FPGA realization of a public key block cipher

Stig Fjellskaalnes

Master of Science in Computer Science
Submission date: June 2009
Supervisor: Danilo Gligoroski, ITEM
Co-supervisor: Mohamed El-Hadedy, ITEM
Problem Description

Recently, a new public key algorithm have been proposed by Gligoroski, Markovski and Knapskog. The algorithm belongs to the class of public keys algorithms realized by multivariate quadratic equations. The authors found out a new class of quasigroups that have special form when expressed as Boolean functions. The quasigroups are multivariate quadratic.

One important characteristic for this new public key algorithm is that it is very fast. Realized in software it can produce digital signatures around 300 times faster than RSA (1024 bit public key length). However, in hardware the algorithm can achieve speeds equivalent to symmetric key primitives both in signature generation and in its verification. That means the algorithm realized in hardware can be 1,000 to 10,000 times faster than corresponding public key algorithms (RSA, Diffie Helman or Elliptic Curve algorithms) realized also in hardware.

The student will have a task to write a VHDL code and to realize the algorithm in FPGA, both encryption and decryption. The realization will use variable public and private keys stored in RAM, not fixed keys stored in ROM blocks.

Assignment given: 15. January 2009
Supervisor: Danilo Gligoroski, ITEM
Abstract

This report will cover the physical realization of a public key algorithm based on multivariate quadratic quasigroups. The intension is that this implementation will use real keys and data. Efforts are also taken in order to reduce area cost as much as possible. The solution will be described and analyzed. This will show whether the measures were successful or not.
## Contents

1 Introduction 9

2 Theory 11
   2.1 MQQ in general 11
   2.2 Logic optimization through minimization 11
      2.2.1 Minimization with Karnaugh maps 13
      2.2.2 ESPRESSO-II minimization algorithm 13

3 Hardware implementation of MQQ 17
   3.1 Hardware implementations in general 17
   3.2 The actual implementation 18
      3.2.1 Encryption 18
      3.2.2 Decryption 23
   3.3 Data and keys 28
   3.4 Optimizing the stored fixed values 28

4 Results 31
   4.1 Synthesis 31
   4.2 Verification of functionality through simulation 31
      4.2.1 Encryption 31
      4.2.2 Decryption 33

5 Discussion and conclusion 39
   5.1 Discussion 39
      5.1.1 Synthesis 39
      5.1.2 Design behaviour simulation and verification 40
   5.2 Conclusion 40
   5.3 Future work 40
   5.4 Contributors 40

6 Abbreviations 41

A VHDL source code 45
   A.1 Encryption 45
   A.2 Decryption 53

B Espresso minimization 67

C Synthesis report from Decryption 69
List of Figures

2.1 Karnaugh diagram with reduction .................................................. 14
3.1 Encryption top level ................................................................. 18
3.2 Encryption state diagram ............................................................ 19
3.3 Expander internal architecture ..................................................... 21
3.4 Expander state diagram ............................................................. 21
3.5 Public Matrix internal architecture .............................................. 22
3.6 Public Matrix state diagram ......................................................... 23
3.7 Decryption internal architecture ............................................... 24
3.8 Decryption top level state diagram ............................................ 24
3.9 Private Matrix internal architecture .......................................... 25
3.10 Private Matrix state diagram .................................................... 25
3.11 Dobbertin internal architecture ................................................. 27
3.12 Sequencer internal architecture ............................................... 27
3.13 Sequencer state diagram ......................................................... 28
3.14 Master Read Only Memory (ROM) internal architecture ............... 29
4.1 Encryption counter synchronization ............................................. 32
4.2 Encryption top module missing state ......................................... 32
4.3 Expander behaviour ............................................................... 33
4.4 Public matrix calculation ......................................................... 34
4.5 Public matrix bit-by-bit XOR .................................................... 34
4.6 Simulation of startup ............................................................. 35
4.7 Stored private key as a function ............................................... 35
4.8 XOR procedure ................................................................. 35
4.9 Dobbertin calculation ............................................................ 36
4.10 Sequencer startup ............................................................... 36
4.11 Master ROM calculation ......................................................... 37
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Key generation algorithm</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Encryption in Multivariate Quadratic Quasigroups (MQQ)</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>MQQ public key for 160 bit MQQ</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Decryption procedure</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Truth table</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>ESPRESSO-II minimization algorithm</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Expansion</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Espresso input file format</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Synthesis results of the MQQ decryption procedure</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Synthesis results of the MQQ encryption procedure</td>
<td>32</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This thesis will cover an attempt to realize a fairly new public key algorithm on an Field-progammable Gate Array (FPGA), the MQQ algorithm. Previous attempts of implementing this algorithm has shown that the area consumption of this design is enormous, and must be reduced in order to implement this algorithm in hardware in a practical manner. There are optimization techniques that can, in theory, be used to store the information more efficient than storing it directly, which will be investigated.

In order to compare the results of the optimization, the decryption design from the TTM4530 report written by the same author of this master thesis will be used as a reference.
Chapter 2

Theory

In this chapter the theoretical background for the techniques used in the implementation will be presented.

2.1 MQQ in general

MQQ is, compared to Rivest-Shamir-Adleman (RSA), Diffie-Hellman (DH) and Elliptic curve cryptography (ECC), a new type of public key algorithm. Calculations of encryption and decryption are done with logic operations such as AND and XOR between the actual data and the public and private keys. This is a high contrast to traditional public key algorithms, where encryption and decryption is done by using more complex mathematical operation. In RSA, encryption of a message is done the following way, \( c = m^e \pmod{n} \), where \( c \) is the encrypted message, \( m \) is the original message, \((n, e)\) is the public key. Encryption is done the same way, \( m = c^d \pmod{n} \), \( d \) being the private key exponent. The MQQ public key consists of boolean values arranged in a \( n \times n \) matrix generated randomly. The MQQ private key is derived from the output of the public key. It follows the following procedure, given in table 2.1.

The MQQ encryption is performed with an expansion of the input data. Then each term is anded with the respective term from the private key. The number of bits determines the number of equations that are to be performed. More bits mean more equations. With 160 bits input data, there will be 160 equations, and each equations have 12881 terms. The result of this operation is joined in a resultant vector, which will be the encrypted data, as shown in table 2.1. '+' is here an XOR, and multiplication. Basically, the encryption can be represented as the equation \( y = P(x) \equiv y = A \cdot X \).

For 160 bit MQQ, the public key is defined as a matrix of \( 160 \times 12881 \) elements, in the form presented in table 2.1.

Decryption is described in table 2.1. The decryption is done in seven stages total where logical operations (AND, XOR) are done with the input value, and the result is the original data in cleartext.

2.2 Logic optimization through minimization

As described, the public and private keys in MQQ are boolean values stored in matrices with considerable size. In earlier implementations of the MQQ the keys have been represented as fixed numbers. Logic optimization and minimization methods can be used to reduce the storage needs for these blocks. This report will explore if this is the case with MQQ.
Algorithm for generating Public and private keys for the MQQ scheme

Input: Integer n, where $n = 5k$ and $k \geq 28$

Output: public key $P$: $n$ multivariate quadratic polynomials $P_i(x_1, ..., x_n)$, $i = 1, ..., n$

1. Generate a nonsingular $n \times n$ boolean matrix $T$ (uniformly at random).
2. Call the procedure of definition for $P'(n) : 0,1^n \rightarrow 0,1^n$ and from there also obtain the quasigroups $*_1, ..., *_8$
3. Compute $y = T(P'(T(x)))$ where $x = x_1, ..., x_n$
4. Output: the public key is $y$ as $n$ multivariate quadratic polynomials $P_i(x_1, ..., x_n)$, $i = 1, ..., n$ and the private key is the tuple $T, *_1, ..., *_8$

Table 2.1: Key generation algorithm

<table>
<thead>
<tr>
<th>MQQ encryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0^{(1)} + (a_1^{(1)} \times x_1) + (a_2^{(1)} \times x_2) + ... + (a_{12881}^{(1)} \times x_{159} \times x_{160})$</td>
</tr>
<tr>
<td>$a_0^{(2)} + (a_1^{(2)} \times x_1) + (a_2^{(2)} \times x_2) + ... + (a_{12881}^{(2)} \times x_{159} \times x_{160})$</td>
</tr>
<tr>
<td>$a_0^{(3)} + (a_1^{(3)} \times x_1) + (a_2^{(1)} \times x_2) + ... + (a_{12881}^{(3)} \times x_{159} \times x_{160})$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>$a_0^{(160)} + (a_1^{(160)} \times x_1) + (a_2^{(160)} \times x_2) + ... + (a_{12881}^{(160)} \times x_{159} \times x_{160})$</td>
</tr>
</tbody>
</table>

Table 2.2: Encryption in MQQ

<table>
<thead>
<tr>
<th>160 bit MQQ public key</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0^{(1)} a_1^{(1)} a_2^{(1)} a_3^{(1)} a_4^{(1)} ... a_{12881}^{(1)}$</td>
</tr>
<tr>
<td>$a_0^{(2)} a_1^{(2)} a_2^{(2)} a_3^{(2)} a_4^{(2)} ... a_{12881}^{(2)}$</td>
</tr>
<tr>
<td>$a_0^{(3)} a_1^{(3)} a_2^{(3)} a_3^{(3)} a_4^{(3)} ... a_{12881}^{(3)}$</td>
</tr>
<tr>
<td>$a_0^{(4)} a_1^{(4)} a_2^{(4)} a_3^{(4)} a_4^{(4)} ... a_{12881}^{(4)}$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>$a_0^{(160)} a_1^{(160)} a_2^{(160)} a_3^{(160)} a_4^{(160)} ... a_{12881}^{(160)}$</td>
</tr>
</tbody>
</table>

Table 2.3: MQQ public key for 160 bit MQQ
Algorithm for decryption/signing with the private key \((T, *_1, \ldots, *_8)\)

<table>
<thead>
<tr>
<th>Input: A vector (y = (y_1, \ldots, y_n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: A vector (x = (x_1, \ldots, x_n)) such that (P(x) = y)</td>
</tr>
</tbody>
</table>

1. Set \(y' = T^{-1}(y)\)
2. Set \(W = (y'_1, y'_2, y'_3, y'_4, y'_5, y'_6, y'_7, y'_8) = \text{Dob}^{-1}(W)\)
3. Compute \(Z = (Z_1, Z_2, Z_3, Z_4, Z_5, Z_6, Z_7, Z_8)\)
4. Set \(y_1 \leftarrow Z_1, y_2 \leftarrow Z_2, y_3 \leftarrow Z_3, \ldots, y_{16} \leftarrow Z_{16}, y_{21} \leftarrow Z_{21}, y_{26} \leftarrow Z_{26}, y_{31} \leftarrow Z_{31}, y_{36} \leftarrow Z_{36}, y_{41} \leftarrow Z_{41}\)
5. Represent \(y'\) as \(y' = Y_1 \ldots Y_k\) where \(Y_i\) are vectors of dimension 5
6. By using the left parastrophes \(_i\) of the quasigroups \(*_i\), \(i = 1, \ldots, 8\), obtain \(x' = X_1 \ldots X_k\), such that: \(X_1 = Y_1, X_2 = X_1 \setminus Y_2, X_3 = X_2 \setminus Y_3\) and \(X_i = X_{i-1} \setminus (i+2) \mod 6 \setminus Y_i\)
7. Compute \(x = S^{-1}(x')\)

Table 2.4: Decryption procedure

### 2.2.1 Minimization with Karnaugh maps

Karnaugh maps are widely used when it comes to minimizing and optimizing logical expressions, and to ease the use of boolean algebra. Given the following truth table 2.2.1, this table will be translated to a karnaugh map [5]. The expression can further be reduced, and the result \(Y\) from the truth table can be expressed as the boolean function \(Y = aB + bD + Cd\) (big capitals = negation).

### 2.2.2 ESPRESSO-II minimization algorithm

Since the key size of MQQ for 160 bit is big, there is another, more efficient method for minimizing large matrices of boolean values, by using an algorithm called ESPRESSO. The ESPRESSO-II minimization algorithm has been implemented as a lightweight program. The algorithm is listed in table 2.2.2.

Here follows a brief presentation of the different procedures in ESPRESSO-II minimization algorithm [4]. The algorithm starts with an UNWRAP, which is a preprocessor that has to discover any incoming cube sharing whatever may be present in the incoming data. COMPLEMENT computes R or D if F and R are given as the input. EXPAND replaces the cubes of F by prime implicants and makes sure that coverage is minimal as to single-cube containment. The consequence will be that EXPAND reduces the number of cubes in F. The routine ESSENTIAL_PRIMES locates the essential primes which must be present in every cover of F. When detected, they are added to the don’t-care set D, which prevents the primes from appearing more than one time. This routine is only executed during the first iteration of LOOP1. IRREDUDANT_COVER sorts the covers of F into totally redundant, relatively essential and partially redundant. All cubes that are totally redundant are discarded, and a minimal subset
Figure 2.1: Karnaugh diagram with reduction

Table 2.5: Truth table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Begin
F ← UNWRAP(F)
R ← COMPLEMENT(F,D)
φ1* ← φ2* ← φ3* ← φ4* ← COST(F)

LOOP1: (φ, F) ← EXPAND(F,R)
  if (first - pass)
    (φ, F, D, E) ← ESSENTIAL_PRIMES(F,D)
    if (φ ≡ φ1*) goto OUT
    φ1* ← φ
    (φ, F) ← IRREDUANT_COVER(F,D)
    if φ ≡ φ2* goto OUT
    φ2* ← φ

LOOP2: (φ, F) ← REDUCE(F,D)
  if φ ≡ φ3* goto OUT
  φ3* ← φ
  goto LOOP1

OUT: if (φ ≡ φ4*) goto QUIT
 (φ', F) ← LAST_GASP(F,D,R)
  if (φ ≡ φ') goto QUIT
  φ1* ← φ2* ← φ3* ← φ4* ← φ'
  goto LOOP2

QUIT: F ← F ∪ E
      D ← D − E
      (φ, F) ← MAKE_SPARSE(F,D,R)
return (φ, F)
End

Table 2.6: ESPRESSO-II minimization algorithm
of the two other types plus D will be sufficient to cover all minterms for F. REDUCE improves the result over the local minimums that IRREDUDANT_COVER obtains. This is done by taking each cube $c \in F$ and then reducing it to the smallest cube $\widetilde{c}$. LAST_GASP is reminiscent of REDUCE, but uses an order independent reduction process. The ESPRESSO-II algorithm finishes with MAKE_SPARSE. Essential primes are first taken out of the don’t-care set D and put back in the cover F. Then the procedure considers the number of cubes in the cover as final. It also attempts to reduce the number of literals by “lowering” the outputs and “raising” the inputs. MAKE_SPARSE also attempts to make the final cover minimal, in a way that no input literal, output literal or product term can be removed while retaining coverage of $ff$. 
Chapter 3

Hardware implementation of MQQ

Since the assignment is to realize the MQQ algorithm in hardware, typically an FPGA implementation has been made. This chapter will describe, in detail, a hardware implementation of MQQ written in the hardware description language Very-High-Speed Integrated Circuits Hardware Description Language (VHDL). Both the encryption and decryption has been implemented. As mentioned earlier, since the implementation is intended for realization, optimization and minimization techniques has been tried in order to reduce area cost on the FPGA. Xilinx ISE 10.1 (with all updates and service packs installed) has been used as the Integrated Development Environment (IDE) for this design. ISE also contains the synthesis tool Xilinx Synthesis Tool (XST), which is necessary in order to prepare the design for upload to FPGA. There exists an earlier hardware implementation of MQQ written by Mohamed El-Hadedy. That implementation was developed with emphasis on speed only, no area reduction efforts were made. In this design, a real private key has been used. But, the public key is a randomly generated key with no relation to the private key, because the real public key was not available. This report has the shortened version of the VHDL source code, the complete source code files are located in the digital attachment.

3.1 Hardware implementations in general

In order to fully demonstrate the speed of an algorithm, a hardware implementation is needed. In a software environment a program runs on a microprocessor. There, the speed is dependent on the Central Processing Unit (CPU) utilization. This is not the case with hardware implementations. Since a fixed, physical area is being assigned to the design, the run-time of the algorithm is about the same every time the algorithm is running with little or no deviation in run-time. Hardware implementations are useful because the encryption/decryption, and in many cases, key generation, will use dedicated hardware in its operation. This will increase speed, decrease delay and use of resources. In systems where hardware implementations of a cryptographic algorithm is used, the hardware implementations is referred to as a hardware accelerator. Hardware accelerators are especially useful in embedded systems when processing resources are limited. The procedure of encryption/decryption does not change, only the keys and data to be used. Therefore, in an embedded system it will save time, and probably energy to implement the algorithm in hardware.
3.2 The actual implementation

As mentioned earlier, the implementation is written in **VHDL**. All components, both in encryption and decryption have been implemented as a Mealy type Finite State Machine (FSM), with clocked (synchronous) output. The reset signal is active high (has value '1' when active), and synchronous. This is done because an FSM is convenient in hardware realization, which will help to keep the data flow in order. The reason for keeping most of the design clocked is because when a process is clocked, the registers remember the values to the next clock cycle. In this setting it is important, if the output had been made combinatorial (asynchronous), it had been necessary to set the value of each register in every state, and the probability for latches would have been present (a latch is a register with a value that does not change at any time. “Enable” signals are also used, both en_in and en_out. The en_in signal for each module determines whether the module is active or not. It is implemented because it makes it possible to turn off other modules than those that are active. For instance, if the sequencer module is processing the data, the Dobbertin component is not needed, and en_in for Dobbertin can be set to '0'. This means that the Dobbertin module is inactive, which will prevent Dobbertin to send data at the wrong time. And it may also save power. If the Dobbertin module would not have this feature, the module would have been active constantly while the circuit had been working. Enable out (en_out) signals for the different sub components are used as hand-shake signal. When a module signalizes that its data is ready for the next module, that module’s en_out is set to '1'.

3.2.1 Encryption

As explained in the theory chapter, the encryption of an input data is calculated as given in table 2.1. Since there are many equations with many terms, the calculation must be split up in more than one stage.

Encryption of MQQ is implemented as the figure 3.1 shows. There are two components, Expander and Public Matrix. Expander expands the input vector from 160 bits to 12881 bits as table 3.2.1 shows, the Public Matrix does the calculation \( y = P(x) \equiv y = A \cdot X \).
**Encryption top module**

The encryption component is the top level module that controls the data flow, and the state diagram is listed in figure 3.2. The initial state is IDLE, which starts the encryption process by activating the EXPAND module. The top module will stay in EXPAND state until the en_out flag from expander gets the value '1', then the state machine moves to PUB_MATR. As with state EXPAND, the top module will stay in PUB_MATR until the public matrix en_out flag is set to 1, then the state machine moves back to EXPAND. A common criteria for both PUB_MATR and EXPAND is that if the ready flag, controlled by public matrix is set to '1', the state machine moves to state RES, which indicates that the encryption is completed. In RES, the state machine moves back to IDLE for new data to be encrypted.

**Expander**

This component does the \( AND \) expansion of the 160 bit input data into 12881 bits. Since 12881 bits will be too much to send to the public matrix at once, the 12881 bits of data is multiplexed into 80 vectors of length 160 bit, and the last 81 bits is sent as a single vector. This demands two output vectors of 160 and 81 bits respectively, as the figure 3.3 shows. There are three states in expander, IDLE, COMB and SEL, as shown in figure 3.4. IDLE is the first state, where the input vector is imported into the module. The signal en_out is set to '0', indicating that data is not ready to be sent to Public Matrix. After this has been done, expander moves to the COMB state. Here the expansion takes place, from 160 bit to 12881 bits. This is done by doing \( AND \) operations between the input and a tmp register, which holds the same value as input. The result is stored in the 12881 bit vector. In theory this should be done within one clock cycle. When this is done, the module reaches the SEL stage. Here the output is calculated. An iterator, inc, is used to push 160 bit of data from the 12881 bit vector to be sent to Public Matrix by writing to output._1. Which 160 bits from the 12881 bit vector to be sent
is determined by the counter, and commented in the source code located in the appendix part of the report \(A.2\). At the same time the last 81 bit of the 12881 bit vector is also written to the output register output\(_2\). The flag en\(_{\text{out}}\) is now set to ‘1’, data is ready to be transmitted. The state machine now goes back to IDLE, and this process is repeated, until all data has been sent. When this is complete, the iterator stops.

### Public Matrix

The Public matrix is the component where the encryption calculation is taking place. It is implemented in the way demonstrated in figure 3.5. The state machine is presented in figure 3.6.

Public Matrix has eight states. Idle is the initial state where temporary registers used in the module is set to ‘0’. Signals such as en\(_{\text{out}}\), the iterator cnt\(_2\) and done (the ready bit explained in the top level) is also set to 0 here. IDLE initiates the calculation. MATR\(_\text{AND}\) is the state where the data from Expander is AND\(_{\text{ed}}\) with the public key. Public Matrix moves to MATR\(_\text{XOR160}\) where the result from MATR\(_\text{AND}\) is XOR\(_{\text{ed}}\) with the previous vector, or stored for next iteration with AND\(_{\text{ed}}\). This loop between MATR\(_\text{AND}\), MATR\(_\text{XOR160}\) and SYNC\(_\text{160}\) (the synchronization state for the XOR\(_{\text{ed}}\) vector) will run 80 times since there are 80 vectors of length 160 to be AND\(_{\text{ed}}\) and XOR\(_{\text{ed}}\). The last 81 bit is AND\(_{\text{ed}}\) with the respective vectors from the public key one time and are awaiting the bit-by-bit XOR.

When the MATR\(_\text{AND}\), MATR\(_\text{XOR160}\) and SYNC\(_\text{160}\) loop is finished (iterator cnt will have value 79) the next stage in Public Matrix stars, the bit-by-bit XOR, where the result vectors (160 bit and 81 bit) will be XOR\(_{\text{ed}}\) down to one bit only. This is done in states MATR\(_\text{XOR1}\) with the SYNC\(_\text{1}\) as the synchronization state. cnt\(_2\) is the iterator which keeps track of how many times the iteration has been done. When completed (cnt\(_2\) gets value 159), the top module goes to SYNC state, where a last XOR is performed, between the 160 bit vectors and the 81 bit vector. The result is written to a resultant vector, which will be the encrypted data. In state SEND, the ready bit is set to ‘1’ to indicate that the data is now ready, and the answer is sent on the output port.

\[
X = \begin{bmatrix}
1 \\
x_1 \\
. \\
. \\
x_n \\
x_1 x_2 \\
x_2 x_3 \\
. \\
. \\
x_{158} x_{160} \\
x_1 x_2 \\
x_2 x_3 \\
. \\
. \\
x_{158} x_{160} \\
x_1 x_2 \\
x_2 x_3 \\
. \\
. \\
x_{158} x_{160}
\end{bmatrix}
\]

| Table 3.1: Expansion |

---

20
Figure 3.3: Expander internal architecture

Figure 3.4: Expander state diagram
Figure 3.5: Public Matrix internal architecture
3.2.2 Decryption

Decryption is implemented in four modules, Private Matrix T, Dobbertin ROM, Sequencer and Private Matrix S, shown in figure 3.7. The component Decryption is the top module that instantiates the four sub-components. It is also in the decryption procedure the logic optimization and minimization are being used. By that way it is possible to observe whether the area cost for Decryption can be reduced compared to storing the public key as fixed values, using a program that is an implementation of the ESPRESSO-II minimization algorithm presented in the theory chapter. To determine the optimization effect, the design in this assignment will be compared with the design from [8], made by the same author as this report.

The decryption top module has ten states, hence figure 3.8. As with encryption top module, the state machine in decryption controls the sub components within decryption.

Private Matrix

There are two instances of Private Matrix, the T and S matrix. They correspond to the first and the seventh step in the decryption algorithm of MQQ. Table 3.9 shows the architecture. Private Matrix T and Private Matrix S are identical, but contains different parts of the private key. One important notice is that the private key now is stored as a function of the global iterator cnt, rather than fixed values as done before in [8]. Another modification that has been made is that in the [8] implementation of Public Matrix, there were many 1 bit registers such as tmp.xxx, xor.xxx, sync_xor.xxx (where xxx is a number between 1 and 160). These registers have been replaced with 160 bit registers such as tmp, matr_xor and sync_xor.

Private Matrix has five states, IDLE, ANDOP, XORING, SYNC and PUSH, which figure 3.10 shows.

In IDLE, values from the ROMs are written to corresponding signals and en_out is set to '0' (low). When this is done and the control logic has verified the writing to the signal from rom, state ANDOP is initiated.
Figure 3.7: Decryption internal architecture

Figure 3.8: Decryption top level state diagram
Figure 3.9: Private Matrix internal architecture

Figure 3.10: Private Matrix state diagram
In ANDOP, the logical AND operation is done between the input vector and the corresponding stored vector from the ROM. To illustrate this better, when the counter (cnt) has value 10, the Private Matrix runs at 11th time. In the previous implementation in [8], the signal from rom Xxx (xxx is a number between 1 and 160) will have stored the 11th vector from the ROM array. When this is done, ANDOP state is finished, and next state will be XORING. In the current implementation, the signals from rom xxx have been removed completely, the ROM blocks that existed in [8] have now been replaced by signals which hold a function of the global counter.

The bit-by-bit XOR operation takes place in the state XORING. This state uses an internal counter, count xor to keep track of how may times the XOR operation is done. In [8], a temporary signal, tmp xxx was used to store the temporary value corresponding to the position of the vector androm in xxx. The 160 tmp xxx signals have been replaced by a 160 bit vector tmp. An XOR is then done between the tmp(position between 0 and 159) and androm in xxx at position counter+1. When the counter reaches 3, the 5 bit result from anding input and the stored ROM vector is bit-by-bit XORed into a single bit, and the state machine shifts state to SYNC.

A modification from the original design is that a new step is being introduced, a SYNC state. This is to keep synchronization, and to complete the XOR step. Here is a description why this step is needed. When the Private Matrix runs for the first time, the sync xor xxx get the value from the xor xxx, the result after the bit-by-bit XOR operation. Again, the sync xor xxx and xor xxx signals have been replaced by 160 bit vectors sync xor and matr xor, respectively. For the second run and so on, a new XOR operation is initiated between the matr xor and the sync xor, the latter signal has the value of the previous XOR operation. This is necessary to make sure that the bit-by-bit XOR operation runs as many times as it is supposed to. And when this operation is done, en out is set to ‘1’ (high). That means that the value can be written to register x, a synchronization register for the XORed bits.

When the counter reaches value 31, it means that the register x contains the result, and the Private Matrix is ready to send the data to Dobbertin ROM component. This is done in state PUSH_OUT.

**Dobbertin ROM**

The Dobbertin component [3.11] is an implementation of step 2, 3 and 4 in the decryption algorithm [2.1]. Also here, the fixed values stored in a ROM structure have been converted to functions of the 13 bit input vector by using the Espresso application.

**Sequencer**

The sequencer [3.12], that corresponds to the fifth and sixth step of the decryption procedure [2.1] is unchanged from [8].

The component sequencer has seven states. Those are IDLE, MUX2_SEL, SYNC, MUX31_SEL, SEND_TO_MASTER, RECV MR and PUSH (ref. figure 3.13).

In the INIT state, the 160bit input signal from the Dobbertin ROM is split up in an array consisting of 32 vector with length of 5 bits. When this is done, current state changes to MUX2_OUT. Here the first element of the array is being sent through the multiplexer since the selector is set to ‘0’, and becomes the first element in a similar array which will be the result that the sequencer module generates. The first element is written in the output array in state SYNC. The counters are increased by 1, and the state machine shifts to state MUX31_out. Selector of MUX 2 is set to ‘1’ because the values that are supposed to go through that MUX will not be the first element. Then the output from MUX 31 will be the 5 least significant bits.
Figure 3.11: Dobbertin internal architecture

Figure 3.12: Sequencer internal architecture
in a 10 bit vector, the most significant bits are the 5 bit vector from MUX_2. The 10 bit vector is an address, which is the input to the Master_ROM. After Master_ROM has done its job, the result will go through MUX_2 and written on the output register, and also be used as feedback for the new address to be sent into Master_ROM. This procedure will continue until counter_2 reaches value 31, which means that all 160 bits are processed by the sequencer and are ready to be written to the output register, and the value is used by Private_Matrix_S as input.

**Master ROM**

The Master ROM is a subcomponent to the sequencer. Also here, the fixed values stored in ROM blocks in the original implementation have been replaced by functions of the 10 bit input vector. There exists a control ROM which has $25 \times 3$ bit vectors and its function is to be a selector that determines which of the 8 functions to be used. Also, the control ROM has been converted to functions of the counter. The control ROM is being controlled by a counter. In this implementation, this counter is located in the sequencer(counter_2), and the value from the counter is one of the input ports. This is shown in figure 3.14.

### 3.3 Data and keys

Though the assignment indicates that real data and keys are to be used, this is not the case. The key generation calculation is complex, so it is uncertain whether the key generation process may practically be implemented in hardware. According to the main supervisor, the keys for the original implementation were generated in Wolfram Mathematica, and the key generation calculation took considerably long time to complete on a modern computer.

### 3.4 Optimizing the stored fixed values

As mentioned several places in this report, efforts have been made to optimize the stored fixed values, in order to reduce area consumption of the implementation. A program that implements
Figure 3.14: Master [ROM] internal architecture

Table 3.2: Espresso input file format

| .i 3 | ← indicates three input bits |
| .o 2 | ← indicates two output bits |
| .i A B C | ← names the variables in input |
| .o Y Z | ← names the variables in output |
| .p 8 | ← number of terms |
| 000 | ← input value |
| 010 | 01 |
| 011 | 10 |
| 100 | 00 |
| 101 | 01 |
| 110 | (- means don’t care) |
| 111 | -1 |
| .e | ← indicator for end of file |

To minimize a number of values the following must be done. First, one has to know how many input values to be minimized. For instance, if there are 1024 values to be minimized, each of length 5 bit, then each value must be represented by a unique value between 0 and 1023, 10 bit length and in binary form. There will be ten input functions and five output functions. The input file that the espresso application demands must be arranged in the proper format. An authentic file used in this project is located in the appendix of this report (appendix B). After the program has finished the minimization, an output file with the reduced expression is generated. This file has to be modified into valid **VHDL** syntax. This can easily be done with an advanced text editor, such as TextPad.

Here follows an explanation on how to arrange the data, in table 3.4.
Chapter 4

Results

In this chapter the results of the implementation will be presented. There have been attempts to synthesize both encryption and decryption, which is necessary in order to build a physical realization of this MQQ implementation in VHDL. Simulation of all the modules have also been conducted.

4.1 Synthesis

The encryption and decryption procedures have been synthesized against the FPGA meant for the physical realization, the Xilinx Virtex 5 model xc5vlx110t-1-f1136 (speed grade -1). Table 4.1 shows a brief summary of the synthesis report for decryption. Synthesis of the decryption with fixed values (from [8]) will also be presented, to show if the optimization through minimization has been successful or not. Since the synthesis of the encryption procedure failed, the synthesis results for encryption from [3] will be used due to a lack of synthesis results from this implementation, located in table 4.1. This incident will be analyzed and discussed in the Discussion chapter.

It is important to point out that the earlier encryption implementation from [3] was implemented on four FPGAs, so the content in table 4.1 is for one chip out of four.

4.2 Verification of functionality through simulation

Simulation is an important tool to verify that the design acts properly according to the specification made in the source code. The simulation tool used in this assignment is the ModelSim SE 6.3f by Modeltech.

4.2.1 Encryption

The simulation results from Encryption will be presented in this section.

<table>
<thead>
<tr>
<th>ESPRESSO-II minimization</th>
<th>Slice Registers</th>
<th>LUTs</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>5,937</td>
<td>9,950</td>
<td>201.045 MHz</td>
</tr>
<tr>
<td>No</td>
<td>5,910</td>
<td>7,703</td>
<td>214.000 MHz</td>
</tr>
</tbody>
</table>

Table 4.1: Synthesis results of the MQQ decryption procedure
### Encryption top module

Figure 4.1 shows that the encryption top module makes sure that the global counter cnt and the local counter inside Expander increases with an equal number of clock cycles.

![Figure 4.1: Encryption counter synchronization](image1)

When the public matrix module has completed the encryption, the ready bit is set to high, which should make the top module change state to RES (the state where the result from encryption is ready). But figure 4.2 shows that the RES state is never reached, even when the ready bit gets value '1', as pointed out in the figure. The cause for this problem is unclear, debugging has been conducted in order to locate the problem. However, in the process sync run, the controlling process of the Encrypton top module, dataout is to be equal to the result vector final_result, which is the answer from public matrix. By this way, dataout, the output port from Encryption which ultimately holds the answer is set to contain the encrypted data at the right time, which means that by this way, the RES state is not needed, and can be removed from the source code.

![Figure 4.2: Encryption top module missing state](image2)
Expander

The simulation verifies (figure 4.3) that the 160 bit input is expanded into 12881 bits in state COMB, marked in yellow. Also, the 160 bit output from the expanded vector is set in state SEL (red mark). The value of the 160 output changes in the next iteration (green mark), as the counter increases by 160. This is the expected behaviour of the expander module.

Figure 4.3: Expander behaviour

Public Matrix

Figure 4.4 shows that when indata_1 gets a new vector from Expander in state IDLE, the AND operation takes place in the next state, MATR_AND. One clock cycle later, the 160 bit vector XOR operations are performed. When the global counter increases value by 1, the next vector is written from Expander into indata_1, and the same operation cycle repeats until the global counter reaches 79.

When the last 160 bit vector has been XORed with the previous 79 vectors, the Public matrix enters the bit-by-bit XOR procedure, in order to finish off the encryption calculation. Figure 4.5 shows that this process starts when the requirements for entering this phase are fulfilled.

4.2.2 Decryption

Here, the verification for Decryption through simulation is being presented.

Decryption top module

First, it is necessary to verify that all signals initially are set to '0' when reset is active. As the figure 4.6 indicates, this is the case. There is also possible to see that the decryption circuit goes active when en_in is set to '1'.

Private matrix

The output from the stored values depend on the counter cnt. Figure 4.7 shows that it takes six clock cycles before the value from storage is calculated from the cnt value.

The XOR operations seem to work as intended (figure 4.8). The current value, that is an AND between input and the calculated private key value are XORed down to one bit.
Figure 4.4: Public matrix calculation

Figure 4.5: Public matrix bit-by-bit XOR
Figure 4.6: Simulation of startup

Figure 4.7: Stored private key as a function

Figure 4.8: XOR procedure
Dobbertin ROM

As described earlier, the Dobbertin component calculates its output based on the input. Figure 4.9 verifies that that is the case.

Figure 4.9: Dobbertin calculation

Sequencer

When the sequencer goes active (en_in = '1'), the module starts its work. The temporary reg_in splits the 160 bit input into 32 vectors of length 5 (figure 4.10). Since the sequencer has not been subject to any minimization, this implementation is unchanged from the TTM4530 project.

Figure 4.10: Sequencer startup

Master ROM

The output is calculated from the input. There are eight sets of equations, and ctrl determines which set to use. Figure 4.11 shows which value to be sent, and it is marked.
Figure 4.11: Master ROM calculation
Chapter 5

Discussion and conclusion

In this chapter the results will be analyzed and discussed, and a conclusion will be formed on basis of the discussion section.

5.1 Discussion

5.1.1 Synthesis

Encryption

The XST was unfortunately unable to synthesize the encryption procedure, due to the complexity of the design. XST ran for almost 72 hours, eventually the computer crashed. The computer used for synthesis of encryption had an Intel Core 2 Duo T7300, 2.0 GHz, 4 MB cache with 4 GB of system Random Access Memory (RAM) and 4 GB of virtual memory, or swap. Ubuntu 9.04 “Jaunty Jackalope” 64 bit version (X86_64 architecture) was the chosen operating system. On the mentioned computer, by synthesizing in a 32 bit operating system caused memory conflict, because a 32 bit operating system can only address 4 GB of memory in total. The sum of the physical and virtual memory (swap) exceeded 4 GB. It appeared that the encryption synthesis occupied all available memory, 6683 MB, causing the computer to run out of available memory. The fact that virtual memory were used, led to that the performance of XST was seriously hampered by this fact. XST eventually caused the computer to crash when it tried to synthesize the module Expansion, and the synthesis of encryption failed. XST never began to synthesize the Public Matrix It is likely that the encryption design needed more memory to work on, since all available memory, also swap were used. It is however unclear if this would have made a difference. It indicates that the original implementation also caused problems when attempts were made to implement the encryption on a single FPGA.

If the results from 4.1 should be referred to, the encryption procedure takes up an enormous amount of area. The numbers in the table are referring to one single chip, while the original implementation of the MQQ encryption used a total of four FPGAs, which means that the numbers related to number of Look-Up Table (LUT)s used, and number of slice registers can be multiplied by four. This means that a realization of encryption cannot be done on a single FPGA, not even the one intended for this thesis, which is one of the larger FPGAs on the market.

Decryption

As the synthesis results 4.1 indicate, the implementation which has been subject to ESPRESSO-II minimization actually has a higher area usage than the earlier version without the mini-
mization of the ROM blocks. This is a result that was unexpected in relation to minimization theory. One can also observe that the maximum frequency of the minimized design is lower than the unmodified version, which means that if realized in hardware, the minimized solution will run slower than the unminimized version of the Decryption implementation. In theory, the data should have been compressed, causing the data to use less area in hardware. The stored values are not the same in the minimized implementation as the old implementation from [8], but the difference between the two versions compared in number of used LUTs are too significant to only be related to the different stored values. There are also drawbacks concerning storing of data as functions rather than raw values. When storing raw data, there is a possibility that the block RAM of the FPGA may be utilized, meaning that number of used LUTs are reduced. By storing data as functions of a vector, the possibility to store it in block RAM is significantly lowered, meaning that these functions almost certainly will be stored in LUTs, hence increasing total area consumption.

5.1.2 Design behaviour simulation and verification

To fully determine the effect of minimizing the ROM blocks, the design in this report are quite similar to the design in [8], written by the same author as this report.

5.2 Conclusion

Tact that the Encryption procedure did not synthesize due to XST failure, this MQQ can not be realized in a single FPGA which was the desired goal of this assignment. Even if the original implementation of MQQ had been used for realization, it would have been impossible to realize it on one FPGA. Only for encryption, four FPGAs would have been required, which means that a single chip realization for MQQ at this time is not possible.

5.3 Future work

There is a possibility for realization of MQQ that could be investigated, a hardware/software codesign solution. It means that some parts of the design has to be implemented as software that works with the hardware implemented part. There exists a CPU implementation meant for Xilinx FPGAs, the MicroBlaze soft processor core [7]. This may allow software code to run on the soft-core CPU which could make it possible to implement MQQ on a single FPGA. But this solution will not be a pure hardware implementation, which was the intension in this thesis. But to utilize MicroBlaze, additional software is required, the Embedded Development Kit (EDK), which has to be purchased in addition to ISE.

5.4 Contributors

Mohamed El-Hadedy, PhD student at Q2S, has provided guidance and information which has been crucial in the design process.
Chapter 6

Abbreviations

MQQ  Multivariate Quadratic Quasigroups
VHSIC Very-High-Speed Integrated Circuits
VHDL VHSIC Hardware Description Language
FPGA  Field-programmable Gate Array
IDE   Integrated Development Environment
LUT   Look-Up Table
ROM   Read Only Memory
DH    Diffie-Hellman
RAM   Random Access Memory
ECC   Elliptic curve cryptography
RSA   Rivest-Shamir-Adleman
FSM   Finite State Machine
XST   Xilinx Synthesis Tool
CPU   Central Processing Unit
EDK   Embedded Development Kit
Bibliography


[2] A Public Key Block Cipher Based on Multivariate Quadratic Quasigroups
   http://arxiv.org/abs/0808.0247

[3] High Performance Implementation of a Public Key Block Cipher - MQQ, for FPGA Platforms

   http://portal.acm.org/citation.cfm?id=577427

   www.eecs.wsu.edu/~ee214/Fall2008/M4.pdf

   http://diamond.gem.valpo.edu/~dhart/ece110/espresso/tutorial.html


[8] TTM4530 report, FPGA implementetion of a public key block cipher, report located in the digital attachment
Appendix A

VHDL source code

In this chapter shortened versions of the VHDL source code is listed. Full versions of the source code files are located in the digital attachment. This is done because several source code files have over 10000 lines of code.

A.1 Encryption

Listing A.1: Encryption top module
signal cnt : integer range 0 to 80;
signal en_in_exp, en_in_pub : std_logic;
signal en_out_exp, en_out_pub : std_logic;
signal first_data : std_logic_vector(159 downto 0);
signal second_data : std_logic_vector(80 downto 0);
signal final_result : std_logic_vector(159 downto 0);
signal ready : std_logic;

begin

EXP: expander
port map(
clk => clk,
reset => reset,
en_in => en_in_exp,
input => datain,
output_1 => first_data,
output_2 => second_data,
en_out => en_out_exp)
);

PUB MATR: public_matrix
port map(
clk => clk,
reset => reset,
en_in => en_in_pub,
input_1 => first_data,
input_2 => second_data,
cnt => cnt,
output => final_result,
en_out => en_out_pub,
done => ready)
);

sync_run : process(clk)
begin
if (clk'event and clk = '1') then
if (reset = '1') then
state <= IDLE;
dataout <= (others => '0');
elif (en_in = '1') then
state <= next_state;
dataout <= final_result;
end if;
end if;
end process;

output_decode: process(clk, state)
begin
if (clk'event and clk = '1') then
if (reset = '1') then
en_in_exp <= '0';
en_in_pub <= '0';
elif (en_in = '1') then
case (state) is
when IDLE =>
en_in_exp <= '0';
en_in_pub <= '0';
when EXPAND =>
en_in_exp <= '1';
en_in_pub <= '0';
when PUB_MATR =>
en_in_exp <= '0';
en_in_pub <= '1';
if (en_out_exp = '1') then
cnt <= cnt + 1;
if (cnt = 79) then
cnt <= cnt + 0;
end if;
end if;
when RES =>
dataout <= final_result;
when others =>
nul;
end case;
end if;
end if;
end process;

next_state_decode: process(state, en_out_exp, ready)
begin
next_state <= state;
case (state) is
when IDLE =>
  next_state <= EXPAND;
when EXPAND =>
  if (en_out_exp <= '1') then
    next_state <= PUBLMATR;
  elsif (ready = '1') then
    next_state <= RES;
  else
    next_state <= EXPAND;
  end if;
when PUBLMATR =>
  if (en_out_pub <= '1') then
    next_state <= EXPAND;
  elsif (ready = '1') then
    next_state <= RES;
  else
    next_state <= PUBLMATR;
  end if;
when RES =>
  next_state <= IDLE;
when others =>
null;
end case;
end process;
end rtl;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
-- library UNISIM;
-- use UNISIM.VComponents.all;

entity expander is
  port (
    clk, reset, en_in : in std_logic;
    input : in std_logic_vector(159 downto 0);
    output_1 : out std_logic_vector(159 downto 0);
    output_2 : out std_logic_vector(80 downto 0);
    en_out : out std_logic);
end expander;

architecture rtl of expander is

begin
  run : process(clk)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        state <= IDLE;
      elsif (en_in = '1') then
        state <= next_state;
      end if;
    end if;
  end process;

  output_decode : process (clk, state)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        vector <= (others => '0');
        en_out <= '0';
      end if;
    end process;
end rtl;
tmp <= (others => '0');
inc <= 0;
output_1 <= (others => '0');
output_2 <= (others => '0');
if (en_in = '1') then
when IDLE =>
tmp <= input;
en_out <= '0';
when CARRY =>
vector(0) <= '1';
vector(160 downto 1) <= tmp;
vector(319 downto 161) <= tmp(158 downto 0) and input(159 downto 1);
vector(477 downto 320) <= tmp(157 downto 0) and input(159 downto 2);
vector(634 downto 478) <= tmp(156 downto 0) and input(159 downto 3);
vector(790 downto 635) <= tmp(155 downto 0) and input(159 downto 4);
vector(945 downto 791) <= tmp(154 downto 0) and input(159 downto 5);
vector(1099 downto 946) <= tmp(153 downto 0) and input(159 downto 8);
vector(1252 downto 1100) <= tmp(152 downto 0) and input(159 downto 7);
vector(1404 downto 1253) <= tmp(151 downto 0) and input(159 downto 8);
vector(1555 downto 1405) <= tmp(150 downto 0) and input(159 downto 9);
vector(1705 downto 1556) <= tmp(149 downto 0) and input(159 downto 10);
vector(1854 downto 1706) <= tmp(148 downto 0) and input(159 downto 11);
vector(2002 downto 1853) <= tmp(147 downto 0) and input(159 downto 12);
vector(2149 downto 1999) <= tmp(146 downto 0) and input(159 downto 13);
vector(2295 downto 2150) <= tmp(145 downto 0) and input(159 downto 14);
vector(2440 downto 2296) <= tmp(144 downto 0) and input(159 downto 15);
vector(2584 downto 2439) <= tmp(143 downto 0) and input(159 downto 16);
vector(2727 downto 2585) <= tmp(142 downto 0) and input(159 downto 17);
vector(2869 downto 2728) <= tmp(141 downto 0) and input(159 downto 18);
vector(3010 downto 2870) <= tmp(140 downto 0) and input(159 downto 19);
vector(3150 downto 3011) <= tmp(139 downto 0) and input(159 downto 20);
vector(3289 downto 3151) <= tmp(138 downto 0) and input(159 downto 21);
vector(3427 downto 3290) <= tmp(137 downto 0) and input(159 downto 22);
vector(3564 downto 3428) <= tmp(136 downto 0) and input(159 downto 23);
vector(3700 downto 3565) <= tmp(135 downto 0) and input(159 downto 24);
vector(3835 downto 3701) <= tmp(134 downto 0) and input(159 downto 25);
vector(3969 downto 3836) <= tmp(133 downto 0) and input(159 downto 26);
vector(4102 downto 3970) <= tmp(132 downto 0) and input(159 downto 27);
vector(4234 downto 4103) <= tmp(131 downto 0) and input(159 downto 28);
vector(4365 downto 4235) <= tmp(130 downto 0) and input(159 downto 29);
vector(4495 downto 4366) <= tmp(129 downto 0) and input(159 downto 30);
vector(4624 downto 4496) <= tmp(128 downto 0) and input(159 downto 31);
vector(4752 downto 4625) <= tmp(127 downto 0) and input(159 downto 32);
vector(4879 downto 4753) <= tmp(126 downto 0) and input(159 downto 33);
vector(5005 downto 4880) <= tmp(125 downto 0) and input(159 downto 34);
vector(5130 downto 5006) <= tmp(124 downto 0) and input(159 downto 35);
vector(5254 downto 5131) <= tmp(123 downto 0) and input(159 downto 36);
vector(5377 downto 5255) <= tmp(122 downto 0) and input(159 downto 37);
vector(5499 downto 5378) <= tmp(121 downto 0) and input(159 downto 38);
vector(5620 downto 5500) <= tmp(120 downto 0) and input(159 downto 39);
vector(5740 downto 5621) <= tmp(119 downto 0) and input(159 downto 40);
vector(5859 downto 5741) <= tmp(118 downto 0) and input(159 downto 41);
vector(5977 downto 5860) <= tmp(117 downto 0) and input(159 downto 42);
vector(6094 downto 5978) <= tmp(116 downto 0) and input(159 downto 43);
vector(6210 downto 6095) <= tmp(115 downto 0) and input(159 downto 44);
vector(6325 downto 6211) <= tmp(114 downto 0) and input(159 downto 45);
vector(6439 downto 6326) <= tmp(113 downto 0) and input(159 downto 46);
vector(6552 downto 6440) <= tmp(112 downto 0) and input(159 downto 47);
vector(6664 downto 6553) <= tmp(111 downto 0) and input(159 downto 48);
vector(6775 downto 6665) <= tmp(110 downto 0) and input(159 downto 49);
vector(6885 downto 6776) <= tmp(109 downto 0) and input(159 downto 50);
vector(6994 downto 6886) <= tmp(108 downto 0) and input(159 downto 51);
vector(7102 downto 6995) <= tmp(107 downto 0) and input(159 downto 52);
vector(7209 downto 7103) <= tmp(106 downto 0) and input(159 downto 53);
vector(7315 downto 7210) <= tmp(105 downto 0) and input(159 downto 54);
vector(7420 downto 7316) <= tmp(104 downto 0) and input(159 downto 55);
vector(7524 downto 7421) <= tmp(103 downto 0) and input(159 downto 56);
vector(7627 downto 7525) <= tmp(102 downto 0) and input(159 downto 57);
vector(7729 downto 7627) <= tmp(101 downto 0) and input(159 downto 58);
vector(7830 downto 7730) <= tmp(100 downto 0) and input(159 downto 59);
vector(7930 downto 7831) <= tmp(99 downto 0) and input(159 downto 60);
vector(8029 downto 7931) <= tmp(98 downto 0) and input(159 downto 61);
vector(8127 downto 8030) <= tmp(97 downto 0) and input(159 downto 62);
vector(8224 downto 8128) <= tmp(96 downto 0) and input(159 downto 63);
vector(8320 downto 8224) <= tmp(95 downto 0) and input(159 downto 64);
vector(8415 downto 8321) <= tmp(94 downto 0) and input(159 downto 65);
vector(8509 downto 8416) <= tmp(93 downto 0) and input(159 downto 66);
vector(8602 downto 8510) <= tmp(92 downto 0) and input(159 downto 67);
vector(8704 downto 8610) <= tmp(91 downto 0) and input(159 downto 68);
vector(8785 downto 8695) <= tmp(90 downto 0) and input(159 downto 69);
vector(8875 downto 8786) <= tmp(89 downto 0) and input(159 downto 70);
vector(8964 downto 8876) <= tmp(88 downto 0) and input(159 downto 71);
vector(9052 downto 8965) <= tmp(87 downto 0) and input(159 downto 72);
Listing A.3: Public Matrix

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
-- library UNISIM;
-- use UNISIM.VComponents.all;

entity public_matrix is
  port ( clk, reset, en_in : in std_logic;
         input_1 : in std_logic_vector(159 downto 0);
         input_2 : in std_logic_vector(80 downto 0);
         cnt : in integer range 0 to 80;
         output : out std_logic_vector(159 downto 0);
         en_out : out std_logic;
         done : out std_logic
       );
end public_matrix;

architecture rtl of public_matrix is
  type states is (IDLE, MATR_XOR160, MATR_XOR1, SYNC_160, SYNC_1, SYNC, SEND);
  type pr_rom is array (79 downto 0) of std_logic_vector(159 downto 0);
  type pr2_rom is array (0 downto 0) of std_logic_vector(80 downto 0);
  constant rom160_1 : pr_rom := (others => '0');
  constant rom1 : std_logic_vector(80 downto 0) := (others => '0');
  constant rom81_1 : std_logic_vector(80 downto 0) := (others => '0');
  constant rom81_2 : std_logic_vector(80 downto 0) := (others => '0');
end rtl;
```
signal and_rom_1_160 : std_logic_vector(159 downto 0);
signal and_rom_1_81  : std_logic_vector(80 downto 0);
signal xor1_160    : std_logic_vector(159 downto 0);
signal sync_xor1_160 : std_logic_vector(159 downto 0);

signal indata_1 : std_logic_vector(159 downto 0);
signal indata_2 : std_logic_vector(80 downto 0);
signal result   : std_logic_vector(159 downto 0);
signal xor1_1    : std_logic;
signal xor1_2    : std_logic;
signal sync_xor1_1 : std_logic;
signal sync_xor1_2 : std_logic;

signal cnt       : integer range 0 to 80;
signal cnt_2     : integer range 0 to 160;
signal state, next_state : states;

begin
  sync_proc : process(clk)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        state <= IDLE;
      else
        state <= next_state;
      end if;
    end if;
  end process;

  out_decode : process(state, clk)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        indata_1 <= (others => '0');
        indata_2 <= (others => '0');
        and_rom1_160 <= (others => '0');
        and_rom1_81 <= (others => '0');
        xor1_160  <= (others => '0');
        xor1_1    <= '0';
        sync_xor1_160 <= (others => '0');
        xor1_2    <= '0';
      else
        indata_1 <= (others => '0');
        indata_2 <= (others => '0');
        and_rom1_160 <= (others => '0');
        and_rom1_81 <= (others => '0');
        xor1_160  <= (others => '0');
        xor1_1    <= '0';
        sync_xor1_160 <= (others => '0');
        xor1_2    <= '0';
      end if;
    end if;
  end process;
xor1_2 <= '0';
sync_xor1_1 <= '0';
sync_xor1_2 <= '0';
cnt_2 <= 0;
en_out <= '0';
done <= '0';
output <= {others => '0'};
result <= {others => '0'};

else
  case (state) is
    when IDLE =>
      indata_1 <= input_1;
      indata_2 <= input_2;
      en_out <= '0';
      done <= '0';
      when MATR_AND =>
        and_rom1_160 <= rom160_1(cnt) and indata_1;
        if (cnt <= 79) then
          and_rom1_81 <= rom81_1 and indata_2;
          end if;
        when MATR_XOR160 =>
          if (cnt = 0) then
            sync_xor1_160 <= and_rom1_160;
            en_out <= '1';
            else
              xor1_160 <= and_rom1_160 xor sync_xor1_160;
          end if;
        when MATR_XOR1 =>
          en_out <= '0';
          if (cnt_2 = 0) then
            sync_xor1_1 <= sync_xor1_160(cnt_2);
            sync_xor1_2 <= and_rom1_81(cnt_2);
            else
              xor1_1 <= sync_xor1_160(cnt_2) xor sync_xor1_1;
              if (cnt_2 <= 80) then
                xor1_2 <= sync_xor1_2 xor and_rom1_81(cnt_2);
              end if;
          end if;
        when SYNC_160 =>
          sync_xor1_160 <= xor1_160;
          en_out <= '1';
        when SYNC_1 =>
          sync_xor1_1 <= xor1_1;
          sync_xor1_2 <= xor1_2;

52
A.2 Decryption

Listing A.4: Decryption top module
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity decryption is
port (
  clk : in std_logic;
  reset : in std_logic;
  en_in : in std_logic;
  inputs : in std_logic_vector (159 downto 0);
  outputs : out std_logic_vector (159 downto 0);
  en_out : out std_logic
);
end decryption;

architecture rtl of decryption is

type states is (IDLE, SEND, TO_PM, T, RECV, FROM_PM, T, SEND, TO_DOB,
RECV, FROM_DOB, SEND, TO_SEQ, RECV, FROM_SEQ,
SEND, TO_PM, S, RECV, FROM_PM, S, SEND);

component private_matrix_s
port (
  clk : in std_logic;
  reset : in std_logic;
  en_in : in std_logic;
  input : in std_logic_vector (4 downto 0);
  cnt : in std_logic_vector (4 downto 0);
  output : out std_logic_vector (159 downto 0);
  en_out : out std_logic
);
end component;

component dobbertin_rom -- defining the Dobbertin component
  -- in the decryption top level
port (
  clk, reset, en_in : in std_logic;
  db : out std_logic_vector(12 downto 0);
  en_out : out std_logic
);
end component;

component sequencer
port (
  clk , reset, en_in : in std_logic;
  input, seq : in std_logic_vector (159 downto 0);
  en_out : out std_logic;
  output_seq : out std_logic_vector (159 downto 0)
);
end component;

signal state, next_state : states;

begin
  -- signal dec_output : std_logic_vector (159 downto 0);
  signal result_pmt : std_logic_vector (159 downto 0);
  signal result_pms : std_logic_vector (159 downto 0);
  signal result_seq : std_logic_vector (159 downto 0);
  signal seq_in : std_logic_vector (159 downto 0);
  signal dob_vector_in : std_logic_vector (12 downto 0);
  signal dob_vector_out : std_logic_vector (12 downto 0);

  signal count_pmt : std_logic_vector (4 downto 0);
  signal count_pms : std_logic_vector (4 downto 0);
  signal count_seq : std_logic;
  signal en_in_pmt : std_logic;
  signal en_in_pms : std_logic;
  signal en_in_seq : std_logic;
  signal en_out_seq : std_logic;
  signal en_out_pmt : std_logic;
  signal en_out_pms : std_logic;
  signal en_out_dob : std_logic;

  -- counter private_matrix t
  -- counter private_matrix s

  state <= next_state;
end begin;

DR: DOBBERTIN_ROM -- initializing the Dobbertin
PORT MAP (clk => clk,
reset => reset,
en_in => en_in_dob,
en_out => en_out_dob,
z => dob_vector_in,
db => dob_vector_out);

PM_T: PRIVATE MATRIX -- initializing the Private Matrix
PORT MAP (clk => clk,
reset => reset,
en_in => en_in_pm_t,
input => shift_pmt,
cnt => count_t,
output => result_pmt,
en_out => en_out_pm_t);

SEQ: SEQUENCER -- initializing the sequencer
PORT MAP (clk => clk,
reset => reset,
en_in => en_in_seq,
in_seq => seq_in,
en_out => en_out_seq,
output_seq => result_seq);

PM_S: PRIVATE MATRIX -- initializing the Private Matrix
PORT MAP (clk => clk,
reset => reset,
en_in => en_in_pm_s,
input => shift_pms,
cnt => count_s,
output => result_pms,
en_out => en_out_pm_s);

running: process (clk, en_in)
begin
if (clk'event and clk = '1') then
if (reset = '1') then
state <= IDLE;
elsif (en_in = '1') then
state <= next_state;
end if;
end if;
end process;

output_dec: process (clk, en_in, state)
begin
if (clk'event and clk = '1') then
if (reset = '1') then
dob_vector_in <= (others => '0');
dc_in <= (others => '0');
result_pmt <= (others => '0');
result_pms <= (others => '0');
result_seq <= (others => '0');
output <= (others => '0');
shift_pmt <= (others => '0');
shift_pms <= (others => '0');
en_out <= '0';
en_in_dob <= '0';
en_in_pm_t <= '0';
en_in_pm_s <= '0';
en_out_dob <= '0';
en_out_seq <= '0';
end if;
end if;
end process;
count_s <= (others => '0');
seq_in <= (others => '0');

elsif (en_in = '1') then
  case (state) is
  when IDLE =>
  -- decode input <=
  en_in_pm_t <= '0';
  if (count_t <= "1111") then
    count_t <= (others => '0');
  end if;

  when SEND_TO_PM_T =>
  en_in_pm_t <= '1';
  if (count_t = "1111" and en_out_pm_t = '1') then
    null;
  else
    shift_pm <= inputs (((conv_integer(count_t)*5)) downto 0+(conv_integer(count_t)*5))
    if (en_out_pm_t = '1') then
      count_t <= count_t + 1,
    end if;
  end if;

  when RECV_FROM_PM_T =>
  case (result_pm_t) is
  when ("ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
if (en_out_pm_s = '1') then
  count_s <= count_s + 1;
end if;
end if;

when RECV_FROM_PM_S =>
  en_in_pm_s <= '0';
  dec_output <= result_pms;
when SEND =>
  outputs <= dec_output;
when others =>
  null;
end case;
end if;
end process;

-- calculates next state
next_state := process(state, count_t, en_out_pm_t, en_out_seq, count_s, en_out_pm_s)
begin
  next_state <= state;
  case (state) is
    when IDLE =>
      next_state <= SEND_TO_PM_T;
    when RECV_FROM_PM_T =>
      if (count_t = 31 and en_out_pm_t = '1') then
        next_state <= RECV_FROM_PM_T;
      else
        next_state <= SEND_TO_PM_T;
      end if;
      whenothers paragraph continues...
    when SEND_TO_SEQ =>
      case (en_out_seq) is
        when '0' =>
          next_state <= SEND_TO_SEQ;
        when '1' =>
          next_state <= RECV_FROM_SEQ;
        when others =>
          next_state <= IDLE;
      end case;
    when RECV_FROM_SEQ =>
      case (en_out_seq) is
        when '0' =>
          next_state <= SEND_TO_SEQ;
        when '1' =>
          next_state <= RECV_FROM_SEQ;
        when others =>
          next_state <= IDLE;
      end case;
    when SEND_TO_PM_S =>
      if (count_s = 31 and en_out_pm_s = '1') then
        next_state <= RECV_FROM_PM_S;
      else
        next_state <= SEND_TO_PM_S;
      end if;
    when RECV_FROM_PM_S =>
      case (result_pms) is
        when ('1' =>
          next_state <= SEND_TO_PM_S;
        when others =>
          null;
      end case;
end when;
end process;


Listing A.5: Private Matrix

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity private_matrix is
  port(
    clk : in std_logic;
    reset : in std_logic;
    en_in : in std_logic;
    cnt : in std_logic_vector(4 downto 0);
    input : in std_logic_vector(4 downto 0);
    output : out std_logic_vector(159 downto 0);
    en_out : out std_logic;
  );
end private_matrix;

architecture rtl of private_matrix is
  type priv_state is (IDLE, ANDOP, XORING, SYNC, PUSHOUT);
  signal rom_001 : std_logic_vector(4 downto 0);
  signal rom_002 : std_logic_vector(4 downto 0);
  signal and_rom_in_1 : std_logic_vector(4 downto 0);
  signal and_rom_in_2 : std_logic_vector(4 downto 0);
  signal tmp : std_logic_vector(159 downto 0);
  signal count_xor : integer range 0 to 4 := 0;
  signal matr_xor : std_logic_vector(159 downto 0);
  signal sync_xor : std_logic_vector(159 downto 0);
  signal enable_out : std_logic;
  signal state, next_state : priv_state;
  signal register_x : std_logic_vector(159 downto 0);
begin
  sync_run : process(clk, en_in, enable_out, register_x)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        output <= (others => '0');
        en_out <= '0';
        state <= IDLE;
      elsif (en_in = '1') then
        state <= next_state;
        en_out <= enable_out;
        if (cnt = "11111" and enable_out = '1') then
          output <= register_x;
        end if;
      end if;
    end if;
    end if;
  end process;

  next_state <= IDLE;
  when others =>
    next_state <= SEND;
  end case;

  next_state <= IDLE;
  when others =>
    next_state <= IDLE;
  end case;
end process;
end rtl;
-- output_dec is made synchronous, so that all signals remember their
-- value at all times when not set again
output_dec : process (clk, state, cnt)
begin
if (clk'event and clk = '1') then
  if (reset = '1') then
    output <= (others => '0');
    enable_out <= '0';
    count_xor <= 0;
    tmp <= (others => '0');
    matr_xor <= (others => '0');
    sync xor <= (others => '0');
    register_x <= (others => '0');
    and_rom_in_l <= "00000";
  elsif (en_in = '1') then
    case (state) is
    when IDLE =>
      -- makes sure that en_out port is set to '0'
      enable_out <= '0';
      -- output <= (others => '0');
    when rom01(4) <= (not cnt(2) and cnt(1) and cnt(0)) or (cnt(4) and not cnt(3) and not cnt(2) and cnt(1)) or (not cnt(4) and not cnt(3) and cnt(1) and cnt(0)) or (cnt(4) and not cnt(1) and not cnt(0)) or (cnt(4) and cnt(3) and cnt(1) and cnt(0)) or (not cnt(4) and cnt(3) and cnt(1) and not cnt(0)) or (not cnt(2) and not cnt(1) and not cnt(0));
    when rom01(3) <= (not cnt(4) and not cnt(3) and not cnt(2) and cnt(0)) or (not cnt(4) and cnt(3) and cnt(1)) and not cnt(0) or (not cnt(4) and not cnt(2) and not cnt(1) and cnt(0)) or (not cnt(4) and not cnt(2) and cnt(1) and cnt(0)) or (not cnt(4) and cnt(3) and not cnt(0)) or (not cnt(3) and not cnt(1) and not cnt(0)) or (not cnt(4) and cnt(3) and not cnt(0)) or (not cnt(2) and not cnt(1) and not cnt(0)) or (not cnt(3) and not cnt(1) and not cnt(0));
    when rom01(2) <= (cnt(3) and cnt(2) and not cnt(0)) or (cnt(4) and cnt(2) and not cnt(1) and cnt(0)) or (not cnt(3) and cnt(2) and cnt(1)) or (cnt(4) and not cnt(2) and cnt(1) and cnt(0)) or (not cnt(3) and cnt(2) and cnt(1)) or (not cnt(4) and cnt(2) and cnt(1)) or (not cnt(4) and cnt(2) and cnt(1)) or (not cnt(3) and not cnt(1) and not cnt(0)) or (not cnt(4) and cnt(3) and not cnt(0)) or (not cnt(2) and not cnt(1) and not cnt(0));
    when rom01(1) <= (cnt(3) and cnt(2) and not cnt(1) and cnt(0)) or (cnt(3) and cnt(2) and not cnt(1) and cnt(0)) or (not cnt(4) and cnt(2) and cnt(1) and cnt(0)) or (not cnt(4) and not cnt(3) and not cnt(1)) and not cnt(0) or (not cnt(4) and not cnt(3) and not cnt(2) and cnt(0)) or (not cnt(4) and not cnt(3) and not cnt(2) and not cnt(1)) or (not cnt(4) and not cnt(3) and not cnt(2) and not cnt(1)) or (not cnt(4) and not cnt(3) and not cnt(2) and not cnt(1));
    when rom01(0) <= (not cnt(3) and not cnt(1) and not cnt(0)) or (not cnt(3) and not cnt(2) and not cnt(1) and cnt(0)) or (cnt(4) and not cnt(2) and not cnt(1) and cnt(0)) or (not cnt(3) and not cnt(2) and not cnt(1) and cnt(0)) or (not cnt(3) and not cnt(2) and not cnt(1) and cnt(0));
    when AND OP =>
      -- makes the logical operation between the input
      -- value and the stored value inside the respective
      -- ROM
    and_rom_in_l <= rom01 and input;
end if;
when XORING =>
  -- the actual bit-by-bit xor operation
  tmp(0) <= and(rom_in,1(count_xor));
  matr_xor(0) <= and(rom_in,1(count_xor+1)) xor tmp(0);

if not (count_xor = 3) then
  count_xor <= count_xor + 1;
end if;

when SYNC =>
  -- sets the xor counter to 0
  count_xor <= 0;
  -- keep synchronization of the xor'ed bits
  if (cnt = "00000") then
    sync_xor <= matr_xor;
    enable_out <= '1';
  else
    sync_xor <= matr_xor xor sync_xor;
    enable_out <= '1';
  end if;

when PUSH_OUT =>
  -- pushes out the processed vector on output
  enable_out <= '1';
  register_x <= sync_xor;

when others =>
  enable_out <= '0';
  rom_001 <= (others => '0');

end case;
end if;
end process;

next_state_dec: process (state, input, cnt, count_xor)
begin
  next_state <= state;
  case (state) is
  when IDLE =>
    -- makes sure that rom has the correct value
    -- as to the running time given by signal cnt
    next_state <= ANDOP;
  when ANDOP =>
    next_state <= XORING;
  when XORING =>
    if (count_xor < 3) then
      next_state <= XORING;
    else
      next_state <= SYNC;
    end if;
  when SYNC =>
    if (cnt = "11111") then
      next_state <= PUSH_OUT;
    else
      next_state <= IDLE;
    end if;
  when PUSH_OUT =>
    next_state <= IDLE;
  when others =>
    next_state <= IDLE;
  end case;
end process;

end rtl;

Listing A.6: Dobbertin ROM
entity Dobbertin_ROM is
port (
  x : in std_logic_vector (12 downto 0);
  clk, reset, en_in : in std_logic;
  db : out std_logic_vector(12 downto 0);
  en_out : out std_logic
); end Dobbertin_ROM;
architecture rtl of Dobbertin_ROM is
begin
process(clk, reset, en_in)
begin
  if(clk'event and clk = '1') then
    if (reset = '1') then
      db <= (others => '0');
      en_out <= '0';
    elsif (en_in = '1') then
      en_out <= '1';
      db(12) <= (not x(12) and not x(11) and not x(10) and not x(9) and not x(8) and not x(7) and
                 x(6) and x(4) and not x(3) and not x(0)) or (.... and x(10) and not x(9) and not x(7)
                 and x(6) and x(5) and not x(4) and x(3) and x(1) and x(0));
    else
      en_out <= '0';
    end if;
  end if;
end process;
end rtl;

Listing A.7: Sequencer
begin
  counter : in std_logic_vector (4 downto 0);
db : out std_logic_vector (4 downto 0) -- data bus
end component;

type reg_input is array ((5-1) downto (0)) of std_logic_vector (4 downto 0);
type states is (IDLE, MUX2_SEL, SEND_TO_MASTER, RECVR_M, MUX3_SEL, SYNC, PUSH);
signal reg_input, reg_out : reg_input;
signal sel : std_logic;
signal counter_1 : std_logic_vector (4 downto 0);
signal counter_2 : std_logic_vector (4 downto 0);
signal mux3_out : std_logic_vector (4 downto 0);
signal mux2_out, sync, mr : std_logic_vector (4 downto 0);
signal to_master : std_logic_vector (9 downto 0);
signal from_master : std_logic_vector (4 downto 0);
signal en_out_mux : std_logic;
signal state, next_state : states;
begin
  MA,R: master_rom -- initializing the master rom
    -- component in the sequencer
    -- circuit
    port map(
      clk => clk,
      reset => reset,
      counter => counter_2,
      en_in => en_out_mux,
      en_out => en_out_mux,
      s => to_master,
      db => from_master
    );
  sec_run : process (clk, en_in, counter_1, counter_2)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        state <= IDLE;
      elseif (en_in = '1') then
        state <= next_state;
      end if;
    end if;
  end process;
  output_dec: process (clk, en_in, state)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        reg_in(0) <= (others => '0');
        reg_in(1) <= (others => '0');
        reg_in(2) <= (others => '0');
        reg_in(3) <= (others => '0');
        reg_in(4) <= (others => '0');
        reg_in(5) <= (others => '0');
        reg_in(6) <= (others => '0');
        reg_in(7) <= (others => '0');
        reg_in(8) <= (others => '0');
        reg_in(9) <= (others => '0');
        reg_in(10) <= (others => '0');
        reg_in(11) <= (others => '0');
        reg_in(12) <= (others => '0');
        reg_in(13) <= (others => '0');
        reg_in(14) <= (others => '0');
        reg_in(15) <= (others => '0');
        reg_in(16) <= (others => '0');
        reg_in(17) <= (others => '0');
        reg_in(18) <= (others => '0');
        reg_in(19) <= (others => '0');
        reg_in(20) <= (others => '0');
        reg_in(21) <= (others => '0');
        reg_in(22) <= (others => '0');
        reg_in(23) <= (others => '0');
        reg_in(24) <= (others => '0');
        reg_in(25) <= (others => '0');
        reg_in(26) <= (others => '0');
        reg_in(27) <= (others => '0');
  end process;
reg_in(28) <= (others => '0');
reg_in(29) <= (others => '0');
reg_in(30) <= (others => '0');
reg_in(31) <= (others => '0');
to_master <= (others => '0');
en_out <= '0';
month <= '0';
out_mux <= (others => '0');
sync_mr <= (others => '0');
mux_out <= (others => '0');
---from_master <= (others => '0');
output_seq <= (others => '0');
reg_out <= reg_in;
counter_1 <= "00000";
counter_2 <= "00000";
elseif (en_in = '1') then
  case (state) is
  when IDLE =>
    reg_in(0) <= input_seq(4 downto 0);
    reg_in(1) <= input_seq(9 downto 5);
    reg_in(2) <= input_seq(14 downto 10);
    reg_in(3) <= input_seq(19 downto 15);
    reg_in(4) <= input_seq(24 downto 20);
    reg_in(5) <= input_seq(29 downto 25);
    reg_in(6) <= input_seq(34 downto 30);
    reg_in(7) <= input_seq(39 downto 35);
    reg_in(8) <= input_seq(44 downto 40);
    reg_in(9) <= input_seq(49 downto 45);
    reg_in(10) <= input_seq(54 downto 50);
    reg_in(11) <= input_seq(59 downto 55);
    reg_in(12) <= input_seq(64 downto 60);
    reg_in(13) <= input_seq(69 downto 65);
    reg_in(14) <= input_seq(74 downto 70);
    reg_in(15) <= input_seq(79 downto 75);
    reg_in(16) <= input_seq(84 downto 80);
    reg_in(17) <= input_seq(89 downto 85);
    reg_in(18) <= input_seq(94 downto 90);
    reg_in(19) <= input_seq(99 downto 95);
    reg_in(20) <= input_seq(104 downto 100);
    reg_in(21) <= input_seq(109 downto 105);
    reg_in(22) <= input_seq(114 downto 110);
    reg_in(23) <= input_seq(119 downto 115);
    reg_in(24) <= input_seq(124 downto 120);
    reg_in(25) <= input_seq(129 downto 125);
    reg_in(26) <= input_seq(134 downto 130);
    reg_in(27) <= input_seq(139 downto 135);
    reg_in(28) <= input_seq(144 downto 140);
    reg_in(29) <= input_seq(149 downto 145);
    reg_in(30) <= input_seq(154 downto 150);
    reg_in(31) <= input_seq(159 downto 155);
    sel <= '0';
en_out <= '0';
  when MUX2_SEL =>
    counter_1 <= counter_1 + 1;
  case (sel) is
  when '1' =>
    mux2_out <= sync_mr;
  when '0' =>
    mux2_out <= reg_in(0);
  when others =>
    mux2_out <= (others => 'Z');
  end case;
  when SEND_TO_MASTER =>
    sel <= '1';
to_master(9 downto 5) <= mux2_out;
to_master(4 downto 0) <= mux31_out;
  when RECV_MAR =>
    sync_mr <= from_master;
  when MUX31_SEL =>
  case (counter_1) is
  when "00001" =>
    mux31_out <= reg_in(1);
en_out mux <= '1';
  when "00010" =>
    mux31_out <= reg_in(2);
en_out mux <= '1';
  when "00011" =>
    mux31_out <= reg_in(3);
en_out mux <= '1';
  when "00100" =>

63
mux31_out <= reg_in(4);
en_out_max <= '1';
when "00101" =>
mux31_out <= reg_in(5);
en_out_max <= '1';
when "00110" =>
mux31_out <= reg_in(6);
en_out_max <= '1';
when "00111" =>
mux31_out <= reg_in(7);
en_out_max <= '1';
when "01000" =>
mux31_out <= reg_in(8);
en_out_max <= '1';
when "01001" =>
mux31_out <= reg_in(9);
en_out_max <= '1';
when "01010" =>
mux31_out <= reg_in(10);
en_out_max <= '1';
when "01011" =>
mux31_out <= reg_in(11);
en_out_max <= '1';
when "01100" =>
mux31_out <= reg_in(12);
en_out_max <= '1';
when "01101" =>
mux31_out <= reg_in(13);
en_out_max <= '1';
when "01110" =>
mux31_out <= reg_in(14);
en_out_max <= '1';
when "01111" =>
mux31_out <= reg_in(15);
en_out_max <= '1';
when "10000" =>
mux31_out <= reg_in(16);
en_out_max <= '1';
when "10001" =>
mux31_out <= reg_in(17);
en_out_max <= '1';
when "10010" =>
mux31_out <= reg_in(18);
en_out_max <= '1';
when "10011" =>
mux31_out <= reg_in(19);
en_out_max <= '1';
when "10100" =>
mux31_out <= reg_in(20);
en_out_max <= '1';
when "10101" =>
mux31_out <= reg_in(21);
en_out_max <= '1';
when "10110" =>
mux31_out <= reg_in(22);
en_out_max <= '1';
when "10111" =>
mux31_out <= reg_in(23);
en_out_max <= '1';
when "11000" =>
mux31_out <= reg_in(24);
en_out_max <= '1';
when "11001" =>
mux31_out <= reg_in(25);
en_out_max <= '1';
when "11010" =>
mux31_out <= reg_in(26);
en_out_max <= '1';
when "11011" =>
mux31_out <= reg_in(27);
en_out_max <= '1';
when "11100" =>
mux31_out <= reg_in(28);
en_out_max <= '1';
when "11101" =>
mux31_out <= reg_in(29);
en_out_max <= '1';
when "11110" =>
mux31_out <= reg_in(30);
en_out_max <= '1';
when "11111" =>
mux31_out <= reg_in(31);
en_out_max <= '1';


begin
end process;

next_state_dec : process(state, counter_1, counter_2)
begin
next_state <= state;
case (state) is
when IDLE =>
case (counter_1) is
when "00000" =>
next_state <= MUX2_SEL;
when others =>
next_state <= MUX31_SEL;
end case;
when MUX2_SEL =>
next_state <= SYNC;
when SEND_TO_MASTER =>
next_state <= HEVY_MR;
when HEVY_MR =>
next_state <= MUX2_SEL;
when MUX31_SEL =>
next_state <= SEND_TO_MASTER;
when SYNC =>
if (counter_2 < "11111") then
next_state <= PUSH;
else
next_state <= MUX31_SEL;
end if;
when PUSH =>
next_state <= IDLE;
end case;
end process;

next_state <= state;
case (state) is
when IDLE =>
case (counter_1) is
when "00000" =>
next_state <= MUX2_SEL;
when others =>
next_state <= MUX31_SEL;
end case;
when MUX2_SEL =>
next_state <= SYNC;
when SEND_TO_MASTER =>
next_state <= HEVY_MR;
when HEVY_MR =>
next_state <= MUX2_SEL;
when MUX31_SEL =>
next_state <= SEND_TO_MASTER;
when SYNC =>
if (counter_2 < "11111") then
next_state <= PUSH;
else
next_state <= MUX31_SEL;
end if;
when PUSH =>
next_state <= IDLE;
end case;
end process;
library IEEE;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity master_rom is
  port(
    clk, reset, en_in : in std_logic;
    z     : in std_logic_vector(9 downto 0); -- address bus
    counter : in std_logic_vector(4 downto 0);
    db     : out std_logic_vector(4 downto 0) -- data bus
  );
end master_rom;

architecture rtl of master_rom is
  signal cnt : std_logic_vector(4 downto 0);
  signal ctrl : std_logic_vector(2 downto 0);
begin
  sel_rom : process (clk, ctrl)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
        db <= (others => '0');
        cnt <= (others => '0');
      elsif (en_in = '1') then
        case (ctrl) is
          when "000" =>
            db(4) <= (not x(8) and not x(7) and not x(5) and not x(4) and not x(2) and not x(1) and not x(0)) or (x(8) and not x(7) and x(5) and not x(4) and not x(2) and not x(1) and not x(0)) or (....) and not x(5) and x(4) and x(3) and x(2) and x(1) and x(0));
          when others =>
            db <= (others => 'Z');
        end case;
      end if;
    end if;
  end process;
end master_rom;

control : process(clk)
begin
  if (clk'event and clk = '1') then
    if (reset = '1') then
      ctrl <= (others => '0');
      cnt <= (others => '0');
    elsif (counter < "11111") then
      ctrl(2) <= (not counter(1) and not counter(0)) or (not counter(2) and not counter(1)) or (not counter(3) and not counter(1)) or (not counter(4) and not counter(1));
      ctrl(1) <= (not counter(2));
      ctrl(0) <= (counter(4) and counter(3) and counter(2) and counter(1) and not counter(0)) or (not counter(2) and counter(0)) or (not counter(3) and counter(0)) or (not counter(4) and counter(0));
    else
      cnt <= (others => '0');
    end if;
  end if;
end process;
end rtl;
Appendix B

Espresso minimization

Here is an example of an input file for minimization through the espresso application.

Listing B.1: Control ROM espresso minimization input file

```plaintext
# ROM
Control
. i 5
. o 3
. i lb z(4) z(3) z(2) z(1) z(0)
. ob Ctrl(2) Ctrl(1) Ctrl(0)
 . p 31
00000 110
00001 111
00010 010
00011 011
00100 100
00101 101
00110 000
00111 001
01000 110
01001 111
01010 010
01011 011
01100 100
01101 101
01110 000
01111 001
10000 110
10001 111
10010 010
10011 011
10100 100
10101 101
10110 000
10111 001
11000 110
11001 111
11010 010
11011 011
11100 100
11101 000
11110 001
```

Listing B.2: Control ROM espresso minimization result file

```plaintext
# ROM_Control
Ctrl(2) = (z(1)&!z(0)) | (!z(2)&z(1)) | (!z(3)&z(1)) | (!z(4)&z(1));
Ctrl(1) = (!z(2));
Ctrl(0) = (z(4)&z(3)&z(2)&z(1)&!z(0)) | (!z(2)&z(0)) | (!z(3)&z(0)) | (!z(4)&z(0));
```

67
Appendix C

Synthesis report from Decryption

Here is the complete synthesis report of Decryption.

Listing C.1: Decryption synthesis report

```
Release 10.1 - xst K.39 (lin)
Copyright (c) 1995-2008 Xilinx, Inc.  All rights reserved.
Parameter TMPDIR set to /home/stig/Documents/ttm4900/mqq_final/xst/projnav.tmp
Total REAL time to Xst completion: 0.00 secs
Total CPU time to Xst completion: 0.06 secs
---
Parameter xsthdpdir set to /home/stig/Documents/ttm4900/mqq_final/xst
Total REAL time to Xst completion: 0.00 secs
Total CPU time to Xst completion: 0.06 secs
---
Reading design: decryption.prj

TABLE OF CONTENTS

1) Synthesis Options Summary
2) HDL Compilation
3) Design Hierarchy Analysis
4) HDL Analysis
5) HDL Synthesis
5.1) HDL Synthesis Report
6) Advanced HDL Synthesis
6.1) Advanced HDL Synthesis Report
7) Low Level Synthesis
8) Partition Report
9) Final Report
9.1) Device utilization summary
9.2) Partition Resource Summary
9.3) TIMING REPORT

---

* Synthesis Options Summary *

--- Source Parameters ---
Input File Name : "decryption.prj"
Input Format : mixed
Ignore Synthesis Constraint File : NO
--- Target Parameters ---
Output File Name : "decryption"
Output Format : NGC
Target Device : xc5vlx110t-1-ff1136
--- Source Options ---
Top Module Name : decryption
Automatic FSM Extraction : YES
FSM Encoding Algorithm : Auto
Safe Implementation : No
FMS Style : lut
RAM Extraction : Yes
RAM Style : Auto
```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM Extraction</td>
<td>Yes</td>
</tr>
<tr>
<td>Max Style</td>
<td>Auto</td>
</tr>
<tr>
<td>Decoder Extraction</td>
<td>YES</td>
</tr>
<tr>
<td>Priority Encoder Extraction</td>
<td>YES</td>
</tr>
<tr>
<td>Shift Register Extraction</td>
<td>YES</td>
</tr>
<tr>
<td>Logical Shifter Extraction</td>
<td>YES</td>
</tr>
<tr>
<td>XOR Collapsing</td>
<td>YES</td>
</tr>
<tr>
<td>ROM Style</td>
<td>Auto</td>
</tr>
<tr>
<td>ROM Extraction</td>
<td>YES</td>
</tr>
<tr>
<td>Resource Sharing</td>
<td>YES</td>
</tr>
<tr>
<td>Asynchronous To Synchronous</td>
<td>NO</td>
</tr>
<tr>
<td>Use DSP Block</td>
<td>auto</td>
</tr>
<tr>
<td>Automatic Register Balancing</td>
<td>No</td>
</tr>
<tr>
<td>Target Options</td>
<td></td>
</tr>
<tr>
<td>LUT Combining</td>
<td>off</td>
</tr>
<tr>
<td>Reduce Control Sets</td>
<td>off</td>
</tr>
<tr>
<td>Add IO Buffers</td>
<td>YES</td>
</tr>
<tr>
<td>Global Maximum Fanout</td>
<td>100000</td>
</tr>
<tr>
<td>Add Generic Clock Buffer (BUFG)</td>
<td>32</td>
</tr>
<tr>
<td>Register Duplication</td>
<td>YES</td>
</tr>
<tr>
<td>Slice Packing</td>
<td>YES</td>
</tr>
<tr>
<td>Optimize Instantiated Primitives</td>
<td>NO</td>
</tr>
<tr>
<td>Use Clock Enable</td>
<td>Auto</td>
</tr>
<tr>
<td>Use Synchronous Set</td>
<td>Auto</td>
</tr>
<tr>
<td>Use Synchronous Reset</td>
<td>Auto</td>
</tr>
<tr>
<td>Pack IO Registers into IOBs</td>
<td>auto</td>
</tr>
<tr>
<td>Equivalent register Removal</td>
<td>YES</td>
</tr>
<tr>
<td>General Options</td>
<td></td>
</tr>
<tr>
<td>Optimization Goal</td>
<td>Speed</td>
</tr>
<tr>
<td>Optimization Effort</td>
<td>1</td>
</tr>
<tr>
<td>Power Reduction</td>
<td>NO</td>
</tr>
<tr>
<td>Library Search Order</td>
<td>decryption.lso</td>
</tr>
<tr>
<td>Keep Hierarchy</td>
<td>NO</td>
</tr>
<tr>
<td>Netlist Hierarchy</td>
<td>as_optimized</td>
</tr>
<tr>
<td>RTL Output</td>
<td>YES</td>
</tr>
<tr>
<td>Global Optimization</td>
<td>AllClockNets</td>
</tr>
<tr>
<td>Read Cores</td>
<td>YES</td>
</tr>
<tr>
<td>Write Timing Constraints</td>
<td>NO</td>
</tr>
<tr>
<td>Cross Clock Analysis</td>
<td>NO</td>
</tr>
<tr>
<td>Hierarchy Separator</td>
<td>/</td>
</tr>
<tr>
<td>Bus Delimiter</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>Case Specifier</td>
<td>maintain</td>
</tr>
<tr>
<td>Slice Utilization Ratio</td>
<td>100</td>
</tr>
<tr>
<td>BRAM Utilization Ratio</td>
<td>100</td>
</tr>
<tr>
<td>DSP48 Utilization Ratio</td>
<td>100</td>
</tr>
<tr>
<td>Verilog 2001</td>
<td>YES</td>
</tr>
<tr>
<td>Auto BRAM Packing</td>
<td>NO</td>
</tr>
<tr>
<td>Slice Utilization Ratio Delta</td>
<td>5</td>
</tr>
</tbody>
</table>

**HDL Compilation**

- Compiling vhdl file "~/home/stig/Documents/htm4900/mqq_final/VHDL/master_rom.vhd" in Library work.
- Entity <masterrom> compiled
- Architecture <rtl> compiled.
- Compiling vhdl file "~/home/stig/Documents/htm4900/mqq_final/VHDL/dobbertin_rom.vhd" in Library work.
- Architecture rtl of Entity dobbertin_rom is up to date.
- Compiling vhdl file "~/home/stig/Documents/htm4900/mqq_final/VHDL/private_matrix_t.vhd" in Library work.
- Architecture rtl of Entity private_matrix_t is up to date.
- Compiling vhdl file "~/home/stig/Documents/htm4900/mqq_final/VHDL/sequencer.vhd" in Library work.
- Architecture behavioral of Entity sequencer is up to date.
- Compiling vhdl file "~/home/stig/Documents/htm4900/mqq_final/VHDL/decryption.vhd" in Library work.
- Architecture rtl of Entity decryption is up to date.

**Design Hierarchy Analysis**

- Analyzing hierarchy for entity <decryption> in library <work> (architecture <rtl>.
- Analyzing hierarchy for entity <dobbertin_rom> in library <work> (architecture <rtl>.
- Analyzing hierarchy for entity <private_matrix_a> in library <work> (architecture <rtl>.
- Analyzing hierarchy for entity <sequencer> in library <work> (architecture behavioral>.
- Analyzing hierarchy for entity <masterrom> in library <work> (architecture <rtl>.

70
305 Found 5-bit register for signal <and_rom_in_50>.
306 Found 5-bit register for signal <and_rom_in_51>.
307 Found 5-bit register for signal <and_rom_in_52>.
308 Found 5-bit register for signal <and_rom_in_53>.
309 Found 5-bit register for signal <and_rom_in_54>.
310 Found 5-bit register for signal <and_rom_in_55>.
311 Found 5-bit register for signal <and_rom_in_56>.
312 Found 5-bit register for signal <and_rom_in_57>.
313 Found 5-bit register for signal <and_rom_in_58>.
314 Found 5-bit register for signal <and_rom_in_59>.
315 Found 5-bit register for signal <and_rom_in_60>.
316 Found 5-bit register for signal <and_rom_in_61>.
317 Found 5-bit register for signal <and_rom_in_62>.
318 Found 5-bit register for signal <and_rom_in_63>.
319 Found 5-bit register for signal <and_rom_in_64>.
320 Found 5-bit register for signal <and_rom_in_65>.
321 Found 5-bit register for signal <and_rom_in_66>.
322 Found 5-bit register for signal <and_rom_in_67>.
323 Found 5-bit register for signal <and_rom_in_68>.
324 Found 5-bit register for signal <and_rom_in_69>.
325 Found 5-bit register for signal <and_rom_in_70>.
326 Found 5-bit register for signal <and_rom_in_71>.
327 Found 5-bit register for signal <and_rom_in_72>.
328 Found 5-bit register for signal <and_rom_in_73>.
329 Found 5-bit register for signal <and_rom_in_74>.
330 Found 5-bit register for signal <and_rom_in_75>.
331 Found 5-bit register for signal <and_rom_in_76>.
332 Found 5-bit register for signal <and_rom_in_77>.
333 Found 5-bit register for signal <and_rom_in_78>.
334 Found 5-bit register for signal <and_rom_in_79>.
335 Found 5-bit register for signal <and_rom_in_80>.
336 Found 5-bit register for signal <and_rom_in_81>.
337 Found 5-bit register for signal <and_rom_in_82>.
338 Found 5-bit register for signal <and_rom_in_83>.
339 Found 5-bit register for signal <and_rom_in_84>.
340 Found 5-bit register for signal <and_rom_in_85>.
341 Found 5-bit register for signal <and_rom_in_86>.
342 Found 5-bit register for signal <and_rom_in_87>.
343 Found 5-bit register for signal <and_rom_in_88>.
344 Found 5-bit register for signal <and_rom_in_89>.
345 Found 5-bit register for signal <and_rom_in_90>.
346 Found 5-bit register for signal <and_rom_in_91>.
347 Found 5-bit register for signal <and_rom_in_92>.
348 Found 5-bit register for signal <and_rom_in_93>.
349 Found 5-bit register for signal <and_rom_in_94>.
350 Found 5-bit register for signal <and_rom_in_95>.
351 Found 5-bit register for signal <and_rom_in_96>.
352 Found 5-bit register for signal <and_rom_in_97>.
353 Found 5-bit register for signal <and_rom_in_98>.
354 Found 5-bit register for signal <and_rom_in_99>.
355 Found 3-bit register for signal <count_xor>.
356 Found 3-bit adder for signal <count_xor>.
357 Found 1-bit register for signal <enable_out>.
358 Found 160-bit register for signal <matr_xor>.
359 Found 1-bit xor2 for signal <matr_xor_0000>.
360 Found 1-bit xor2 for signal <matr_xor_0001>.
361 Found 1-bit xor2 for signal <matr_xor_0002>.
362 Found 1-bit xor2 for signal <matr_xor_0003>.
363 Found 1-bit xor2 for signal <matr_xor_0004>.
364 Found 1-bit xor2 for signal <matr_xor_0005>.
365 Found 1-bit xor2 for signal <matr_xor_0006>.
366 Found 1-bit xor2 for signal <matr_xor_0007>.
367 Found 1-bit xor2 for signal <matr_xor_0008>.
368 Found 1-bit xor2 for signal <matr_xor_0009>.
369 Found 1-bit xor2 for signal <matr_xor_0010>.
370 Found 1-bit xor2 for signal <matr_xor_0011>.
371 Found 1-bit xor2 for signal <matr_xor_0012>.
372 Found 1-bit xor2 for signal <matr_xor_0013>.
373 Found 1-bit xor2 for signal <matr_xor_0014>.
374 Found 1-bit xor2 for signal <matr_xor_0015>.
375 Found 1-bit xor2 for signal <matr_xor_0016>.
376 Found 1-bit xor2 for signal <matr_xor_0017>.
377 Found 1-bit xor2 for signal <matr_xor_0018>.
378 Found 1-bit xor2 for signal <matr_xor_0019>.
379 Found 1-bit xor2 for signal <matr_xor_0020>.
380 Found 1-bit xor2 for signal <matr_xor_0021>.
381 Found 1-bit xor2 for signal <matr_xor_0022>.
382 Found 1-bit xor2 for signal <matr_xor_0023>.
383 Found 1-bit xor2 for signal <matr_xor_0024>.
384 Found 1-bit xor2 for signal <matr_xor_0025>.
385 Found 1-bit xor2 for signal <matr_xor_0026>.
386 Found 1-bit xor2 for signal <matr_xor_0027>.
387 Found 1-bit xor2 for signal <matr_xor_0028>.
388 Found 1-bit xor2 for signal <matr_xor_0029>.

73
Related source file is "~/home/stig/Documents/ttm4900/meg_final/VHDL/master_rom.vhd"

<table>
<thead>
<tr>
<th>States</th>
<th>Transitions</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Clock</th>
<th>Clock enable</th>
<th>Reset</th>
<th>Reset type</th>
<th>Reset State</th>
<th>Encoding</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>2</td>
<td>14</td>
<td>0</td>
<td>en_in (positive)</td>
<td>reset (positive)</td>
<td>synchronous</td>
<td>idle</td>
<td>automatic</td>
<td>LUT</td>
</tr>
</tbody>
</table>

Synthesizing Unit <master_rom>

Summary:
43 inserted 1 Comparator(s).
51 inserted 8 D-type flip-flop(s).
56 inserted 160 Multiplexer(s).

Unit <private_matrix.a> synthesized.

Synthesizing Unit <sequencer>

Related source file is "~/home/stig/Documents/ttm4900/meg_final/VHDL/sequencer.vhd".

WARNING: Xst:646 - Signal <cnt> is assigned but never used. This unconnected signal will be trimmed during the optimization process.

Summary:
9 inserted 1 Finite State Machine(s).
25 inserted 2405 D-type flip-flop(s).
26 inserted 1 Adder/Subtractor(s).
49 inserted 1 Comparator(s).
52 inserted 160 Multiplexer(s).

Synthesizing Unit <master_rom>

Summary:
1 found 5-to-1 multiplexer for signal <tmp_dbmux0006> created at line 5043.
2 found 5-to-1 multiplexer for signal <tmp_dbmux0007> created at line 5045.
3 found 5-to-1 multiplexer for signal <tmp_dbmux0008> created at line 5047.
4 found 5-to-1 multiplexer for signal <tmp_dbmux0009> created at line 5049.
5 found 5-to-1 multiplexer for signal <tmp_dbmux0010> created at line 5051.
6 found 5-to-1 multiplexer for signal <tmp_dbmux0011> created at line 5053.
7 found 5-to-1 multiplexer for signal <tmp_dbmux0012> created at line 4929.
8 found 5-to-1 multiplexer for signal <tmp_dbmux0013> created at line 5055.
9 found 5-to-1 multiplexer for signal <tmp_dbmux0014> created at line 5057.
10 found 5-to-1 multiplexer for signal <tmp_dbmux0015> created at line 5059.
11 found 5-to-1 multiplexer for signal <tmp_dbmux0016> created at line 5061.
12 found 5-to-1 multiplexer for signal <tmp_dbmux0017> created at line 5063.
13 found 5-to-1 multiplexer for signal <tmp_dbmux0018> created at line 5065.
14 found 5-to-1 multiplexer for signal <tmp_dbmux0019> created at line 5067.
15 found 5-to-1 multiplexer for signal <tmp_dbmux0020> created at line 5069.
16 found 5-to-1 multiplexer for signal <tmp_dbmux0021> created at line 5071.
17 found 5-to-1 multiplexer for signal <tmp_dbmux0022> created at line 5073.
18 found 5-to-1 multiplexer for signal <tmp_dbmux0023> created at line 4931.
19 found 5-to-1 multiplexer for signal <tmp_dbmux0024> created at line 5075.
20 found 5-to-1 multiplexer for signal <tmp_dbmux0025> created at line 5077.
21 found 5-to-1 multiplexer for signal <tmp_dbmux0026> created at line 5079.
22 found 5-to-1 multiplexer for signal <tmp_dbmux0027> created at line 5081.
23 found 5-to-1 multiplexer for signal <tmp_dbmux0028> created at line 5083.
24 found 5-to-1 multiplexer for signal <tmp_dbmux0029> created at line 5085.
25 found 5-to-1 multiplexer for signal <tmp_dbmux0030> created at line 5087.
26 found 5-to-1 multiplexer for signal <tmp_dbmux0031> created at line 5089.
27 found 5-to-1 multiplexer for signal <tmp_dbmux0032> created at line 5091.
28 found 5-to-1 multiplexer for signal <tmp_dbmux0033> created at line 5093.
29 found 5-to-1 multiplexer for signal <tmp_dbmux0034> created at line 5095.
30 found 5-to-1 multiplexer for signal <tmp_dbmux0035> created at line 5097.
31 found 5-to-1 multiplexer for signal <tmp_dbmux0036> created at line 5099.
32 found 5-to-1 multiplexer for signal <tmp_dbmux0037> created at line 5099.
33 found 5-to-1 multiplexer for signal <tmp_dbmux0038> created at line 5101.
34 found 5-to-1 multiplexer for signal <tmp_dbmux0039> created at line 5103.
35 found 5-to-1 multiplexer for signal <tmp_dbmux0040> created at line 5105.
36 found 5-to-1 multiplexer for signal <tmp_dbmux0041> created at line 5107.
37 found 5-to-1 multiplexer for signal <tmp_dbmux0042> created at line 5109.
38 found 5-to-1 multiplexer for signal <tmp_dbmux0043> created at line 5111.
39 found 5-to-1 multiplexer for signal <tmp_dbmux0044> created at line 5113.

Summary:
9 inferred 1 Finite State Machine(s).
25 inferred 2405 D-type flip-flop(s).
26 inferred 1 Adder/Subtractor(s).
49 inferred 1 Comparator(s).
52 inferred 160 Multiplexer(s).

Unit <private_matrix.a> synthesized.

Summary:
1 found 5-to-1 multiplexer for signal <tmp_dbmux0000> created at line 4931.
2 found 5-to-1 multiplexer for signal <tmp_dbmux0001> created at line 4933.
<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
<th>Implementation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>automatic</td>
<td>LUT</td>
<td></td>
</tr>
</tbody>
</table>

Found 18-bit register for signal <to_master>.

Summary:
- inferred 1 Finite State Machine(s).
- inferred 1 ROM(s).
- inferred 514 D-type flip-flop(s).
- inferred 1 Adder/Subtractor(s).
- inferred 1 Comparator(s).
- inferred 5 Tri-state(s).
- Unit <sequencer> synthesized.

HDL Synthesis Report

Macro Statistics

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># ROMs</td>
<td>1</td>
</tr>
<tr>
<td># Multpliers</td>
<td>2</td>
</tr>
<tr>
<td># Adders/Subtractors</td>
<td>4</td>
</tr>
<tr>
<td>3-bit adder</td>
<td>2</td>
</tr>
<tr>
<td>5-bit adder</td>
<td>2</td>
</tr>
<tr>
<td># Counters</td>
<td>2</td>
</tr>
<tr>
<td>5-bit up counter</td>
<td>2</td>
</tr>
<tr>
<td># Registers</td>
<td>3018</td>
</tr>
<tr>
<td>1-bit register</td>
<td>2617</td>
</tr>
<tr>
<td>State</td>
<td>Encoding</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
</tr>
<tr>
<td>idle</td>
<td>0000</td>
</tr>
<tr>
<td>send_to_pmt</td>
<td>0001</td>
</tr>
<tr>
<td>recv_from_pmt</td>
<td>0010</td>
</tr>
<tr>
<td>send_to_dob</td>
<td>0011</td>
</tr>
<tr>
<td>recv_from_dob</td>
<td>0100</td>
</tr>
<tr>
<td>send_to_seq</td>
<td>0101</td>
</tr>
<tr>
<td>recv_from_seq</td>
<td>0110</td>
</tr>
<tr>
<td>send_to_pm</td>
<td>0111</td>
</tr>
<tr>
<td>recv_from_pm</td>
<td>1000</td>
</tr>
<tr>
<td>send</td>
<td>1001</td>
</tr>
</tbody>
</table>

**State/FSM 2** for best encoding.  
Optimizing FSM <state/FSM> on signal <state> with sequential encoding.

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>000</td>
</tr>
<tr>
<td>send_to_mem</td>
<td>001</td>
</tr>
<tr>
<td>recv_from_mem</td>
<td>010</td>
</tr>
<tr>
<td>send_to_reg</td>
<td>011</td>
</tr>
<tr>
<td>recv_mem</td>
<td>110</td>
</tr>
<tr>
<td>send</td>
<td>111</td>
</tr>
<tr>
<td>sync</td>
<td>101</td>
</tr>
<tr>
<td>push</td>
<td>101</td>
</tr>
</tbody>
</table>

**State/FSM 1** for best encoding.  
Optimizing FSM <SRQ/state/FSM> on signal <state> with gray encoding.

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>000</td>
</tr>
<tr>
<td>send_to_mem</td>
<td>001</td>
</tr>
<tr>
<td>recv_from_mem</td>
<td>010</td>
</tr>
<tr>
<td>send_to_reg</td>
<td>011</td>
</tr>
<tr>
<td>recv_mem</td>
<td>110</td>
</tr>
<tr>
<td>send</td>
<td>111</td>
</tr>
<tr>
<td>sync</td>
<td>101</td>
</tr>
<tr>
<td>push</td>
<td>101</td>
</tr>
</tbody>
</table>

**State/FSM 0** for best encoding.  
Optimizing FSM <PM/LS/state/FSM> on signal <state> with gray encoding.

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>000</td>
</tr>
<tr>
<td>send_to_mem</td>
<td>001</td>
</tr>
<tr>
<td>recv_from_mem</td>
<td>010</td>
</tr>
<tr>
<td>send_to_reg</td>
<td>011</td>
</tr>
<tr>
<td>recv_mem</td>
<td>110</td>
</tr>
<tr>
<td>send</td>
<td>111</td>
</tr>
<tr>
<td>sync</td>
<td>101</td>
</tr>
<tr>
<td>push_out</td>
<td>110</td>
</tr>
</tbody>
</table>

Loading device for application Rf_Device from file '5vlx110.t.lpm' in environment /opt/Xilinx/10.1/ISE.

Synthesizing (advanced) Unit <decryption>.

- Found pipelined multiplier on signal <shift_pmt_mult100000> by adding 2 register level(s).
- Multiplier macros will be implemented on LUT. If you want to force its implementation on block, use option 'block' with option 'block' option.  

Unit <decryption> synthesized (advanced).

Synthesizing (advanced) Unit <sequencer>.

INFO Xst: 2385 – HDL ADVISOR – You can improve the performance of the multipliers Multi_shift_pmt_mult100000 by adding 2 register level(s).

Unit <sequencer> synthesized (advanced).
INFO: Xst:2261 - The FF/Latch <count_s> in Unit <decryption> is equivalent to the following FF/Latch, which will be removed: <Mmult_shift_mult1000001>
INFO: Xst:2261 - The FF/Latch <count_s> in Unit <decryption> is equivalent to the following FF/Latch, which will be removed: <Mmult_shift_mult1000000_0>
INFO: Xst:2261 - The FF/Latch <count_s> in Unit <decryption> is equivalent to the following FF/Latch, which will be removed: <Mmult_shift_mult1000000_2>
INFO: Xst:2261 - The FF/Latch <count_s> in Unit <decryption> is equivalent to the following FF/Latch, which will be removed: <Mmult_shift_mult1000000_3>
WARNING: Xst:4242 - Unit sequencer 5 internal tri-states are replaced by logic (pull-up yes): mux31_out<0>, mux31_out<1>, mux31_out<2>, mux31_out<3>, mux31_out<4>.

Optimizing unit <decryption> ...
Optimizing unit <dobbertin_rom> ...
Optimizing unit <private_matrix_a> ...
WARNING: Xst:293 - FF/Latch <count_sor_2> has a constant value of 0 in block <private_matrix_a>. This FF/Latch will be trimmed during the optimization process.
WARNING: Xst:293 - FF/Latch <count_sor_2> has a constant value of 0 in block <private_matrix_a>. This FF/Latch will be trimmed during the optimization process.
Optimizing unit <master_rom> ...
Optimizing unit <sequencer> ...
Mapping all equations...
Building and optimizing final netlist...
FlipFlop dob_vector_in_12 has been replicated 1 time(s)
FlipFlop dob_vector_in_3 has been replicated 1 time(s)
FlipFlop dob_vector_in_4 has been replicated 10 time(s)
FlipFlop dob_vector_in_5 has been replicated 12 time(s)
FlipFlop dob_vector_in_6 has been replicated 9 time(s)
FlipFlop dob_vector_in_7 has been replicated 10 time(s)
FlipFlop dob_vector_in_8 has been replicated 9 time(s)
FlipFlop dob_vector_in_9 has been replicated 6 time(s)
Final Macro Processing ...

Final Register Report

Macro Statistics
# Registers : 5937
Flip-Flops : 5937

* Partition Report *

Partition Implementation Status
Final Report

Final Results

RTL Top Level Output File Name: decryption.ngr
Top Level Output File Name: decryption
Output Format: NOX
Optimization Goal: Speed
Keep Hierarchy: NO

Design Statistics

# IOs: 324

Cell Usage:

- BELS: 10493
- CND: 1
- INV: 3
- LUT1: 4
- LUT2: 1848
- LUT3: 561
- LUT4: 412
- LUT5: 1443
- LUT6: 5739
- MUXCY: 10
- MUXF7: 518
- MUXF8: 3
- VCC: 1
- XORCY: 10
- XORF: 5937
- FDR: 161
- FDE: 1
- FDRE: 5775
- Clock Buffers: 2
- BUFG: 1
- BUFGP: 1
- IO Buffers: 323
- BUF: 162
- OBUF: 161

Device utilization summary:

Selected Device: 5v1x110f1136-1

Slice Logic Utilization:

- Number of Slice Registers: 5937 out of 69120 (8%)
- Number of Slice LUTs: 9950 out of 69120 (14%)
- Number used as Logic: 9950 out of 69120 (14%)

Slice Logic Distribution:

- Number of LUT Flip Flop pairs used: 11463
- Number with an unused Flip Flop: 5526 out of 11463 (48%)
- Number with an unused LUT: 1513 out of 11463 (13%)
- Number of fully used LUT-FF pairs: 4424 out of 11463 (38%)
- Number of unique control sets: 64

IO Utilization:

- Number of IOs: 324
- Number of bonded IOBs: 324 out of 640 (50%)

Specific Feature Utilization:

- Number of BUFG/BUFGCTRLs: 2 out of 32 (6%)

Partition Resource Summary:

No Partitions were found in this design.

TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
GENERATED AFTER PLACE-and-ROUTE.
### Clock Information

<table>
<thead>
<tr>
<th>Clock Signal</th>
<th>Clock buffer (FF name)</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>BUFGP</td>
<td>5937</td>
</tr>
</tbody>
</table>

### Asynchronous Control Signals Information:

- No asynchronous control signals found in this design

### Timing Summary:

- Speed Grade: -1

  - Minimum period: 4.974 ns (Maximum Frequency: 201.045 MHz)
  - Minimum input arrival time before clock: 4.204 ns
  - Maximum output required time after clock: 3.259 ns
  - Maximum combinational path delay: No path found

### Timing Detail:

- All values displayed in nanoseconds (ns)

### Timing constraint: Default period analysis for Clock 'clk'

- Clock period: 4.974 ns (frequency: 201.045 MHz)
- Total number of paths / destination ports: 102627 / 11860

### Delay:

- 4.974 ns (Levels of Logic = 6)

### Source:

- dob_vector_in_8_5 (FF)

### Destination:

- DR/db_2 (FF)

### Source Clock:

- clk rising

### Destination Clock:

- clk rising

### Data Path: dob_vector_in_8_5 to DR/db_2

<table>
<thead>
<tr>
<th>Cell in-&gt;out</th>
<th>Gate</th>
<th>Net</th>
<th>Logical Name (Net Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FERE-C-&gt;Q</td>
<td>12</td>
<td>0.471</td>
<td>dob_vector_in_8_5 (dob_vector_in_8_5)</td>
</tr>
<tr>
<td>LUT5:10-&gt;O</td>
<td>2</td>
<td>0.094</td>
<td>DR/db_2_or0000891 (DR/db_2_or0000891)</td>
</tr>
<tr>
<td>LUT5:13-&gt;O</td>
<td>1</td>
<td>0.094</td>
<td>DR/db_2_or0000113148 (DR/db_2_or0000113148)</td>
</tr>
<tr>
<td>LUT5:11-&gt;O</td>
<td>1</td>
<td>0.094</td>
<td>DR/db_2_or0000113328 (DR/db_2_or0000113328)</td>
</tr>
<tr>
<td>LUT6:15-&gt;O</td>
<td>1</td>
<td>0.094</td>
<td>DR/db_2_or0000113389 (DR/db_2_or0000113389)</td>
</tr>
<tr>
<td>LUT6:15-&gt;O</td>
<td>1</td>
<td>0.094</td>
<td>DR/db_2_or0000113402 (DR/db_2_or0000113402)</td>
</tr>
<tr>
<td>LUT6:14-&gt;O</td>
<td>1</td>
<td>0.094</td>
<td>DR/db_2_or0000126880 (DR/db_2_or0000126880)</td>
</tr>
<tr>
<td>FERE-D</td>
<td>-0.018</td>
<td></td>
<td>DR/db_2</td>
</tr>
</tbody>
</table>

### Total

- 4.974 ns (1.035 ns logic, 3.939 ns route) (20.8% logic, 79.2% route)

### Timing constraint: Default OFFSET IN BEFORE for Clock 'clk'

- Total number of paths / destination ports: 1062 / 907

### Offset:

- 4.204 ns (Levels of Logic = 5)

### Source:

- inputs<100> (PAD)

### Destination:

- shift_pmt_0 (FF)

### Destination Clock:

- clk rising

### Data Path: inputs<100> to shift_pmt_0

<table>
<thead>
<tr>
<th>Cell in-&gt;out</th>
<th>Gate</th>
<th>Net</th>
<th>Logical Name (Net Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFG:10-&gt;O</td>
<td>1</td>
<td>0.818</td>
<td>inputs&lt;100&gt;BUFG (inputs&lt;100&gt;BUFG)</td>
</tr>
<tr>
<td>LUT2:10-&gt;O</td>
<td>1</td>
<td>0.094</td>
<td>shift_pmt_mux0001&lt;0&gt;1414 (shift_pmt_mux0001&lt;0&gt;1414)</td>
</tr>
<tr>
<td>LUT6:12-&gt;O</td>
<td>1</td>
<td>0.094</td>
<td>shift_pmt_mux0001&lt;0&gt;1620 (shift_pmt_mux0001&lt;0&gt;1620)</td>
</tr>
<tr>
<td>LUT6:14-&gt;O</td>
<td>1</td>
<td>0.094</td>
<td>shift_pmt_mux0001&lt;0&gt;11355 (shift_pmt_mux0001&lt;0&gt;11355)</td>
</tr>
<tr>
<td>FERE-D</td>
<td>-0.018</td>
<td></td>
<td>shift_pmt_0</td>
</tr>
</tbody>
</table>

### Total

- 4.204 ns (1.194 ns logic, 3.010 ns route) (28.4% logic, 71.6% route)

### Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'

- Total number of paths / destination ports: 160 / 160

### Offset:

- 3.259 ns (Levels of Logic = 1)

### Source:

- outputs<159> (FF)

### Destination:

- outputs<159> (PAD)

### Source Clock:

- clk rising

---

84
**Data Path:** outputs<159> to outputs<159>

<table>
<thead>
<tr>
<th>Cell</th>
<th>in→out</th>
<th>fanout</th>
<th>Delay</th>
<th>Delay</th>
<th>Logical Name (Net Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDRE</td>
<td>C→Q</td>
<td>1</td>
<td>0.471</td>
<td>0.336</td>
<td>outputs&lt;159&gt; (outputs&lt;159&gt;)</td>
</tr>
<tr>
<td>OBUF</td>
<td>I→O</td>
<td>2.452</td>
<td></td>
<td></td>
<td>outputs&lt;159&gt;_OBUF (outputs&lt;159&gt;)</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>3.259ns (2.923ns logic, 0.336ns route)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(89.7% logic, 10.3% route)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total REAL time to Xst completion: 19145.00 secs
Total CPU time to Xst completion: 19061.31 secs

Total memory usage is 1005084 kilobytes

Number of errors: 0 (0 filtered)
Number of warnings: 4 (0 filtered)
Number of infos: 14 (0 filtered)