_in := '01010111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '01011'; -- ctext (25 to 32) := data;
data_in := '01011111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '01100'; -- ctext (33 to 40) := data;
data_in := '01100111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '01101'; -- ctext (41 to 48) := data;
data_in := '01101111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '01110'; -- ctext (49 to 56) := data;
data_in := '01110111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '01111'; -- ctext (57 to 64) := data;
data_in := '01111111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '10000'; -- ptext (1 to 8) := data;
data_in := '10000111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '10001'; -- ptext (9 to 16) := data;
data_in := '10001111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '10010'; -- ptext (17 to 24) := data;
data_in := '10010111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '10011'; -- ptext (25 to 32) := data;
data_in := '10011111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '10100'; -- ptext (33 to 40) := data;
data_in := '10100111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '10101'; -- ptext (41 to 48) := data;
data_in := '10101111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '10110'; -- ptext (49 to 56) := data;
data_in := '10110111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

addr_in := '10111'; -- ptext (57 to 64) := data;
data_in := '10111111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;

reset := '1';
data_in := '11111111';
wait for 20 ns;
reset := '0';

end process loader;

tester process
begin
wait;
end process tester;

end test_kc_beh;
entity test_backend is
end test_backend;

architecture test_backend_beh of test_backend is

signal ciphertext  bit_vector(1 to 64);
signal comparetext bit_vector(1 to 64);
signal key_in bit_vector(1 to 56);
signal addr bit_vector(1 to 3);
signal reset bit;
signal key_fnd bit;
signal key_out bit_vector(1 to 8);

component dk_backend
port (ciphertext in bit_vector(1 to 64);
    comparetext in bit_vector(1 to 64);
    key_in in bit_vector(1 to 56);
    addr in bit_vector(1 to 3);
    reset in bit;
    key_fnd out bit;
    key_out out bit_vector(1 to 8));
end component;

begin
UUT: dk_backend
port map (ciphertext, comparetext, key_in,
    addr, reset, key_fnd, key_out);

tester process
begin
   -- 12345678901234567890123456789012
   ciphertext ( 1 to 32) := "101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101010101
addr := '100';
wait for 5 ns;
assert (key_out = '1111000');

addr := '101';
wait for 5 ns;
assert (key_out = '1101000');

addr := '110';
wait for 5 ns;
assert (key_out = '1011000');

reset := '0';
wait for 5 ns;
assert (key_fnd = '0');
wait;
end process tester;

end test_backend_beh;
entity test_frontend is
  generic ( q_size integer := 8;
               d_size integer := 4);
end test_frontend;

architecture test_frontend_beh of test_frontend is
signal data_in      bit_vector(1 to 8);
signal addr_in      bit_vector(1 to 5);
signal write        bit;
signal key_space    bit_vector(1 to 24);
signal ptext        bit_vector(1 to 64);
signal ctext        bit_vector(1 to 64);

component dk_frontend
  port ( data in bit_vector(1 to 8),
          addr in bit_vector(1 to 5),
          write in bit,
          key_space out bit_vector(1 to 24),
          ptext out bit_vector(1 to 64),
          ctext out bit_vector(1 to 64) );
END COMPONENT;

begin
FRONTEND: dk_frontend
  port map (data_in, addr_in, write, key_space, ptext, ctext);

  loader process
  begin
    addr_in <= "00000" ; -- key_space (1 to 8) := data;
    data_in  := "00000111";
    write := '1';
    wait for 2 ns;
    write := '0';
    wait for 2 ns;
    assert (key_space(1 to 8) = "00000111")
      report "keyspace 1 to 8 is incorrect!"
      severity warning;

    addr_in <= "00001" ; -- key_space (9 to 16) := data;
    data_in  := "00001111";
    write := '1';
    wait for 2 ns;
    write := '0';
    wait for 2 ns;
    assert (key_space(9 to 16) = "00001111")
      report "keyspace 9 to 16 is incorrect!"
      severity warning;

    addr_in <= "00010" ; -- key_space (17 to 24) := data;
    data_in  := "00010111";
    write := '1';
    wait for 2 ns;
    write := '0';
    wait for 2 ns;
    assert (key_space(17 to 24) = "00010111")
      report "keyspace 17 to 24 is incorrect!"
      severity warning;

    addr_in <= "01000" ; -- ctext (1 to 8) := data;
    data_in  := "01000111";
    write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ctext(1 to 8) = '01001111')
report 'ctext 1 to 8 is incorrect!'
  severity warning;

addr_in := '01001'; -- ctext (9 to 16) := data;
data_in := '01001111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ctext(9 to 16) = '01001111')
report 'ctext 9 to 16 is incorrect!'
  severity warning;

addr_in := '01010'; -- ctext (17 to 24) := data;
data_in := '01001111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ctext(17 to 24) = '01010111')
report 'ctext 17 to 24 is incorrect!'
  severity warning;

addr_in := '01100'; -- ctext (33 to 40) := data;
data_in := '01001111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ctext(33 to 40) = '01100111')
report 'ctext 33 to 40 is incorrect!'
  severity warning;

addr_in := '01110'; -- ctext (41 to 48) := data;
data_in := '01101111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ctext(41 to 48) = '01110111')
report 'ctext 41 to 48 is incorrect!'
  severity warning;

addr_in := '01111'; -- ctext (57 to 64) := data;
data_in := '01111111';
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ctext(57 to 64) = '01111111')
report 'ctext 57 to 64 is incorrect!'
severity warning;

addr_in := "10000"; -- ptext (1 to 8) := data;
data_in := "10001111";
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ptext(1 to 8) = "10000111")
        report ptext 1 to 8 is incorrect!
severity warning;

addr_in := "10001"; -- ptext (9 to 16) := data;
data_in := "10001111";
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ptext(9 to 16) = "10001111")
        report ptext 9 to 16 is incorrect!
severity warning;

addr_in := "10010"; -- ptext (17 to 24) := data;
data_in := "10010111";
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ptext(17 to 24) = "10010111")
        report ptext 17 to 24 is incorrect!
severity warning;

addr_in := "10011"; -- ptext (25 to 32) := data;
data_in := "10011111";
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ptext(25 to 32) = "10011111")
        report ptext 25 to 32 is incorrect!
severity warning;

addr_in := "10100"; -- ptext (33 to 40) := data;
data_in := "10100111";
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ptext(33 to 40) = "10100111")
        report ptext 33 to 40 is incorrect!
severity warning;

addr_in := "10101"; -- ptext (41 to 48) := data;
data_in := "10101111";
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ptext(41 to 48) = "10101111")
        report ptext 41 to 48 is incorrect!
severity warning;

addr_in := "10110"; -- ptext (49 to 56) := data;
data_in := "10110111";
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ptext(49 to 56) = "10110111")
        report ptext 49 to 56 is incorrect!
severity warning;
addr_in := "10111"; -- ptext (57 to 64) := data;
data_in := "10111111";
write := '1';
wait for 2 ns;
write := '0';
wait for 2 ns;
assert (ptext(57 to 64) = "10111111")
    report "ptext 57 to 64 is incorrect!"
    severity warning;
wait;
end process loader;
et test_frontend_beh;
entity test_core is
end test_core;

architecture testbench of test_core is

signal clk : bit;
signal key : bit_vector(1 to 56);
signal ptext : bit_vector(1 to 64);
signal ctext : bit_vector(1 to 64);
signal key_out : bit_vector(1 to 56);

Constant num_clocks : integer := 10;
Constant ptext_in : bit_vector(1 to 64) :=
X'F670F9F30F80B_D1' ;

type cipher_test_vect is array (1 to 10) of bit_vector (1 to 64);
type key_test_vect is array (1 to 10) of bit_vector (1 to 56);
Constant ctxt_vect : cipher_test_vect :=
X"200B5795A1B8F5C3", X"E7E44098D4E570", X"C6FD330C0298372",
X"250195B49ACB8FC6", X"7A22D281F90224A", X"89D9956D27685C3",
X"B227E7AF8ABED6B", X"508DE92189A61CA", X"A2CB8D423CCAFF5",
X"90CA8E390114E77";

Constant key_vect : key_test_vect :=
X'FFFFFFFFFFFFFFF', X'FFFFFFFF3FFFFFFFFF',
X'FFFFFFFF1FFFFFFFFF', X'FFFFFFFF0FFFFFFFFF',
X'FFFFFFFF07FFFFFFFFF', X'FFFFFFFF03FFFFFFFFF',
X'FFFFFFFF01FFFFFFFFF', X'FFFFFFFF00FFFFFFFFF',
X'FFFFFFFF007FFFFFFFFF', X'FFFFFFFF003FFFFFFFFF';

Constant clockspeed : time := 5 ns;

component dk_core
port (clk in bit;
key in bit_vector(1 to 56);
ptext in bit_vector(1 to 64);
ctext out bit_vector(1 to 64);
key_out out bit_vector(1 to 56));
end component;

begin
  uut dk_core
  port map (clk, key, ptext, ctext, key_out);

  clock process
  begin
    clk := '0';
    wait for clockspeed;
    clk := '1';
    wait for clockspeed;
  end process clock;

  tester process
  begin
    ptext := ptext_in;
  end process tester;
end
for i IN 1 to num_clocks loop
    -- do the actual setting of input stuff...
    key := key_vect(i);
    wait for 2 * clockspeed;
end loop;
wait;
end process tester;

result_verify process begin
    wait for 16 * 2 * clockspeed;
    for i IN 1 to num_clocks loop
        -- do the actual testing of stuff...
        assert ctext = ctxt_vect(i)
            report "Ciphertext is incorrect!"
            severity warning;
        assert key_out = key_vect(i)
            report "Key is incorrect coming out!"
            severity warning;
        wait for 2 * clockspeed;
    end loop;
    wait;
end process result_verify;
end testbench;
entity test_keygen is
  generic ( q_size integer := 8;
            d_size integer := 4);
end test_keygen;

architecture test_keygen_beh of test_keygen is

  signal keyspace bit_vector(1 to 24);
  signal clk bit;
  signal reset bit;
  signal done bit;
  signal key bit_vector(1 to 56);

  component dk_keygen
  port (keyspace in bit_vector(1 to 24);
       clk in bit;
       reset in bit;
       done out bit;
       key out bit_vector(1 to 56));
  end component;

begin
  UUT: dk_keygen
    port map (keyspace, clk, reset, done, key);

  clockgen process
  begin
    clk := not clk;
    wait for 5 ns;
  end process clockgen;

  starter process
  begin
    keyspace(1 to 8) := X'9A' ;
    keyspace(9 to 16) := X'BC' ;
    keyspace(17 to 24) := X'DE' ;
    wait for 1 ns;
    reset := '1';
    wait for 1 ns;
    reset := '0';
    wait;
  end process starter;
end test_keygen_beh;
Appendix C:

Synopsys Generated Schematics
Figure C-2 Front end before optimization
Figure C-3 Front end after optimization
(30 combinational area, 152 noncombinational area)
Figure C-4 Key generator before optimization
Figure C-5 Key generator after optimization
(26 combinational area, 88 noncombinational area)
Figure C-6 Encryption core before optimization
Figure C-7 Encryption core after optimization
(15424 combinational area, 1920 noncombinational area)
(worst path timing: 55.57 ns)
Figure C-8 One round of encryption after optimization
(782 combinational area, 64 noncombinational area)
(worst path timing: 44.71 ns)
Figure C-9 Back end before optimisation
Figure C-10 Back end after optimisation
(147 combinational area, 57 noncombinational area)
Appendix D:

Simulation Results for Encryption Core