High Speed FPGA Architectures for the Data Encryption Standard

by
Jens-Peter Kaps

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical Engineering

May, 1998

Approved:

Prof. Christof Paar
ECE Department
Thesis Advisor

Prof. Wayne P. Burleson
ECE Department,
University of Mass., Amherst
Thesis Committee

Prof. Yusuf Leblebici
ECE Department
Thesis Committee

Prof. John Orr
ECE Department Head

Abstract

Most modern security standards and security applications are defined to be *algorithm independent*, that is, they allow a choice from a set of cryptographic algorithms for the same function. Since the Data Encryption Standard (DES) is currently the most widely used private-key encryption algorithm, DES is usually amongst them. Field Programmable Gate Arrays (FPGA) are reconfigurable hardware devices. They can switch algorithms on-the-fly. Thus, cryptographic algorithms which are implemented on FPGAs provide an an ideal match for algorithm independent security applications. On FPGAs, cryptographic algorithms can run much faster than on software while preserving the security of traditional hardware solutions. At the same time, FPGAs allow potentially the same flexibility as software does. Although there have been a few previous reports on DES implementations on reconfigurable devices, there has been no systematic treatment of that matter.

We designed and implemented various architecture options with strong emphasis on high-speed performance. Techniques like *pipelining* and *loop unrolling* were used and their effectiveness for DES on FPGAs investigated. We also performed optimization on a lower level. The most interesting result is that we could achieve data rates of up to 384 Mbit/s using a standard Xilinx FPGA (speed-grade -3). This result is by factor 30 faster than software implementations while we are still maintaining flexibility.

Preface

I would like to thank the many people who contributed to this work. First, my advisor *Christof Paar* for his advice and support throughout this entire project. He never lost faith in my abilities and encouraged me to finish this work just in time. Not to mention his contribution to the coffee maker in the lab which was badly needed. Next I would like to thank Gregory Haskins who gave me a crash course on Workview Office and a thorough introduction to this project. Martin Rosner worked on a different project but using the same software tools. Together we overcame many hurdles with the tools and finally managed to place and route and also simulate our designs. Many long and sleepless nights in the lab also helped us to become good friends. Furthermore I want to thank the *Algorithm Agile* MQP-group (Frank Wong, Pik-Ying Kwok and M. James Allred) for making me adjusting my design so it runs in a real world application and not just in theory. This was a valuable experience. Special thanks go to the system administrators here at the ECE Department Murtaza Amiji and Brady Schulman. They managed to keep the systems alive in spite of our continuous attacks. I also want to thank the National Science Foundation for partially funding this research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

We are in the midst of a shift toward an information society. In a recent study [10] *Dataquest* reports that at the end of 1997 82 million computers were connected to the Internet. They projected the number of computers connected to the Internet for the year 2001 to be 268 million. With this immense growth the Internet also becomes more and more attractive as a market place. Other areas of communications are growing too, e.g., the wireless communication market, electronic payment systems (*home banking*), to name just a few. At the same time security aspects of information and communication systems are of growing concern. Tapped mobile phone conversations, stolen credit card numbers, faked bank transactions are just a few examples of threats imposed by an unprotected communication infrastructure. The central tool for achieving the desired security is cryptography.

Already in 1972, the National Bureau of Standards, now the National Institute of Standards and Technology (NIST), was aware of the potential thread to computer and communications data. They initiated a program to develop a standardized encryption algorithm. In 1976 the Data Encryption Standard (DES) was released. Since then DES was approved by the American National Standards Institute (ANSI X3.92) and renamed Data Encryption Algorithm (DEA), by the International Standards Organization (ISO) and many bank standards. DES is being reviewed every five years for renewed approval. The next review is scheduled for this year and it is expected that DES will not be reapproved for another five years. DES is currently the most widely used private-key algorithm and it is also part of many other standards e.g., for ATM cell encryption, the Secure Socket Layer protocol, and for various ANSI banking standards. Even if DES is not being reapproved, it is still important and will continue to play a major role for several more years.

Most new security standards and security applications are defined to be *algorithm independent*. That is, for a given security service such as privacy, a number of different algorithms can be used alternatively. This situation applies to public-key based services as well as to private-key services. It is fairly easy to switch crypto algorithms in software, but it is difficult in hardware. On the other hand, hardware solutions provide a better speed and higher physical security. One answer to this problem is reconfigurable hardware, based on modern field programmable gate array, or FPGA, devices. FPGAs can switch algorithms, they can thus be used to build *algorithm agile* applications. This means that the same device can be used for different algorithms, the nature of the algorithms does not matter. In cryptographic applications, an FPGA can be used for the realization of several different encryption algorithms. Although at a given time only one algorithm is configured, the FPGA can be reconfigured with a different algorithm on-the-fly if needed. Moreover the same FPGA can therefore be used for public-key and private-key algorithms. In summary, cryptographic algorithms on FPGAs bear a number of advantages such as:

- **Algorithm agility**, the same FPGA can be reprogrammed on the fly to support different algorithms,
- **Scalable security**, through different versions on the same algorithm (e.g., DES

and triple-DES),

- **Alterable architecture parameters**, e.g., desirable features such as variable S-boxes, variable number of rounds, or different modes of operation can easily be realized,

- **Resource efficient** the same resource can be used for private and public-key algorithms.

Although there have been a few previous reports on DES implementations on reconfigurable devices, there has been no systematic treatment of the matter. In this thesis, several architectural options for DES implementation on FPGAs are investigated and implemented with a strong emphasis on high-speed architectures.

1.2 Thesis Outline

Chapter 3 describes the design and implementation cycle. Furthermore it gives an overview of the hardware and software tools we used for our research. In addition it includes some remarks on the performance and effectiveness of the tools.

Chapter 4 provides an introduction to the Data Encryption Standard. It also concerns the modes of operation and enhancements to DES.

Chapter 5 explores different architecture options for DES like loop unrolling and pipelining. At the end it provides an overview of the architecture versions we decided to implement.

Chapter 6 is concerned with the design of the circuit. DES is broken down into small elementary computational units and some optimizations are performed.

Chapter 7 describes the implemented architectures in detail. It explains our choice of device and gives an overview of the source code. The signals of the control logic for each architecture are discussed in detail.

Chapter 8 presents the results of our implementations of the different architectures. We compare the achievements of pipelining and loop unrolling and discuss the influence of chip parameters.

Chapter 9 concludes this work with a short summary of the results and some recommendations for further research.

Chapter 2

Previous Work

This chapter summarizes previous work on hardware implementations of DES. It distinguishes between ASIC and FPGA implementations and also mentions future technologies which might become important for DES implementations.

2.1 Early Work

Early references for custom hardware implementations are [6] and [11]; both papers were presented at *CRYPTO 84*. [6] describes an DES implementation which supports all four modes of operation. The maximum speed of this chip is said to be 20 Mbit/sec.

The paper [11] concerns an *LSI digital encryption processor*. It enables a user to program any mode of operation. The maximum speed is given as 4.72 Mbit/sec.

In 1988 [8] was published. It describes a CMOS chip in 3-μm double-metal technology which can achieve a data rate of 32 Mbit/sec. This is 60% faster than the implementation shown in [6]. It also supports all modes of operation.

Earlier reference [3] is the first paper which is mainly concerned with increasing the performance of DES by restructuring the algorithm. This paper mentions the *one-round sub-key precomputation* as a speed-up technique. Another interesting idea that is presented in this paper is *XOR rearrangement* which takes one XOR-delay out of the critical path. We did not employ this approach in our design, as modern FPGA synthesizing tools optimize the low level logic themselves. The data rate of the implementation was not mentioned.

2.2 Current Implementations

Modern custom hardware implementations can achieve data rates of 1 Gbit/sec and beyond. Reference [2] was the first report of a custom chip, employing modern Gallium Arsenide technology to achieve 1 Gbit/sec. In a later publication of the same research group [5] they describe this design in more detail. They also mention that the fasted chip they tested could run at 1.4 Gbit/sec. One major disadvantage of this design is, that only a 7 bit wide port is available for loading the master key. That means that frequent key changes slow this chip down significantly.

The first paper to show an implementation of DES on FPGAs is [9]. Their approach generates key-specific circuitry for the Xilinx FPGAs. One drawback of this approach is that a binary image (bit-stream) for each key has to be precomputed before it can be used in the device. We experienced run times of the synthesis and place and route tools from 4 hours to longer than weeks on high power workstations. This is a task that can not be accomplished on the fly. Hence, prestored binary images limits the number of keys that can be used drastically. Furthermore even their fastest implementation without decryption and adjusted to one key, is in the same device (although a slower speed grade) by factor three slower and requires almost twice as much logic resources as the design we present in this paper *DES_ED16*.

A very interesting technology, especially for algorithm agile implementations is presented in [4]. The new technology *Dynamically Programmable Gate Arrays (DPGAs)* support a single cycle, array wide context switch. That means that it take only one clock cycle for the device to switch to an entirely different algorithm. With current FPGAs this takes 10's of milliseconds due to limited bandwidth to off-chip memories [14]. Although [4] does not target cryptographic applications in particular, DPGAs seem highly attractive for these purposes.

Chapter 3

Methodology

This chapter describes the design procedure we applied for our research. It also describes our choice of tools in hardware and software as well as it includes some remarks on the performance and effectiveness of the tools.

3.1 The Design Cycle

The general design cycle for this work consisted of the following steps:

1. Research of DES algorithm

2. Researching architecture options

3. Optimizing the DES architecture

4. VHDL implementation of basic DES function blocks

5. Creating multiple versions of the DES design employing different architecture options

6. Verifying each version on the register-transfer-level (RTL)

7. Synthesis and logic optimization

8. Place and Route for a specific device

9. Back-annotated verification of the design

The steps outlined above were performed more or less in this order. Steps 1 trough 4 were performed first and sometimes even concurrently; e.g., during the VHDL implementation of the basic function blocks some more ideas for optimization developed.

Steps 5 to 9 were performed in this order for each design separately. The next design was started usually while the current design was in the *Place and Route* stage, because this particular stage took the longest time. In case a verification step did not give the desired results, we had to go back some steps, usually till step 5 or even 4, to fix that problem and start the design process again from there.

Early in the design we decided upon a FPGA vendor and a device family as described in Subsection 7.1. That decision was based majorly on previous work in this area done by Haskins (see [7]). Availability of the actual Chip and the tools were another important reason. This enabled us to use vendor specific macros (LogiBLOX, see Chapter 6.3.1).

3.2 Tools

The entire design, with the exception of the LogiBLOX, was implemented using VHDL. Each design was tested at the register-transfer-level (RTL), i.e., right from the VHDL files and LogiBLOX VHDL simulation models. This way we could find logical errors and major timing problems early in the design phase. For the rtl-level simulation Synopsys VHDL analyzer (`vhdlan`) version 1997.08 was used.

The next step is to synthesis the design and create an optimized netlist describing the gate level design in Xilinx format. Synopsys `fpga_analyzer` version 1997.08 accomplished this task.

The netlist is used by Xilinx to place and route the design for a specific device. The result is a bit-stream to program the chip, a simulation model as well as exact timing results. The Xilinx design-manager `dsgnmgr` version M1.3.7 was employed for this.

The final step is to verify the design once again, this time with the simulation model generated by the Xilinx tools. This simulation model contains the actual physical net, CLB, and pad delays introduced from the device. The Synopsys VHDL analyzer (`vhdlan`) was used once again to verify this back-annotated design.

3.2.1 Xilinx Synopsys Interface

The *Xilinx-Synopsys-Interface (XSI)* design tool kit allows to implement Xilinx Field Programmable Gate Arrays (FPGA) designs using the Synopsys FPGA Compiler. It includes all libraries necessary for Synopsys `fpga_analyzer` to optimize the design for the FPGA and for Synopsys `vhdlan` to read the back-annotated designs from Xilinx for past place and route verification. Figure 3.1 presents a flow chart diagram of the design flow with the XSI tools.

3.2.2 Simulation and Verification

As stated before, the design is verified twice during the design process. First the RTL-level simulation of the VHLD source code and the behavioral models of the LogiBLOX and second after place and route.

For both simulations the same *test bench* can be used. The test bench is a VHDL file which contains test vectors and the order and timing of how they are going to be applied to the design. A sample test bench can be found in Appendix E and the

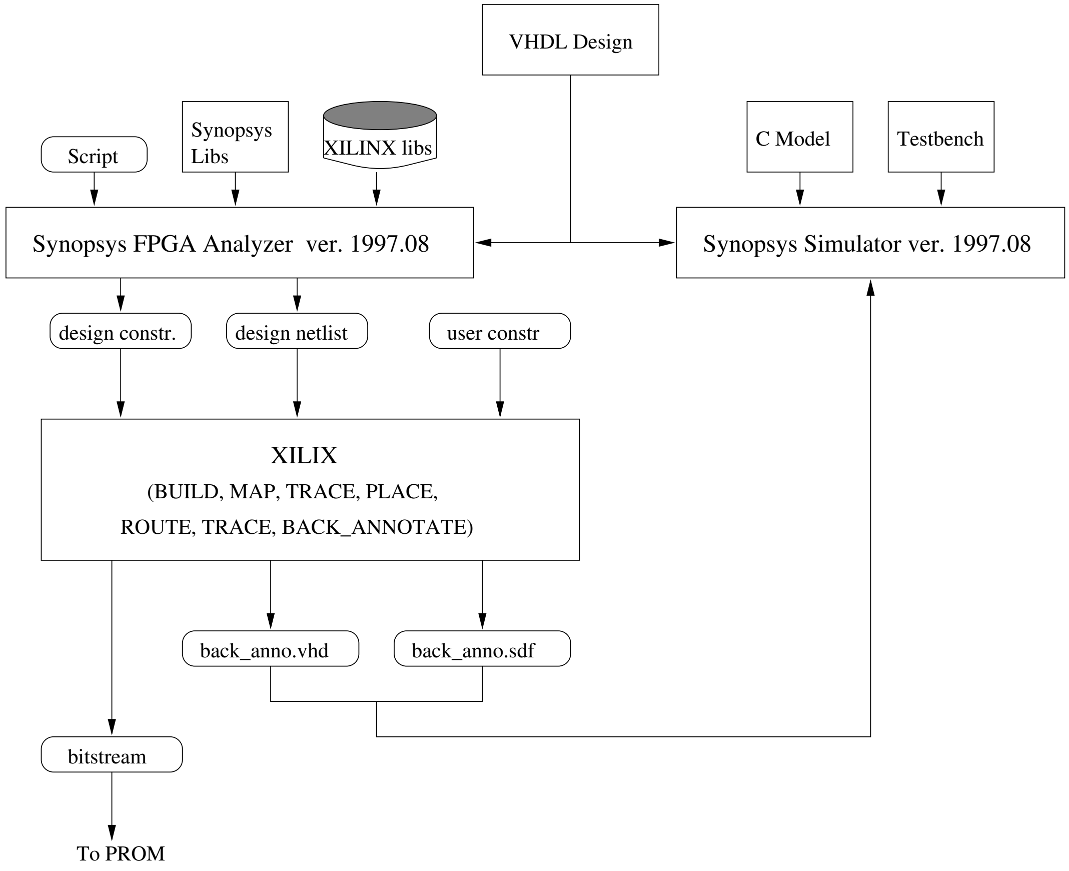


Figure 3.1: XSI design flow

result of a past place and route timing simulation in Appendix D.

The test vectors for the design were generated by a DES design written in C. This program also provides results from within the DES design, so that smaller entities could be tested and errors could be tracked down easily to single VHDL files.

The Synopsys simulator can work in two different modes: *compiled mode* and *interpreted mode*. In order to run in compiled mode a C-compiler is necessary. On the HP-Workstation on which Sysnopsys is installed, a C-compiler was not available to us. Therefore we had to run the simulator in interpreted mode. That required editing of all the library files used by the design: `mvlutil.vhd`, `mvlarith.vhd`, `logiblox.vhd`, `simprim_Vcomponents.vhd`, `simprim_Vpackage.vhd`, and `simprim_VITAL.vhd` and of course `time_sim.vhd`, which is the result of the past place and route timing simulation.

Sample script files to invoke the simulation are presented in Appendix A.1 and Appendix A.2.

3.2.3 Synthesis

In the middle of our research we switched synthesis tools from *Workview Office* to *Synopsys*. That also included a shift from *Windows* to *UNIX*. It was found that the Synopsys tools are much more powerful than the Workview Office environment, but also much more difficult to learn. The documentation accompanying Synopsys is quite extensive and very helpful.

One interesting result of that switch is that the design *DES16 v1.1* (see 6.3) synthesized with Workview Office could run at a maximum speed of 62 Mbit/sec, whereas adjusted to Synopsys and synthesized the same design could run at a maximum speed of 88 Mbit/sec.

Another major advantage of Synopsys is the ability to run *script files*. All necessary steps to synthesize and optimize a design, prepare summaries and specifying the setup parameters, can be included in a script file. A sample script file is provided in

Appendix B.

3.2.4 Place and Route

The Xilinx place and route tools were used on the HP Workstations as well as on Windows computers. The Windows computers were Pentium based PCs running at 200 Mhz, whereas the HPs are running at 60 MHz and at 75 Mhz. Therefore the Xilinx tools were much faster on the PCs, but still the pace and route process took in some cases more than a week. The results achieved using the Xilinx tools on the PCs were comparable with the results achieved using the Xilinx tools on the HPs.

The input to the place and route tools is a design netlist and constraints file generated by Synopsys, as well as user constraints, specifying the maximum clock period desired and pin assignments. The output of this process is a bit-stream file that can be used to program the FPGAs and the back-annotated design.

Furthermore the Xilinx tools perform a timing analysis after place and route which shows the **minimum clock period** for the given design. This clock period is guaranteed by Xilinx for the design and therefore is to be seen as rather pessimistic. We are using this timing result for our speed calculations.

Chapter 4

DES Algorithm

The Data Encryption Standard was published by the National Bureau of Standards in 1975. DES is a so-called *Block Cipher*, i.e., it encrypts or decrypts a whole block of data bits at once as opposed to stream ciphers which encrypt or decrypt a bit-stream bit by bit. Figure 4.1 shows a basic I/O diagram of DES.

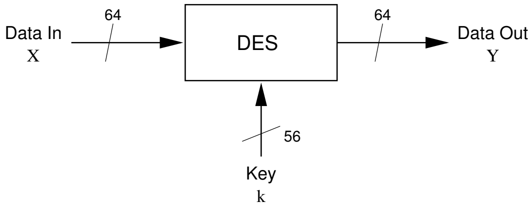
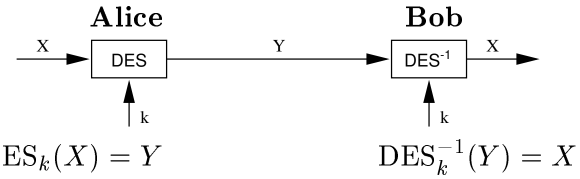


Figure 4.1: Overview of DES

DES encrypts blocks of 64 bits length (plaintext) with a 56 bits long key. The result is a ciphertext of equal length to the plaintext. During the explanation of DES in this chapter we will concentrate on the encryption function The decryption, which is almost identical to the encryption function. function will be discussed in Section 4.3. Our description will highlight the internal functions of DES which are important for a hardware implementation.

Here is a small example of how DES works. Alice and Bob are sharing the same key k. Alice encrypts the plaintext x and sends the encrypted version y over the network to Bob. Bob uses the same key and the inverse of the DES function to recover the plaintext x.



4.1 The DES Core Function

Figure 4.2 shows an overview of the whole DES-Algorithm. The plaintext input of DES x gets permuted by the *initial permutation* IP resulting in x_0. For the next step x_0 is split up into the higher (first) 32 bits L_0 and the lower (last) 32 bits R_0 (little endian): $\mathrm{IP}(x) = L_0 R_0$.

This is the input for the main DES function, the so called *Feistel Network*. It contains an iterative structure; a certain function is executed 16 times where the input of the next round is the output of the previous round. Figure 4.3 shows one round of DES.

The index i indicates for the current iteration and can therefore take the values $1 \leq i \leq 16$. The result of one round of the DES algorithm can be described as:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

where $\oplus$ denotes the exclusive-or of two bit-strings. The f-function of each round is dependent on R_{i-1} and the *sub-key* K_i of the 56-bit key. After the 16th round R_{16} and L_{16} get swapped resulting in $R_{16}L_{16}$ and the final permutation IP^{-1} which

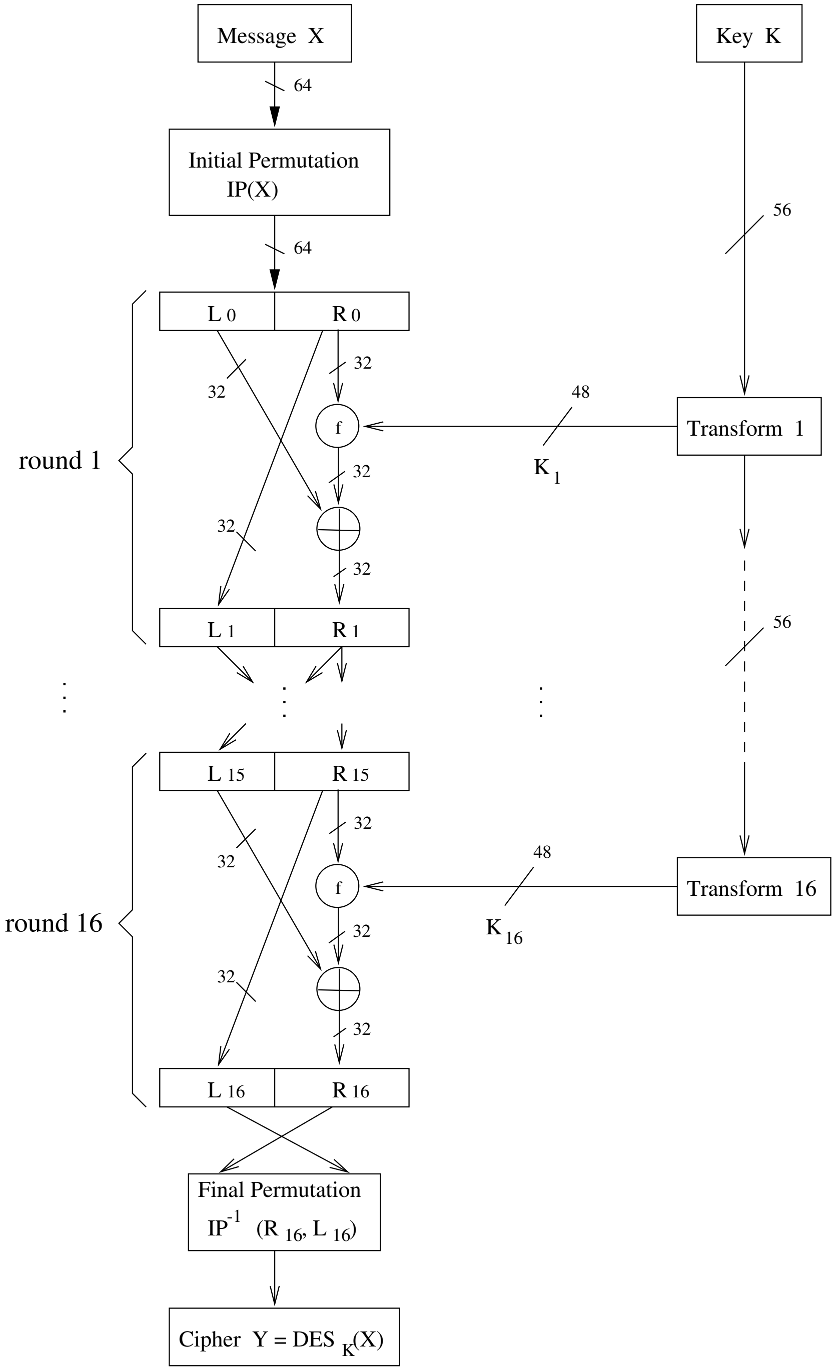


Figure 4.2: Functional block diagram of DES

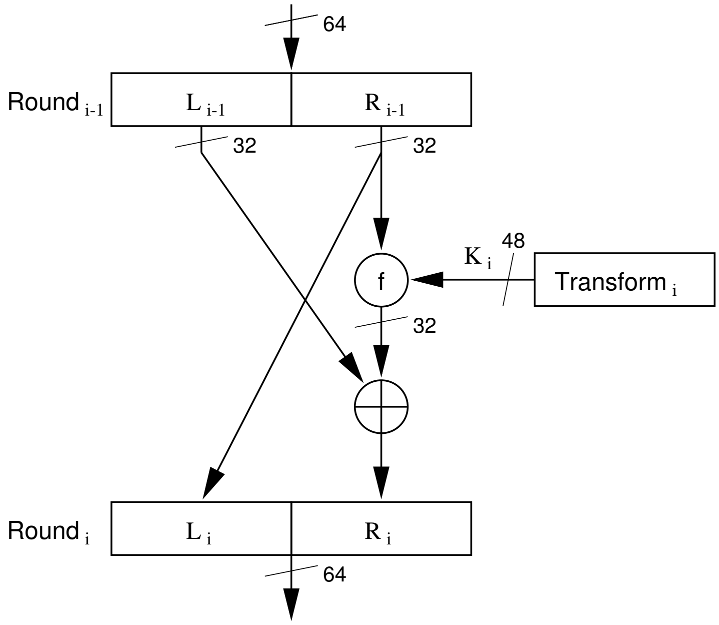


Figure 4.3: DES Feistel network

is inverse to the initial permutation is applied. This generates the final ciphertext y =IP$^{-1}(R_{16}L_{16})$.

The f-function (see Figure 4.4) takes the 32 bits of R_{i-1} as input and expands it to 48 bits; 16 bits of R_{i-1} are appearing twice at the output E(R_{i-1}). The 48 bits are combined via an exclusive-or with the 48 bits sub-key K_i from the key transformation: E$(R_{i-1}) \oplus K_i$. This result is split into 8 blocks of 6 bits each which form the input of the *S-Boxes*. The S-Boxes are basically look-up tables which assign each 6-bit input value a 4-bit value. The eight 4-bit values get combined to 32 bits and a permutation P is applied. The resulting bit-string is $f(R_{i-1}, K_i)$.

4.2 DES Key Scheduling

Each round of DES requires a distinct sub-key K_i. These sub-keys are generated from the key K. The key K is 64 bits long and contains eight parity check bits, so

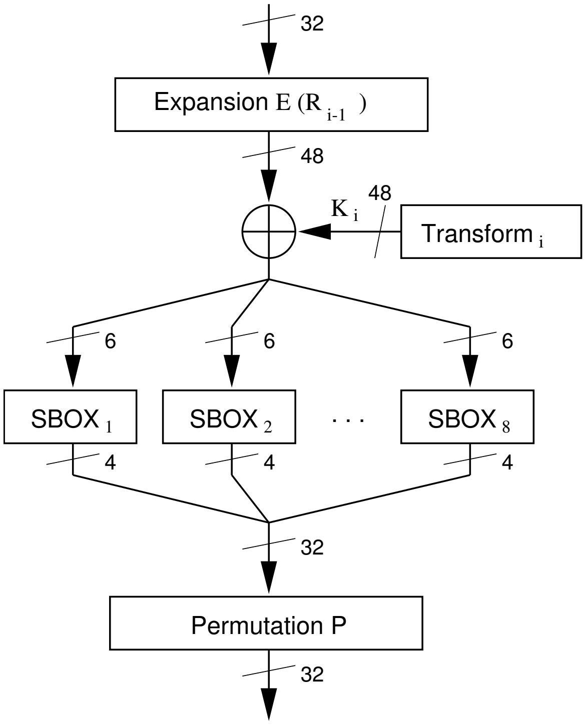


Figure 4.4: DES f-function

the effective key is 56 bits long. The 56 bit key is also the input the design described here expects.

The sub-key generation is also an iterative process comprising 16 rounds. The 56-bit key gets permuted by the permutation PC-1 and then split up into two halfs, each 26 bits long: PC-1$(K) = C_0D_0$, where C_0 denotes the higher (first) 32 bits and D_0 the lower (last) 32 bits (little endian).

For each round of the Feistel network a new sub-key is being generated. Figure 4.5 shows one iteration of the DES key schedule. With each iteration C_{i-1} and D_{i-1} are rotated left (cyclic shift left) denoted as LS_i. Depending on i, C_{i-1} and D_{i-1} are shifted one position (for $i =$ 1,2,9,16) or two positions (otherwise).

$$C_i = LS_i(C_{i-1})$$

$$D_i = LS_i(D_{i-1})$$

The result C_i and D_i are passed as input to the next round and are also permuted with the permutation PC-2 to form the sub-key: K_i =PC-2($CiDi$). This permutation reduces the number of bits from 56 to 48.

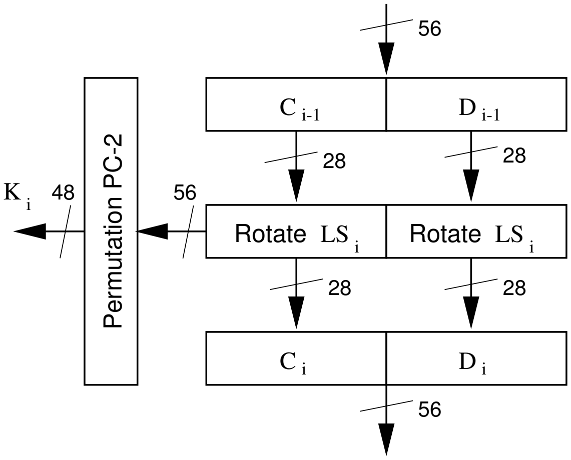


Figure 4.5: DES key schedule

4.3 Decryption

DES decryption uses the same algorithm as encryption. The only difference is that the sub-keys have to be generated in a reverse order $K_{16}, \ldots, K_1$. The result will be the plaintext x. In order to create the sub-keys in the reverse order, C_{i-1} and D_{i-1} have to be cyclicly shifted right, as opposed to left for encryption, depending on i. The following Table 4.1 shows how many positions C_{i-1} and D_{i-1} have to be shifted.

Iteration	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Encryption	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1
Decryption	0	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Table 4.1: DES sub-key shift schedule

4.4 DES Modes of Operation

Four modes of operation have been standardized for DES (see [13] page 83): *electronic codebook mode* (ECB), *cipher block chaining mode* (CBC), *cipher feedback mode* (CFB), and *output feedback mode* (OFB).

Electronic Codebook Mode (ECB) is the simplest approach for using a block cipher. The plaintext is divided into 64 bit long blocks X_i and each block is encrypted separately (see Figure 4.6). Identical plaintext blocks result in identical ciphertext blocks: $Y_i = e_k(X_i)$. The major problem with this simple mode of operation is that ciphertext substitution attacks can be performed.

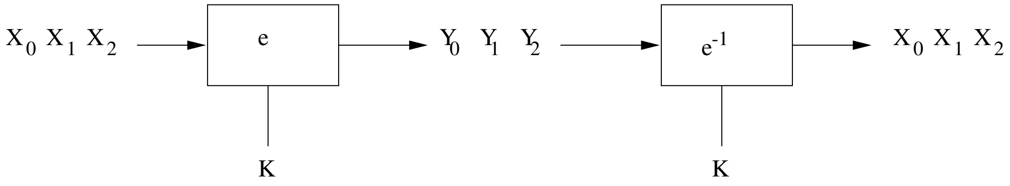


Figure 4.6: Electronic Codebook Mode

Cipher Block Chaining Mode (CBC) employs an initialization vector IV and a feedback loop. Each block of ciphertext depends on all previous blocks of ciphertext (see Figure 4.7). The first block of the plaintext is XORed with the initialization vector before it is encrypted. All consecutive blocks is XORed with the encrypted previous block before they are encrypted: $Y_0 = e_k(X_0 \oplus IV)$ and $Y_i = e_k(X_i \oplus Y_{i-1})$ for $i \geq 1$.

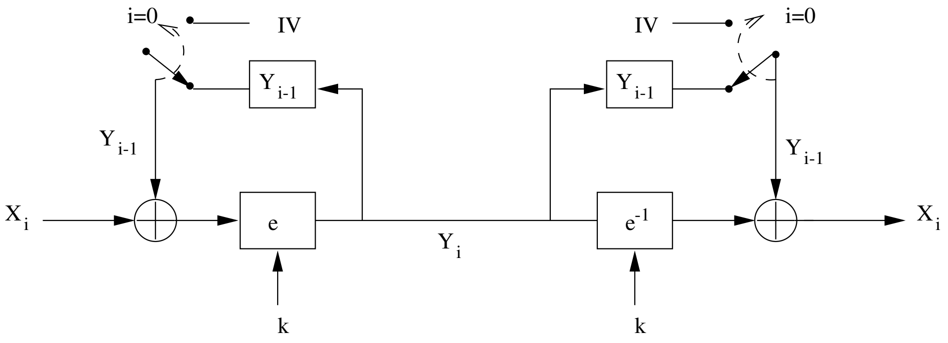


Figure 4.7: Cipher Block Chaining Mode

Cipher Feedback Mode (CFB) is often employed to encrypt messages smaller than 64 bits; it does not require padding. Figure 4.8 shows a schematic of the CFB. A shift-register is preloaded with an initialization vector IV in stage $i = 0$. The parallel output of this 64 bits wide shift register is encrypted: $\tilde{z}_0 = e_k(IV)$.

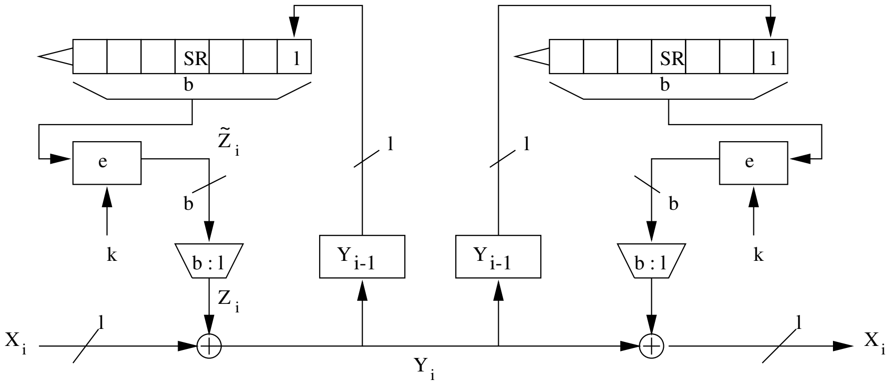


Figure 4.8: Cipher Feedback Mode

The leftmost l bits are taken $\tilde{z}_i \rightarrow z_i$ and XORed with the l bits long plaintext to generate the ciphertext: $Y_i = X_i \oplus z_i$. The shift register is then shifted by l bits and Y_i loaded into the rightmost position. Encryption of the new shift register contents creates the new $\tilde{z}_{i+1}$.

Output Feedback Mode is similar to CFB except that output of the encryption function is used as feedback and not the ciphertext.

4.5 DES Enhancements

DES can be made more secure if it is used three times (*triple encryption*) in a row.

Two different type of triple encryption are very common: **encrypt-decrypt-encrypt** and **encrypt-encrypt-encrypt**.

For the encrypt-decrypt-encrypt type usually only two keys are used. The plaintext X gets encrypted with the first key $e_{k1}(X)$, decrypted with the second key $e_{k2}^{-1}(e_{k1}(X))$ and then encrypted again with the first key: $Y = e_{k1}(e_{k2}^{-1}(e_{k1}(X)))$.

Figure 4.9 shows the encrypt-encrypt-encrypt type. The plaintext X gets encrypted three times in a row with a different key for each: $Y = e_{k3}(e_{k2}(e_{k1}(X)))$.

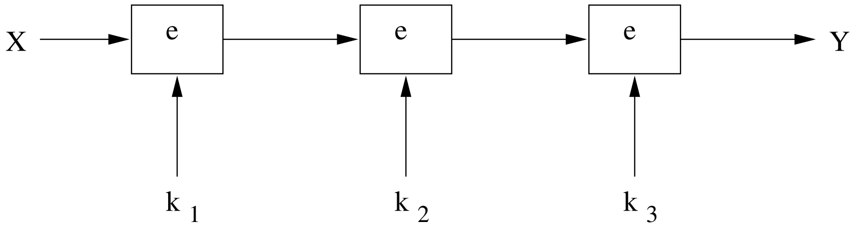


Figure 4.9: Triple encryption

Please note that double encryption does not result in a significantly larger key space than single encryption due to the *meet-in-the-middle* attack. Due to this attack, the key space of triple encryption is roughly $2^{2\times 56} = 2^{112}$ [1].

Chapter 5

Architecture

The first step for an effective implementation of DES is to structure the algorithm and evaluate the resulting architecture options.

5.1 Structuring DES

As described in Section 4 the DES algorithm contains an iterative structure. Data is passed through the Feistel Network, as shown in Figure 4.3, 16 times, each time with a different sub-key from the key transformation. Figure 5.1 shows this using a flow-chart. The plaintext is the input and the iteration counter i is set to 1. The Feistel Network is shown as a box labeled $Round_i$. After each round, i is tested if it is smaller than 16 and if so, i is incremented by one, the current output is fed-back into the Feistel Network, and the next iteration starts. After 16 rounds the calculation of the ciphertext y is done.

From the flowchart we can derive the block diagram of DES which is closer to the hardware implementation and therefore enables us to investigate further enhancements. The block diagram shown in Figure 5.2 comprises the same design as the flowchart.

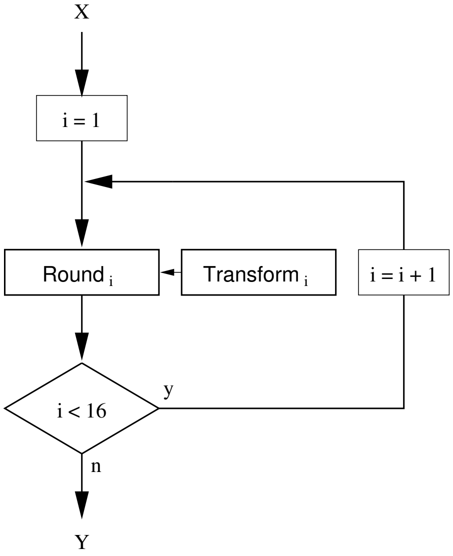


Figure 5.1: Flowchart of DES

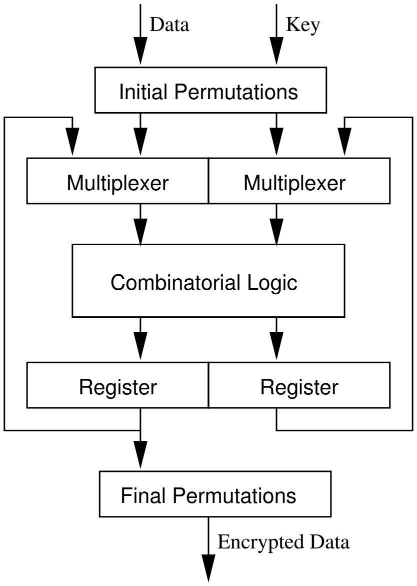


Figure 5.2: DES block diagram

As we have seen in Section 4, the incoming data and key are passed through initial permutations. Then the data passes 16 times through the Feistel Network and also

16 sub-keys are generated simultaneously. Both, the Feistel Network operation and the sub-key generation is denoted in the block diagram as *Combinatorial Logic* (CLU, combinatorial logic unit). In order to be able to loop the output back to the input of the combinatorial logic unit we need *Registers* and *Multiplexers*. The multiplexer switches the inputs of the combinatorial logic unit between data from the previous round and new input data and key. The registers store the results of each loop and pass them on to the multiplexer. The output of the data register passes through the *Final Permutation*. For simplicity the result of each loop passes through the final permutation and then to the output. It is the responsibility of a control logic to signal an external entity if the output is valid or not.

5.2 Loop Unrolling

In this section we will discuss the first general technique for accelerating a DES hardware implementation. Loop Unrolling is the concatenation of two combinatorial units in order to half the number of iterations. This means that with one clock cycle two rounds of DES will be calculated. Figure 5.3 shows the block diagram. This block diagram differs from Figure 5.2 only in the 2nd combinatorial logic unit. The initial and final permutations as well as the registers and multiplexers are the same.

Where is now the speed improvement? In the not unrolled version, one iteration of DES has the following simple timing model: $T_{mux} + T_{cl} + T_{reg}$ where T_{mux} denotes the time a signal needs to pass through a multiplexer, T_{cl} the delay introduced by the combinatorial logic, and T_{reg} the delay introduced by the register. So for the whole 16 rounds this sums up to: $16 * T_{mux} + 16 * T_{cl} + 16 * T_{reg}$.

The equation for the loop unrolled version looks like this: $T_{mux} + 2 * T_{cl} + T_{reg}$. This has to be executed 8 times, so that the over-all delay is now: $8 * T_{mux} + 16 * T_{cl} + 8 * T_{reg}$. The same principle can be applied to four unrolled DES rounds. The following list

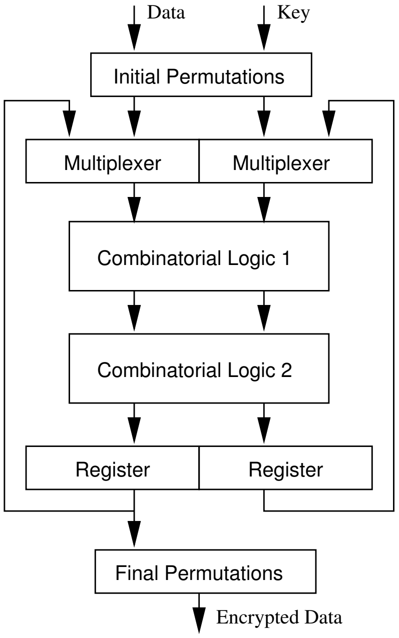


Figure 5.3: Block diagram of DES with 2 unrolled loops

shows the timing for each case.

$$\begin{array}{rcl}
\text{DES not unrolled} & : & 16 * T_{mux} + 16 * T_{cl} + 16 * T_{reg} \\
\text{2 unrolled loops} & : & 8 * T_{mux} + 16 * T_{cl} + 8 * T_{reg} \\
\text{4 unrolled loops} & : & 4 * T_{mux} + 16 * T_{cl} + 4 * T_{reg}
\end{array}$$

Obviously we can not reduce the delay introduced by the combinatorial logic units but we reduced the runs through the multiplexers and buffers by half. But there is another motivation for speed increase if modern design methods are applied. It is possible that the synthesis tools can optimize an unrolled design better, and therefore the logic can potentially be reduced. Also the routing can be more effective.

5.3 Pipelining

We now discuss the second architectural principle for accelerating DES. Pipelining tries to achieve a speed improvement in a different way. Instead of processing one block of data at a time, a pipelined design can process two or more data blocks. A design with two pipelines is shown in Figure 5.4. The block diagram in Figure 5.4 is very similar to the one with the two unrolled loops (Figure 5.3). The only difference is the additional buffer between the combinatorial logic units.

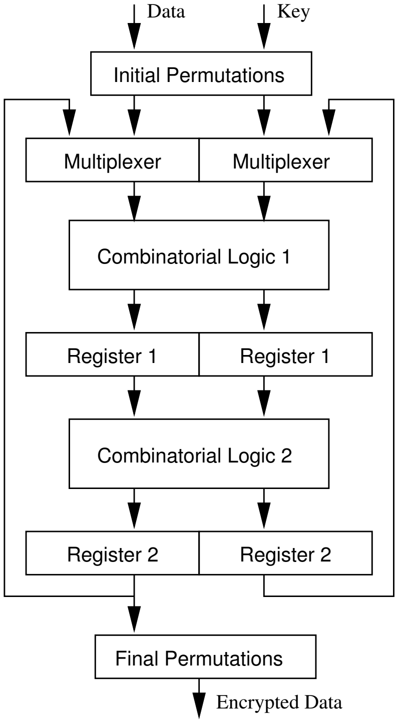


Figure 5.4: Block diagram of DES with 2 pipelines

The first block of data x_1 and the associated key k_1 are loaded and passed through the initial permutations and the multiplexer. The 1st combinatorial logic unit computes $x_{1,1}$ and $k_{1,1}$ which is stored into the 1st register block. On the next clock cycle $x_{1,1}$ and $k_{1,1}$ leave the 1st registers and the 2nd combinatorial logic unit computes $x_{1,2}$ and $k_{1,2}$ which is put into the 2nd register block. At the same time the second block of data x_2 and key k_2 are loaded and passed through the initial permutations, and the multiplexer, and the 1st combinatorial unit computes $x_{2,1}$ and $k_{2,1}$ which get moved into the 1st register block.

Now the pipeline is filled and with each clock cycle another iteration for two pairs of data and key are computed. The data which has entered the pipeline first, will also exit it first. At that time the next data and key pair can be loaded.

The advantage of this design is that two or more data–key pairs can be worked upon at the same time. As there is still only one instance of the initial permutations, the multiplexer and the final permutation, the cost in terms of resources on the chip will not be twice as high as if we implemented two full non pipelined DES designs. Also there has to be only one control logic which is just slightly more complicated than for a non pipelined DES design. The maximum clock speed should be roughly the same as during one clock cycle the same amount of logic resources has to be traversed as in the non pipelined design. It is also straight forward to design pipelines with more than two stages, e.g., with four.

5.4 Combination of Pipelining and Loop Unrolling

It is possible to combine both architecture acceleration techniques that we just described. Each pipeline would contain two unrolled loops. The resulting block diagram shown in Figure 5.5 looks similar to Figure 5.4 except that each combinatorial logic unit is duplicated. During one clock cycle two iterations of two data–key pairs get computed: $x_{1,4}$ and $k_{1,4}$ get computed from $x_{1,2}$ and $k_{1,2}$, and $x_{2,2}$ and $k_{2,2}$ get computed from x_2 and k_2.

5.5 Comparison and Design Decisions

As described in Section 4.4 some DES modes of operation require that the output of DES is used to compute the next input (e.g., the CFB mode). If such a mode is to be used, a pipelined design would not work, as it processes two data–key pairs at the same time. A loop unrolled design would work fine and is the only method for

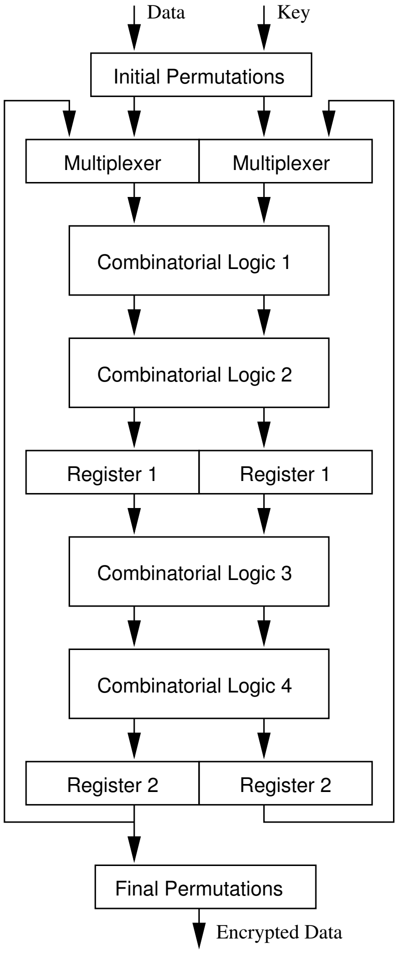


Figure 5.5: Block diagram of DES with 2 unrolled loops within 2 pipelines

speed-up that can be applied for such modes. In an application that is not subject to this constraint, like ECB-mode or ATM-counter mode, the pipelined versions can be used. A pipelined design should result in a higher speed-up than a loop unrolled design.

One major objective of this thesis was to obtain a realistic comparison of the different acceleration methods (loop unrolling, pipelining, combination of both). Table 5.1 shows the architecture versions we decided to implement.

Name	**Description**
DES_ED16	standard DES (16 iterations)
DES_ED8	DES with 2 unrolled loops (8 iterations)
DES_ED4	DES with 4 unrolled loops (4 iterations)
DES_ED16x2	DES with 2 pipelines
DES_ED16x4	DES with 4 pipelines
DES_ED8x2	DES with 2 pipelines each containing 2 unrolled loops

Table 5.1: Implemented DES architectures

Chapter 6

DES Design

This section is concerned with the design of the circuit. The next step after analyzing the architecture of DES is to break DES down into small elementary computational units, so called *function blocks* and then to analyze how they can be implemented efficiently. After this some further optimization can be done.

6.1 DES Function Blocks

In this section we will only describe a not-unrolled and not-pipelined version of DES. Also, only encryption is possible. The function blocks developed can then also be used for the more advanced designs.

In the previous section we have shown that the DES design comprises the initial permutation, the final permutation, registers and multiplexers. The combinatorial logic unit needs to be investigated further. It contains the Feistel network and the key scheduling.

The Feistel network, as shown in Figure 4.3, comprises a 32-bit XOR and the f-function. The f-function is composed of an expansion box, a 48-bit XOR, eight S-Boxes and a permutation box.

The key schedule needs shift registers and a permutation box. The shift registers have to rotate the bits by one or two positions depending on the round and change directions if the mode changes between encryption or decryption. The basic function blocks for all these operations are

- Permutation Boxes and Expansion Boxes
- Registers
- Multiplexers
- Standard Logic Functions (XOR)
- S-Boxes
- Shift Registers

6.2 Logic Resources

Every function block listed in the previous section will be analyzed here and ways to implement them will be shown.

6.2.1 Permutation Boxes and Expansion Boxes

Permutation boxes reorder the bits of a bit-string. Expansion boxes are a special form of permutation boxes; they also duplicate bits. Reordering and duplication of bits requires no logic resources, it can be implemented by wiring only. The outputs of the previous logic block are wired in a different (permuted) order to the next logic block. If the permutation is directly at the input or at the output of the device, which is the case for the initial permutations and the final permutation, the reordering takes place in the wiring of the I/O pins of the device and the logic blocks they are connected to. Therefore a permutation or expansion causes no additional delays, except some wiring delays if it complicates the wiring. Following is an example of the VHDL description of the *PC1BOX*, which is the initial permutation for the key.

```
library ieee;
use ieee.std_logic_1164.all;

ENTITY pc1box IS PORT
    ( CD     : IN  std_logic_vector (56 downto 1);
      KS     : OUT std_logic_vector (56 downto 1));
END pc1box;

ARCHITECTURE behave OF pc1box IS

BEGIN

    KS(1)  <= CD(53); KS(2)  <= CD(46); KS(3)  <= CD(39); KS(4)  <= CD(32);
    KS(5)  <= CD(52); KS(6)  <= CD(45); KS(7)  <= CD(38); KS(8)  <= CD(31);
    KS(9)  <= CD(24); KS(10) <= CD(17); KS(11) <= CD(10); KS(12) <= CD(3);
    KS(13) <= CD(51); KS(14) <= CD(44); KS(15) <= CD(37); KS(16) <= CD(30);
    KS(17) <= CD(23); KS(18) <= CD(16); KS(19) <= CD(9);  KS(20) <= CD(2);
    KS(21) <= CD(50); KS(22) <= CD(43); KS(23) <= CD(36); KS(24) <= CD(29);
    KS(25) <= CD(22); KS(26) <= CD(15); KS(27) <= CD(8);  KS(28) <= CD(1);
    KS(29) <= CD(25); KS(30) <= CD(18); KS(31) <= CD(11); KS(32) <= CD(4);
    KS(33) <= CD(54); KS(34) <= CD(47); KS(35) <= CD(40); KS(36) <= CD(33);
    KS(37) <= CD(26); KS(38) <= CD(19); KS(39) <= CD(12); KS(40) <= CD(5);
    KS(41) <= CD(55); KS(42) <= CD(48); KS(43) <= CD(41); KS(44) <= CD(34);
    KS(45) <= CD(27); KS(46) <= CD(20); KS(47) <= CD(13); KS(48) <= CD(6);
    KS(49) <= CD(56); KS(50) <= CD(49); KS(51) <= CD(42); KS(52) <= CD(35);
    KS(53) <= CD(28); KS(54) <= CD(21); KS(55) <= CD(14); KS(56) <= CD(7);

END behave;
```

6.2.2 Registers

Registers (data buffers) can be implemented either in combinatorial logic or using RAM elements. Most modern FPGAs have RAM/ROM elements built in which are more effective than combinatorial logic for these purposes.

6.2.3 Multiplexers

Multiplexers can easily be implemented using combinatorial logic. The synthesizing tools will try to use predefined functions from the FPGA vendor to implement them. The same is valid for the registers too. Here is an example of the VHDL description of a 32-bit multiplexer.

```
library ieee;
use ieee.std_logic_1164.all;

ENTITY mux32 IS PORT
    ( A   : IN  std_logic_vector (31 downto 0);
      B   : IN  std_logic_vector (31 downto 0);
      O   : OUT std_logic_vector (31 downto 0);
      sel : IN std_logic);
END mux32;

ARCHITECTURE behave OF mux32 IS

    signal element : std_logic_vector (31 downto 0);

BEGIN

    O <= element;
    element <= B WHEN sel = '1' ELSE
               A;

END behave;
```

6.2.4 Standard Logic Functions

Standard logic functions, such as AND, OR, XOR are composed of basic gates. Their performance does not depend of the width of the bit-string they have to operate upon, e.g., the 32-bit XOR performs equally to the 48-bit XOR used in the Feistel network. Following is a VHDL example of the 32-bit XOR.

```
library ieee;
use ieee.std_logic_1164.all;

ENTITY xormod IS PORT
    ( A : IN  std_logic_vector (31 downto 0);
      B : IN  std_logic_vector (31 downto 0);
      Q : OUT std_logic_vector (31 downto 0));
END xormod;

ARCHITECTURE behave OF xormod IS

BEGIN
        Q <= A XOR B;
END behave;
```

6.2.5 S-Boxes

S-Boxes are look-up tables which are of size 6 x 4 and therefore contain 64 4-bit values (see Section 4.1). The implementation of the S-Boxes is cruical for an efficient DES design [7]. If they are implemented via combinatorial logic they need hundreds of logic elements. A study by Greg Haskins [7] shows that using ROM elements is the most efficient way to implement S-Boxes.

6.2.6 Shift Registers

The shift registers[1] used in the key schedule can be classified as *combinatorial shifters*, *decisive shifters* and *directional shifters*. Figure 6.1 shows an overview of the different shifters. All these shifters rotate a four-bit bit-string by at most 1 bit.

Combinatorial Shifters shift by a fixed number of positions and they shift always, not depending on a clock. They are essentially permutations.

Decisive Shifters have an additional input upon which they decide if the data should be shifted or not. A decisive shifter can be realized with a multiplexer.

Directional Shifters are very similar to decisive shifters. They also have the additional input upon which they decide if the data has to be shifted right or left. A directional shifter can also be realized with a multiplexer.

[1] shift registers are used here synonymously for rotators

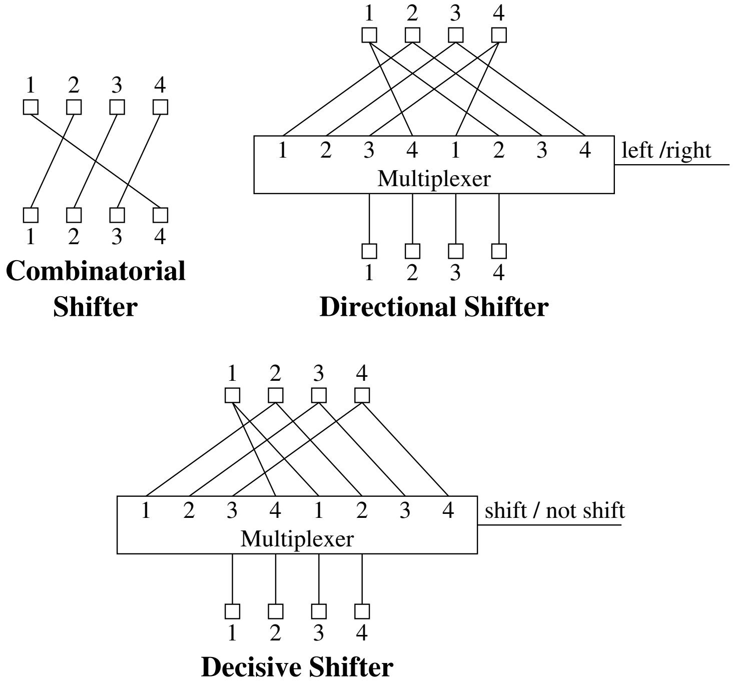


Figure 6.1: Implementation of shift registers

6.3 Optimizations

Figure 6.2 shows a detailed block diagram of standard DES. It is a refinement of the high-level diagram in Figure 5.2. It contains all the function blocks discussed in the previous section. This design has been implemented under the name *DES16 Version 1.1.*

6.3.1 LogiBLOX

One simple way of optimizing the design is to use LogiBLOX. LogiBLOX are preconfigured, optimized modules for Xilinx FPGAs. The performance of the LogiBLOX does not depend on the quality of the synthesizing tool, as modules described in VHDL would.

We created a design using LogiBLOX named: *DES16 Version 1.2.* The functional

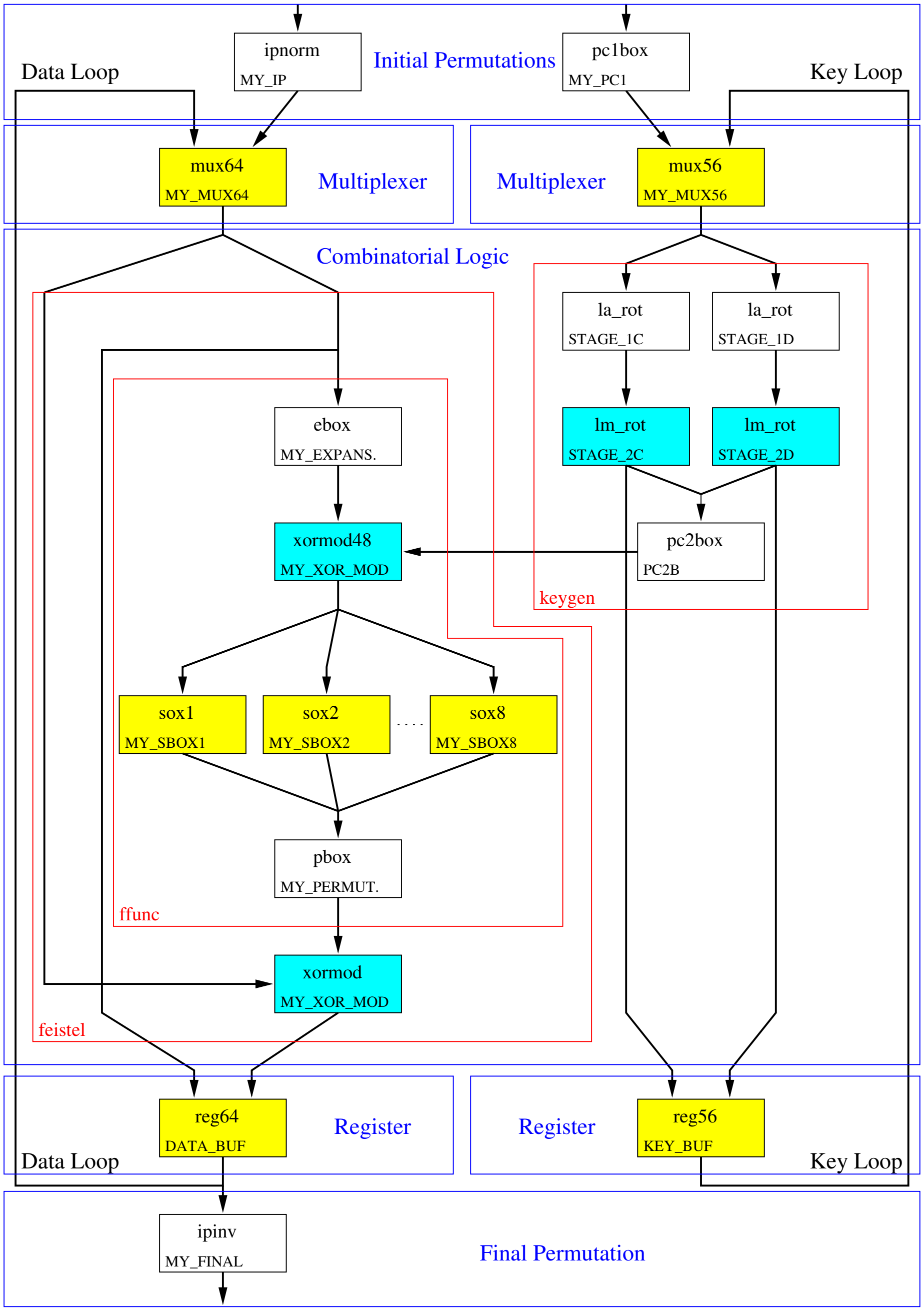


Figure 6.2: Detailed block diagram of DES

blocks we implemented in this version as LogiBLOX have a light grey background in the block diagram shown in Figure 6.2.

6.3.2 Timing Analysis

Before the actual implementation we can do a rough timing analysis of the design shown in Figure 6.2. Boxes with a white background denote permutation and expansion boxes as well as combinatorial shifters. They are just wiring resources so they can be assumed to be very fast. Boxes with a background color are using logic resources, so it will take some time for data to propagate through them.

Each iteration, except the 1st, starts with the data and the key coming out of the registers and through the multiplexers. Then the data passes through an expansion box and into an 48-bit XOR. The key passes through a combinatorial shifter and then through a decisive shifter. The result of this goes through a permutation and also to the 48-bit XOR.

The data XOR-ed with the sub-key is applied to the S-Boxes, another permutation and finally through another XOR. After this data and key are at the input of the registers. Figure 6.3 shows how these function elements are executed in successive order from left to right. Function elements executed concurrently are shown in the same column.

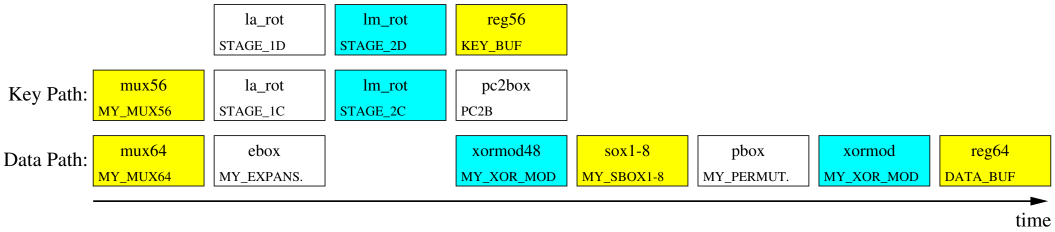


Figure 6.3: Rough timing diagram of DES

This diagram shows a problem. The 48-bit XOR (xormod48) can not be executed[2] until the key is propagated through the conditional shifters (lm_rot).

6.3.3 Improved Sub-Key-Generation Logic

The problem shown in the timing analysis section (Section 6.3.2) leads to a different approach for the sub-key generation. As Figure 6.3 shows, the problem is that the current sub-key is generated too late and the data path has to "wait". After the sub-key generation is done the data-path has to execute more steps. That time is unused on the key-path. A higher level of parallelism would be valuable.

The solution to this problem is to perform the sub-key computations while the data moves through the S-Boxes and the final XOR. That means, that the sub-key would have to be precomputed by one clock cycle and send to the XOR (xormod48) right at the beginning of the next clock cycle.

In order for this to work we have to be able to give the f-function during the 16th round the 16th sub-key and at the same time load a new key and pre-compute the 1st sub-key for the next data packet. Therefore we have to move the multiplexer between the permutation (pc2box) and the rest of the key generation. Figure 6.4 shows how a sub-key generation according to this schema would look like.

6.3.4 Encryption – Decryption

As we are generating the sub-key during the time the data moves through the S-Boxes and the final XOR, we have more time than we would need for just a sub-key generation for encryption.

It is possible to include the logic for decryption too at the expense of more logic resources, but with the same time constraints. As described in Subsection 4.3 decryption means that we have to shift the key right, either none times, or one time, or

[2] executed means that it will produce the final result

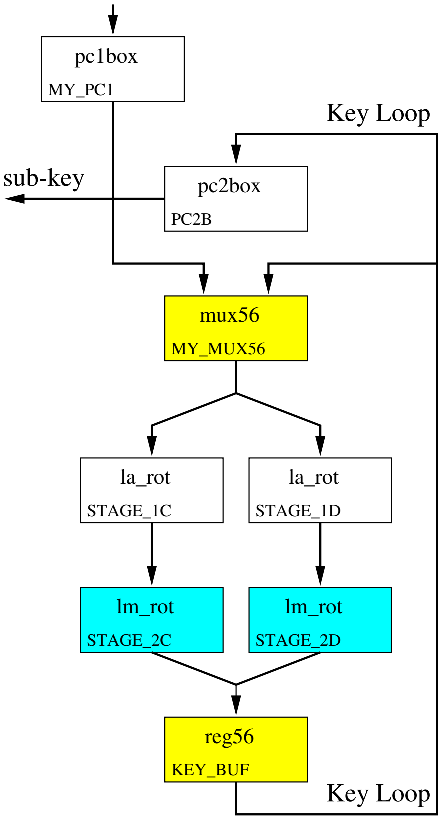


Figure 6.4: New sub-key generation

two times. Therefore we can not use combinatorial shifters but only decisive shifters. The key propagates through two branches. In the first branch it is shifted left by 1 or 2 positions, depending on the round, for encryption. In the second branch the key is shifted right by 0, 1, or 2 positions, depending on the round, for decryption. A multiplexer at the end switches between the results of the two branches and therewith switches between encryption or decryption. This way a simple directional shifter is implemented. Figure 6.5 shows the block diagram for this.

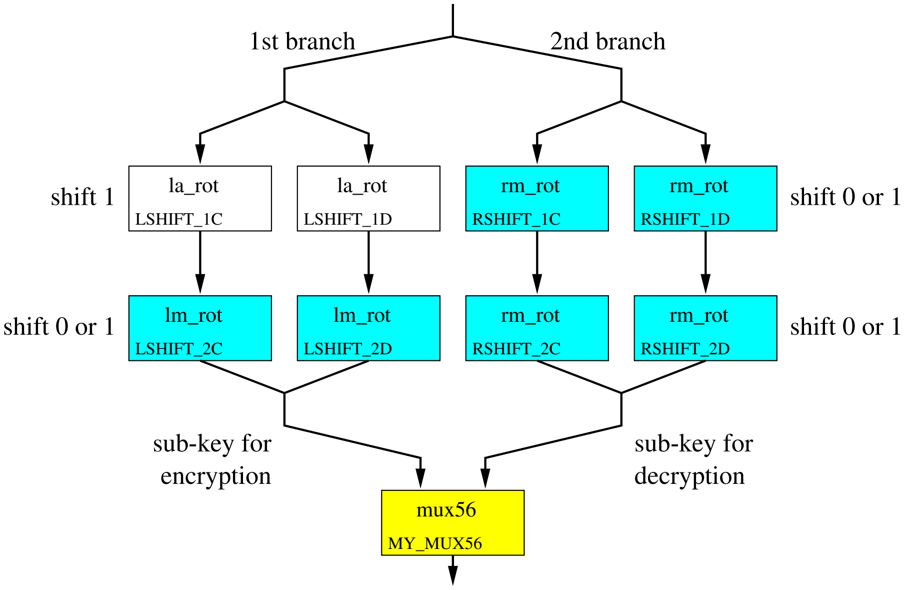


Figure 6.5: Encryption – decryption sub-key generation

The functional blocks named *rm_rot* are decisive right shifters, the blocks named *la_rot* are combinatorial left shifters and *lm_rot* are decisive left shifters. The timing diagram for this design is shown in Figure 6.6.

The white permutation boxes are assumed to be free of delay. The grey boxes are assumed to have all the same delay. As the sub-key is ready immediately the data-path and the key-path are almost independent. As opposed to the timing diagram shown in Figure 6.3 which had 6 grey boxes in a row (5 in the data path and 1 wait

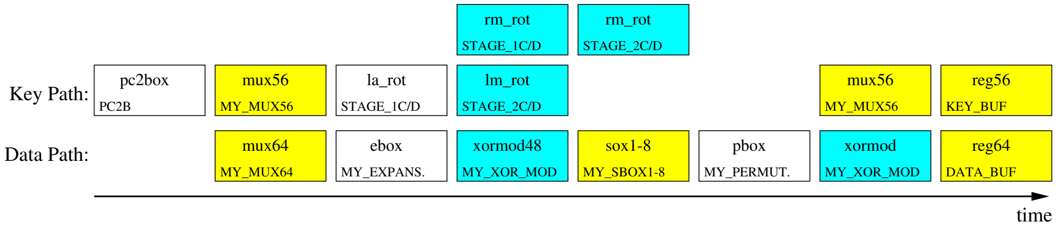


Figure 6.6: Encryption – decryption timing diagram

state for the sub-key) this diagram has only 5 grey boxes in a row. We implemented this version of DES under the name: *DES_ED16*.

6.4 Control Logic

The control logic for this DES design is a simple state machine. A non loop unrolled implementation of DES needs 16 iterations to compute the cipher text. This can be realized with a state machine comprising 16 states ordered in one loop.

In Section 6.3.3 we showed the advantages of computing the sub-key one round in advance. For this to work we need to create a state machine with an initial state to preload the key before the data is loaded in state 1. In state 16, while the last iteration of the data is calculated, the key for the next operation is preloaded. The state machine does not need to return to the initial state but can continue right to state 1. Figure 6.7 shows the state transition diagram.

The transition from one state to the next in sequence is triggered by the clock. A clock enable signal is also implemented which makes it possible to stop the state machine in any given state for as many clock cycles as wanted. A reset signal in any state causes the state machine to return to the INIT state.

The control signals the state machine controls are not shown in the diagram as they vary from design to design. However, here is a short overview about the control signals the state machine has to provide.

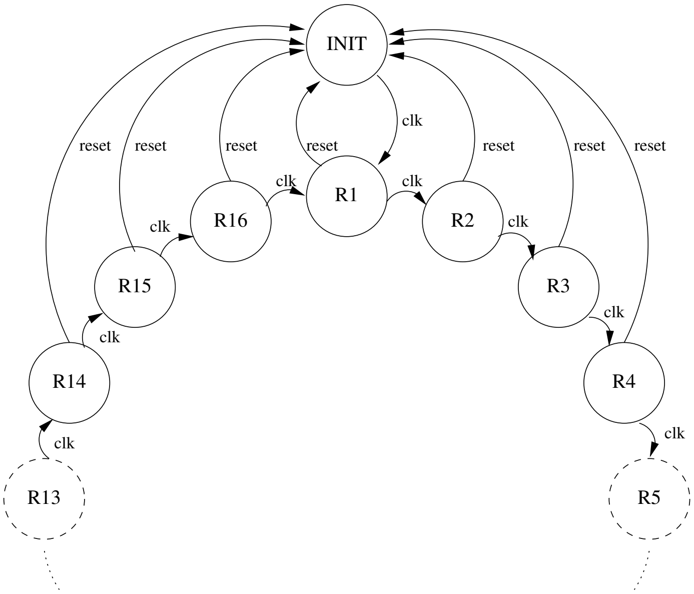


Figure 6.7: DES control state machine

KE *key expected*, signals an external entity that a key is expected at the inputs (KE = high)

IE *input expected*, signals an external entity that the data is expected at the inputs (IE = high)

OV *output valid*, signals an external entity that the output data is valid (OV = high), otherwise the data at the output is not valid (OV = low)

Data_Sel signal for the input multiplexer of the data path to either load new data (data_sel = low) or forward data from the data loop (data_sel = high)

Key_Sel signal for the input multiplexer of the key path to either load a new key (key_sel = high) or forward the key from the key loop (key_sel = low)

SFT *shift*, signals the sub-key generation logic to not shift the key (SFT = low) if in decryption mode (ST1 has to signal *one position*).

ST1 *shift two* signals the sub-key generation logic to shift the key by one (ST1 = low) or two positions (ST1 = high).

Other control signals are needed for different versions of the design. These are described in the respective sections. The state machine for a design with loop unrolling contains as many states as iterations needed plus one initial state. That means, the state machine for a design with 2 unrolled loops comprises $16/2 + 1 = 9$ states and for a design with 4 unrolled loops only $16/4 + 1 = 5$ states. Therefore loop-unrolling results in simpler state machines.

6.5 Filling Pipelines

A pipelined design introduces an initial delay. The reason is that the pipelines have to be filled first. In an ideal 4 pipeline design it would take 4 clock cycles to fill the pipelines.

The designs of type *DES_ED** listed in Table 5.1 and the design *DES_MQP* with encryption and decryption mode, perform key precomputation. As described in Section 6.3.4 the key has to be loaded one clock cycle before the data. Therefore it is possible to use the same input pins for key and data. The data multiplexer and key multiplexer can demultiplex the combined input at no additional cost. The advantage is that less IO-pins are used.

The multiplexed data-key input complicates the loading of pipelines. The key has to be loaded first and then the associated data. This requires that during the clock cycle after half the pipelines are filled nothing is loaded. Starting with the following clock cycle the rest of the pipelines can be filled. Table 6.1 shows how the pipelines can be filled the most efficient way. In the states not shown no key or data is loaded. The state *R16* behaves the same way as the state *INIT*.

From Table 6.1 it can be seen that a design with eight pipelines could not be

State	INIT	R1	R2	R3	R4	R5	R6	R7	R8	R9
Input for 2 pipeline design	K1	D1	–	K2	D2	–	–	–	–	–
Input for 4 pipeline design	K1	D1	K2	D2	–	K3	D3	K4	D4	–

Table 6.1: Loading pipelines

implemented in this way. It would take eight states (*INIT* – *R7*) to load the first four key–data pairs, and during *R8* nothing could be loaded. It would take another eight states to load the remaining four key–data pairs. But during state *R16* the next *K1* is to be loaded. Therefore only seven out of eight pipelines could be used. A solution to this problem is to have separate key and data busses.

Chapter 7

DES Implementation

We implemented various architecture versions of DES (see Table 5.1) and a modified version for an *MQP* (Major Qualifying Project or senior thesis). These architectures were also ported to different chips.

7.1 FPGA Choice

We have chosen FPGAs from Xilinx for our implementation. This decision was based on research described in [7]. The major relevant discovery in [7] was that it is difficult to implement more than one set of S-boxes with other commercial available reconfigurable devices such as Altera EPLDs. However, multiple sets of S-boxes are needed for loop unrolling and for pipelining. We therefore had to choose a vendor who could supply us with devices large enough for this task.

7.2 VHDL-Source

For each function block (see Section 6.1) a separate VHDL file was created, except for the ones implemented using LogiBLOX. Table 7.1 lists the files and their function.

Filename	**Function**
bigbuff.vhd	64 bit data buffer and 56 bit key buffer with one clock each
bigmux.vhd	64 bit data and 56 bit key multiplexer with two switches
control.vhd	Control Logic (state machine)
des.vhd	Top level description file for DES
ebox.vhd	Expansion permutation
feistel.vhd	One iteration of the feistel network
ffunc.vhd	F-Function
initial.vhd	The initial permutations for the plaintext and key
ipinv.vhd	inverse initial permutation
ipnorm.vhd	Initial Permutation
iteration.vhd	One complete iteration inc feistel and sub-key generation
key1gen.vhd	Key generation first round only
keygen.vhd	Key generation
la_rot.vhd	Combinatorial Rotation Unit. Performs a 1 bit cyclic left shift automatically
lm_rot.vhd	Combinatorial Left Rotation Unit. Performs a 1 bit cyclic left shift or a pass through, depending on the mode bit
module_pack.vhd	module definition file
mux32.vhd	32 bit 2x1 multiplexer
mux56.vhd	56 bit 2x1 multiplexer
mux64.vhd	64 bit 2x1 multiplexer
pbox.vhd	Permutation box
pc1box.vhd	PC-1 DES Key Scheduler permutation
pc2box.vhd	PC-2 Key Scheduler permutation
ra_rot.vhd	Combinatorial Rotation Unit. Performs a 1 bit cyclic right shift
reg28.vhd	28 bit register
reg56.vhd	56 bit register
reg64.vhd	64 bit register
rm_rot.vhd	Combinatorial Left Rotation Unit. Performs a 1 bit cyclic right shift or a pass through, depending on the mode bit
sboxes.vhd	Main SBOX module
xormod.vhd	32 bit XOR Module
xormod48.vhd	48 bit XOR Module

Table 7.1: VHDL source files and their function

These files can be divided into files that describe core functions and are not depending on other files, and files that describe higher level modules. The file *module_pack.vhd* contains the component instantiation of all the components (modules). The core function files are: ebox.vhd, ipinv.vhd, ipnorm.vhd, la_rot.vhd, lm_rot.vhd, mux32.vhd, pbox.vhd, pc1box.vhd, pc2box.vhd, ra_rot.vhd, reg28.vhd, rm_rot.vhd, xormod.vhd, and xormod48.vhd. Some files are written in two versions, one using VHDL to describe the core functions and the other employing LogiBLOX to provide the core function: mux56.vhd, mux64.vhd, reg56.vhd, and reg64.vhd. The other files are depending on these core modules or LogibBLOX.

The same files are used in different revisions for the various implementations. In order to keep track of which revision of a certain file is used in which version of the DES implementation a revision control system *RCS* was employed.

7.3 LogiBLOX

We created LogiBLOX versions for registers, multiplexers, S-Boxes and some shifters. The LogiBLOX were not subject to frequent changes, so there was no need to have them managed by *RCS*. Furthermore the LogiBLOX are all instantiated from within VHDL files. Table 7.2 lists the LogiBLOX created and their function.

LogiBLOX can be created by the interactive graphical tool *lbgui*. The tool creates **.ngo* files which are inferred by the Xilinx design manager, VHDL simulation models, and instantiation templates.

7.4 Designs Implemented

We implemented DES in several versions to compare the different architectures and the influence of the size of the FPGAs on the maximum speed. Many designs were implemented in the chip: `XC4013-3-PG223`. This device offers enough resources even

Filename	**Function**
mux16l	16 bit multiplexer
mux32l	32 bit multiplexer
mux8l	8 bit multiplexer
reg16c	16 bit register with clock enable
reg16l	16 bit register
reg32c	32 bit register with clock enable
reg32l	32 bit register
reg8c	8 bit register with clock enable
reg8l	8 bit register
shift2	2 bit shift register with clock enable, MSB out, LSB in, and parallel out
shift4	4 bit shift register with clock enable, MSB out, LSB in, and parallel out
sox1	S-Box 1
sox2	S-Box 2
sox3	S-Box 3
sox4	S-Box 4
sox5	S-Box 5
sox6	S-Box 6
sox7	S-Box 7
sox8	S-Box 8

Table 7.2: LogiBLOX and their function

for more advanced designs than the simple *DES16*. Furthermore a group of students is using this device for their MQP. The design *DES_MQP* was tailored to their specific needs.

7.4.1 DES16

This is the very first design we implemented. One encryption requires 16 clock cycles, no pipelining or unrolling techniques were employed. *DES16* only supports encryption and the sub-keys are not precomputed.

We implemented two versions of *DES16*. In version 1.1 only the S-Boxes were implemented using LogiBLOX. In version 1.2 LogiBLOX were used also for the registers and multiplexers.

The schematic of *DES16* for both versions is shown in Figure 6.2. The target for both versions is the chip `XC4013E-3-PG223`.

7.4.2 DES_ED16

DES_ED16 is the first design using the *one round sub-key precomputation* technique described in Chapter 6.3.3 and the modification for encryption – decryption shown in Chapter 6.3.4. All subsequent designs are employing these features. One encryption or decryption takes 16 clock cycles.

DES_ED16 was implemented in three different versions. The difference between the three versions is only the target device. Version 1.1 is implemented on the device `XC4013E-3-PG223`, version 1.2 on the device `XC4008E-3-PG233` and version 1.3 on the device `XC4025E-3-PG223`, these devices differ in the amount of logic resources they provide.

The control logic for this design has to provide the following signals: *ke*, *ie*, *ov*, *data_sel*, *key_sel*, *SFT*, and *ST1* (for a description see Chapter 6.4). Figure 7.1 shows the timing diagram for these signals.

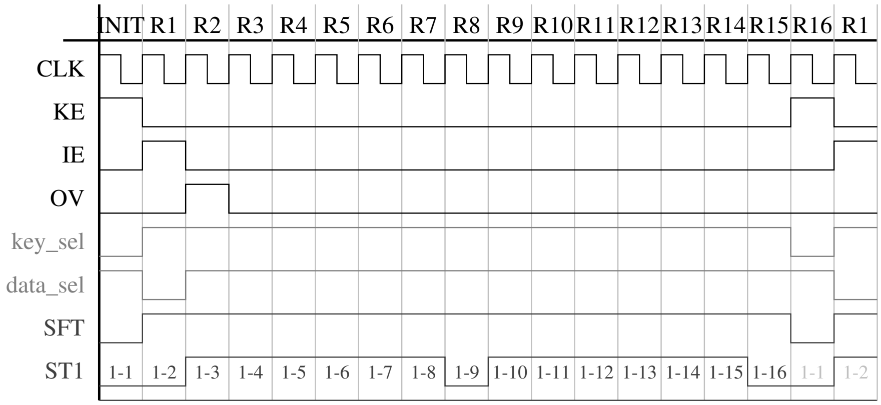


Figure 7.1: Control signals for *DES_ED16*

The signal OV is without any function for this design. The output is valid at the same time new data gets loaded. The signal *IE* is to be used for both purposes, output valid and data input expected. The numbers shown next to the signal *ST1* denote the sub-key computed during the respective state. In state *INIT* the 1st sub-key is generated, during state *R1* the 2nd sub-key, and so on. During state *R16* the 1st sub-key of the next key is computed which is indicated through light shade of gray.

7.4.3 DES_MQP

The design *DES_MQP* is a special design for an MQP based on the *DES_ED16* design. The only difference is that it uses a bidirectional 64-bit bus for data and key input and data output.

DES_MQP was implemented in 2 different versions. The only difference between these versions is the target device. Version 1.1 was implemented on the device `XC4013-3-PG223` with speed grade *-3*, version 1.2 on a device with speed grade *-2*: `XC4013-2-PG233` .

The control signals are the same as for *DES_ED16* shown in Figure 7.1. The signal OV indicates that the output is put on the bidirectional bus. If OV is low the output is tri-stated. This means that this chip is accessing the bus for only three clock cycles, *loading key*, *loading data* and *output result*. During the remaining $16 - 3 = 13$ clock cycles the bus is tri-stated. While the bus is try-stated by one chip other chips could access that it. Up to 5 chips could be run in parallel of the same bus (16 clock cycles divided by 3 clock cycles for I/O per chip) if their loading and output cycles are scheduled in the right order.

7.4.4 DES_ED8

This is the first loop unrolled design. One encryption or decryption takes 8 clock cycles, therefore the state machine has to support only 9 states. *DES_ED8* was targeted for the `XC4013-3-PG223` device in which it fits comfortably.

The control logic has one additional signal *ST2*. It has basically the same function as *ST1* but operates on the second sub-key generator. Figure 7.2 shows the timing diagram. During the state *R8* the 16th sub-key gets generated in the 2nd sub-key generator and the 1st sub-key generator calculates the 1st sub-key for the next data packet, indicated through a light shade of gray. The mode (encryption or decryption) of the next data packet is entirely independent of the mode for the current one.

During state *R1* data is first encrypted with the precomputed first sub-key from the previous state. At the same time the second sub-key is precomputed in the second sub-key generator. As soon as the second sub-key generator is finished the first sub-key generator produces the third sub-key to be used in the next state and the data is encrypted with the precomputed second sub-key.

7.4.5 DES_ED4

This is the second loop unrolled design with 4 unrolled loops. One encryption or decryption takes 4 clock cycles. The state machine supports 5 states. *DES_ED4* was tar-

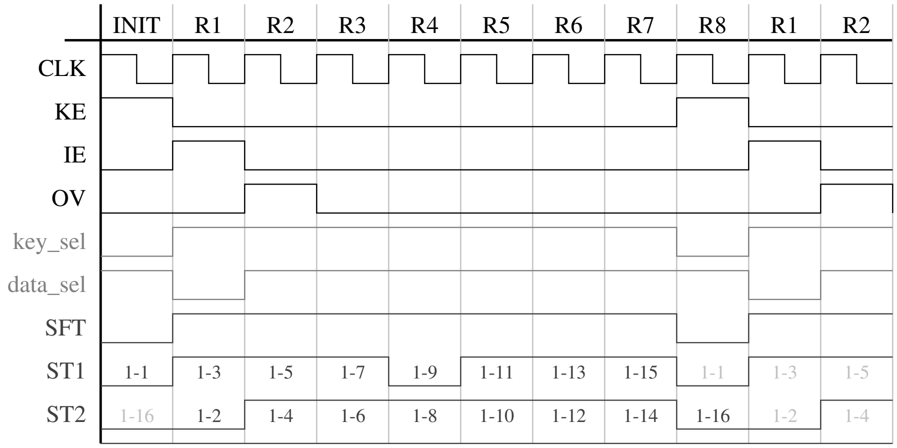


Figure 7.2: Control signals for *DES_ED8*

geted for the `XC4028EX-3-PG299` device. An implementation on the `XC4025E-3-PG223` device failed even though it has enough logic resources. The lack of wiring resources made a change from the `XC4000E` series to the `XC4000EX` series necessary.

The control logic has three additional signals *ST2*, *ST3*, and *ST4*. These signals operate on the second, third and fourth sub-key generators. Figure 7.3 shows the timing diagram. During state *R4* the 14th, 15th, and 16th sub-key get generated by the second, third and fourth sub-key generator, and the first sub-key generator generates the first sub-key for the next data packet, indicated through light shade of gray. The mode (encryption or decryption) of the next data packet is entirely independent of the mode for the current one.

During the state *R1* data is first encrypted with the precomputed first sub-key from the previous state. At the same time the second sub-key is created by the second sub-key generator. As soon as the second sub-key generator is finished the third sub-key generator generates the third sub-key and at the same time data is encrypted with the second sub-key and so on.

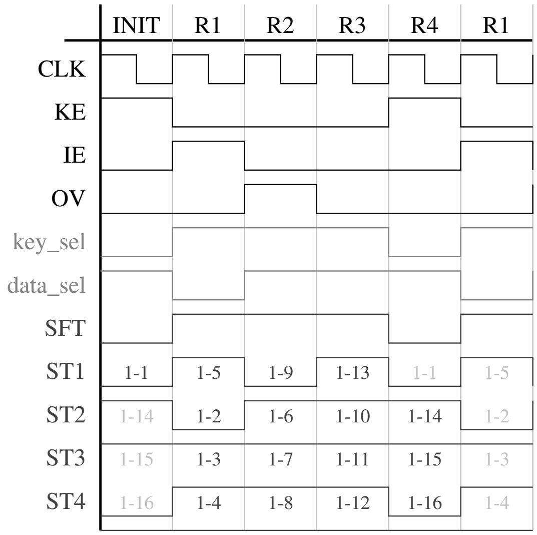


Figure 7.3: Control signals for *DES_ED4*

7.4.6 DES_ED16x2

DES_ED16x2 is the first pipelined design. One encryption or decryption takes 16 clock cycles, two operations can run at the same time. The modes of both operations (encryption or decryption) are independent of each other; one data block can be encrypted while the other is being decrypted, or both can be encrypted or decrypted. *DES_ED16x2* was targeted for the `XC4013E-3-PG223` device.

DES_ED16x2 has one additional signal: *ST2* which operates on the second sub-key generator. Figure 7.4 shows the timing diagram. As this is a pipelined design it can work on 2 data blocks at the same time, hence the notation *2-14* which denotes the 14th sub-key for the 2nd data block. The key for the first data block gets loaded during state *R16* or *INIT* followed by the first data block in the next state. The key for the second data block gets loaded during state *R3* followed by the second data block in the next state.

During state *R16* the 14th sub-key for the second data block is being generated by the second sub-key generator, and the first sub-key generator computes the 1st sub-key of the new first data block. In state *R1* the first sub-key generator computes the 15th sub-key for the second data block and the second sub-key generator the 2nd sub-key of the new first data block, and so on.

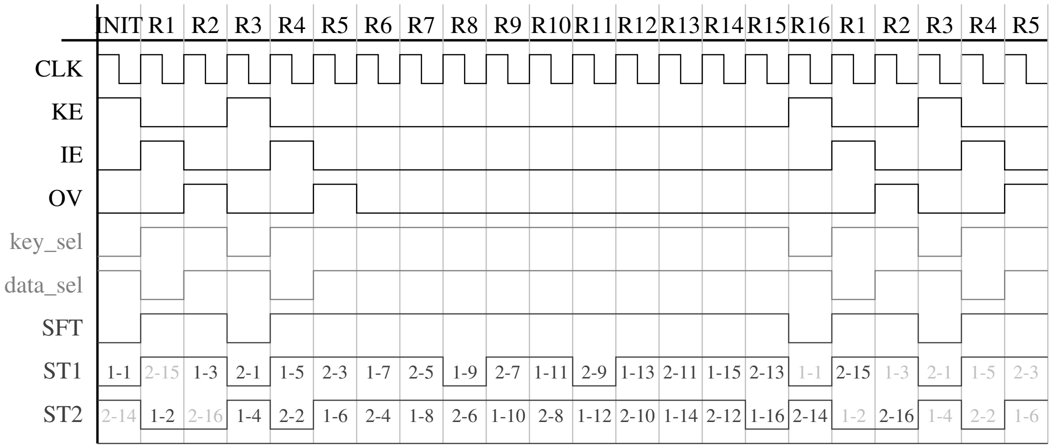


Figure 7.4: Control signals for *DES_ED16x2*

7.4.7 DES_ED16x4

This is the second pipelined design, comprising 4 pipelines. Each encryption or decryption takes 16 clock cycles; 4 operations can be handled at the same time. The modes of the operations are independent from each other; the mode (encryption or decryption) can be selected for each operation separately.

DES_ED16x4 was implemented in 2 different versions. The only difference between these versions is the target device. Version 1.1 was implemented on the `XC4025E-3-PG223` device and version 1.2 a device of a different family: `XC4028EX-3-PG299`.

The control logic provides three additional signals: *ST2*, *ST3*, and *ST4* which operate on the second, third and fourth sub-key generator. Figure 7.5 shows the timing diagram. The sub-key generation is straight forward and can be seen in the Figure.

The key for the first data block gets loaded during state *R16* or *INIT* followed by the first data block in the next state. The key for the second data block gets loaded during state *R2* followed by the second data block in the next state. The key for the fourth and fifth block get loaded during the states *R5* and *R7* respectively, the data blocks follow one stage later *R6* and *R8*. Initially it takes 8 clock cycles for all the pipelines to get filled.

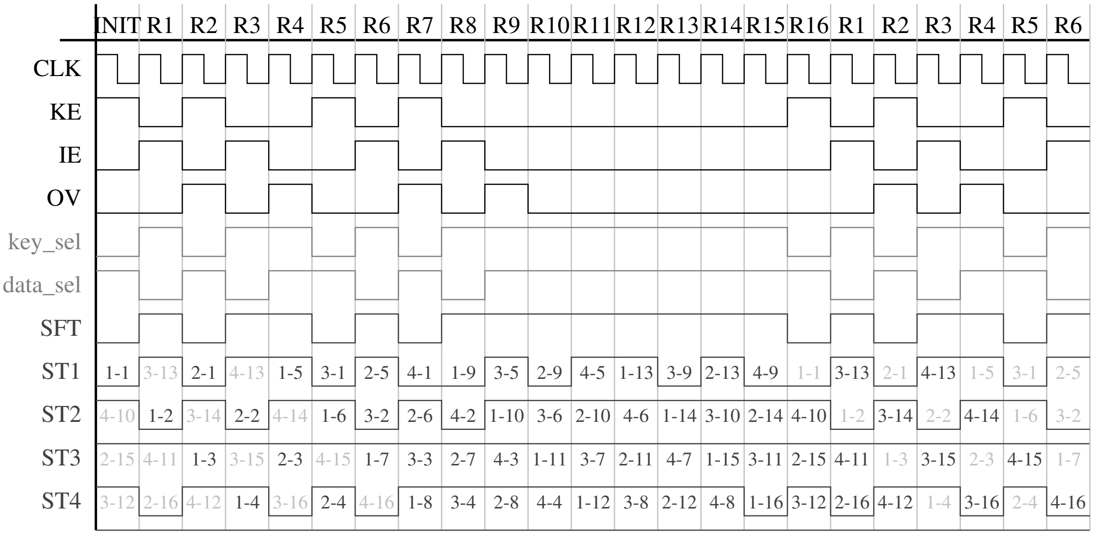

INIT R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15 R16 R1 R2 R3 R4 R5 R6
CLK
KE
IE
OV
key_sel
data_sel
SFT
ST1 1-1 3-13 2-1 4-13 1-5 3-1 2-5 4-1 1-9 3-5 2-9 4-5 1-13 3-9 2-13 4-9 1-1 3-13 2-1 4-13 1-5 3-1 2-5
ST2 4-10 1-2 3-14 2-2 4-14 1-6 3-2 2-6 4-2 1-10 3-6 2-10 4-6 1-14 3-10 2-14 4-10 1-2 3-14 2-2 4-14 1-6 3-2
ST3 2-15 4-11 1-3 3-15 2-3 4-15 1-7 3-3 2-7 4-3 1-11 3-7 2-11 4-7 1-15 3-11 2-15 4-11 1-3 3-15 2-3 4-15 1-7
ST4 3-12 2-16 4-12 1-4 3-16 2-4 4-16 1-8 3-4 2-8 4-4 1-12 3-8 2-12 4-8 1-16 3-12 2-16 4-12 1-4 3-16 2-4 4-16

Figure 7.5: Control signals for *DES_ED16x4*

7.4.8 DES_ED8x2

This design is a mixture between a pipelined and a loop unrolled design. It contains two pipelines with each two unrolled loops. Each encryption or decryption takes 8 clock cycles, 2 operations can be processed at the same time. The modes of both operations (encryption or decryption) are independent from each other. *DES_ED8x2* was targeted for the `XC4028EX-3-PG299` device.

The loading of the keys and the data packets is similar to the design *DES_ED16x2*. But after 8 clock cycles the result is already computed and the next loading cycle begins.

The control logic provides three additional signals: *ST2*, *ST3*, and *ST4* which operate on the second, third and fourth sub-key generator. Figure 7.6 shows the timing diagram. The sub-key generation is straight forward and can be seen in the diagram.

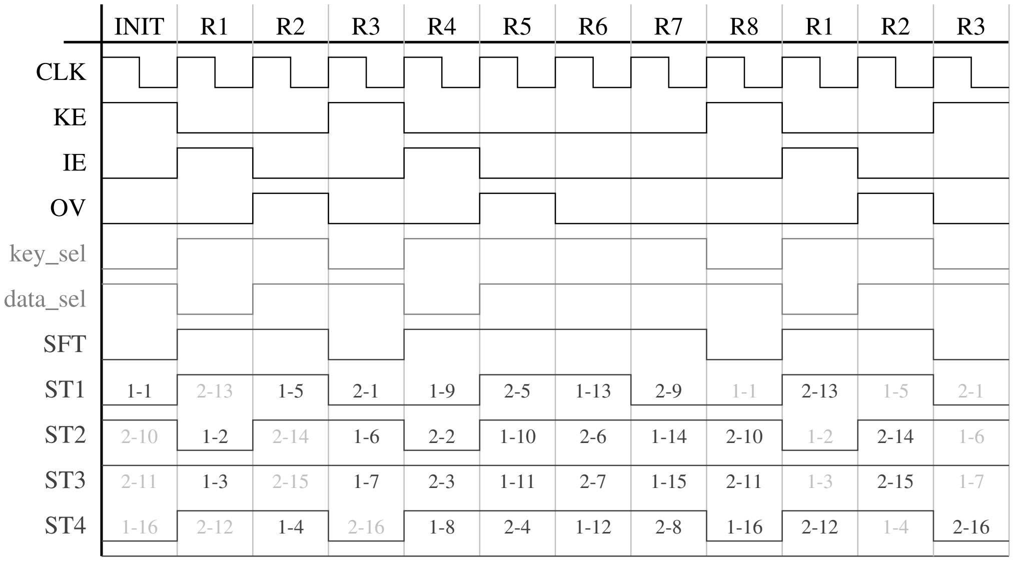


Figure 7.6: Control signals for *DES_ED8x2*

Chapter 8

Results

We implemented multiple versions of each architecture option listed in Table 5.1 in order to evaluate their effectiveness. We also implemented some designs multiple times with varying chip parameters in order to judge their influence on the performance. In the following sections we compare the different designs. In most cases the designs are compared to the design *DES_ED16* which serves as our reference model.

The unit **CLB** stands for *combinatorial logic block* which is employed by Xilinx to measure the amount of logic resources on a device. We are using it here to compare the amount of logic resources used by a given design.

The abbreviation **CLU** stands for *combinatorial logic unit* (see Chapter 5).

8.1 Loop Unrolling

We implemented two loop unrolled versions: *DES_ED8* and *DES_ED4*. The design *DES_ED8* contains two combinatorial logic units (*CLU*, see Section 5.2) and therefore encrypts or decrypts one data block in 8 clock cycles. The design *DES_ED4* contains four CLUs and provides the result after 4 clock cycles. Both designs are compared with the design *DES_ED16* in Table 8.1.

Design	**Chip**	**CLBs**	**CLBs** per CLU	**Min CLK** in ns	**Data Rate** per CLU in Mbit/s	**Data Rate** in Mbit/s
DES_ED16	XC4008E-3-PG223	262	262	40.4	94.5	**94.5**
DES_ED8	XC4013E-3-PG223	443	222	54.0	70.6	**141.3**
DES_ED4	XC4028EX-3-PG299	722	241	86.7	44.0	**176.0**

Table 8.1: Comparison of loop unrolled architectures

The design *DES_ED8* is 50% faster than *DES_ED16* whereas the resource consumption (in CLBs) increases by 69%. The design *DES_ED4* is only 25% faster than *DES_ED8*, the speed increase is only half as much as from the first unrolling. The resource consumption increases by 63%.

The number of CLBs divided by the number of CLUs indicates that the amount of logic resources consumed per unrolled CLU is almost constant. The speed divided by the number of CLUs shows that the speed for one CLU in the design *DES_ED4* is less then half the speed of *DES_ED16*. From this we can see that the further we unroll the design the lesser amount of speed-up we can gain.

8.2 Pipelining

We implemented two pipelined designs, *DES_ED16x2* and *DES_ED16x4*. The design *DES_ED16x2* contains two CLUs and therefore 2 pipelines and the design *DES_ED16x4* contains four CLUs and therefore 4 pipelines. The encryption or decryption of one block of data takes in both cases 16 clock cycles. Table 8.2 compares both designs with the design *DES_ED16*.

The speed divided by the number of CLUs shows that is stays almost constant for all designs. The lower speed for the design *DES_ED16x2* is caused by the lack of wiring resources on the device which results in a less efficient design. This phenomenon is further examined in Section 8.4.3.

Design	**Chip**	**CLBs**	**CLBs** per CLU	**Min CLK** in ns	**Data Rate** per CLU in Mbit/s	**Data Rate** in Mbit/s
DES_ED16	XC4008E-3-PG223	262	262	40.4	94.5	**94.5**
DES_ED16x2	XC4013E-3-PG223	433	217	43.5	87.7	**175.3**
DES_ED16x4	XC4028EX-3-PG299	741	185	39.7	96.0	**384.0**

Table 8.2: Comparison of pipelined architectures

The amount of logic resources consumed per implemented CLB is decreasing if we create more pipelines. This is due to the fact that the control unit does not get more complicated if we implement more pipelines. Also the multiplexers are implemented only once.

It is interesting to compare the pipelined designs with the loop unrolled designs. It can be seen that *DES_ED16x2* is both faster and smaller than the loop unrolled *DES_ED8*. The difference is even more dramatically if the *DES_ED16x4* is compared with the *DES_ED4*. *DES_ED16* is more than twice as fast as *DES_ED4* and utilizes almost the same amount of CLBs.

8.3 Combination of Pipelining and Loop Unrolling

A design that contains loop unrolling as well as pipelining is in the simplest version already as large as the largest designs we have implemented so far which were *DES_ED16x4* and *DES_ED4*. Therefore we implemented only the design *DES_ED8x2* which contains 4 CLUs; 2 in each of the 2 pipelines. Table 8.3 compares this design with *DES_ED16x2* and *DES_ED8*.

It is not easy to compare this mixed design with the two other designs. The minimum clock period shows that the time it takes for two CLUs (loop unrolled) to execute in the design *DES_ED8x2* is faster than in the design *DES_ED8*. It is of course slower, but surprisingly not much, than one CLU in the design *DES_16x2*.

Design	**Chip**	**CLBs**	**Min CLK**	**Data Rate**	**Data Rate**
			in ns	p. pipeline in Mbit/s	in Mbit/s
DES_ED8x2	XC4028EX-3-PG299	733	48.0	158.8	**317.6**
DES_ED16x2	XC4013E-3-PG223	433	43.5	87.7	**175.3**
DES_ED8	XC4013E-3-PG223	443	54.0	141.3	**141.3**

Table 8.3: Comparison of a combined architecture with others

8.4 Chip Dependencies

During implementation of our designs we experienced that the result of an implementation is depending on the chip parameters. These are investigated further here.

8.4.1 Chip Sizes

We implemented the design *DES_ED16* on chips of three different sizes. Table 8.4 compares these implementations. The size of a chip is measured in number of CLBs.

Design	**Chip**	**CLBs**	**CLBs**	**Min CLK**	**Data Rate**
		on Chip	used	in ns	in Mbit/s
DES_ED16	XC4008E-3-PG223	324	262	40.4	**94.5**
DES_ED16	XC4013E-3-PG223	576	262	41.8	**91.2**
DES_ED16	XC4025E-3-PG223	1024	262	45.5	**83.9**

Table 8.4: Comparison of different chip sizes

The interesting result is that the bigger a chip is, the slower the design gets. Even though a bigger chip provides more logic and routing resources, and the place and route tool has an easier job of optimizing, the time it takes for data to propagate through the chip is longer. The floor plans of the `XC4025E-3-PG223` and `XC4008E-3-PG223` can be found in Appendix C.

8.4.2 Speed Grades

The speed grade is defined by Xilinx as the time it takes for a signal to propagate through one combinatorial level (see [15]). We implemented the design *DES_MQP* for three different speed grades: *-1*, *-2*, and *-3*.

The Xilinx *Timing Analyzer* has a feature that enables the user to calculate the minimum clock period for any selected speed grade based on a placed and routed design. These results are unfortunately not comparable to the results we go when we synthesized and placed and routed a design from scratch for a new speed grade. The results presented in Table 8.5 are generated using the later approach.

Design	**Chip**	**Data Rate Grade**	**CLBs**	**Min CLK** in ns	**Data Rate** in Mbit/s
DES_MQP	XC4013E-3-PG223	-3	294	40.9	**93.3**
DES_MQP	XC4013E-2-PG223	-2	294	36.5	**104.6**
DES_MQP	XC4013E-1-PG223	-1	294	29.3	**130.1**

Table 8.5: Comparison of different speed grades

The change of speed grades from *-3* to *-2* resulted in a 10% performance increase. The change from *-2* to *-1* resulted in a further performance increase of 24%.

8.4.3 Device Families

The `XC4000EX` series offers almost twice the routing capacity of the `XC4000E` series (see [14]). As seen in Section 8.2 the routing resources can influence the performance of the design. To examine this further we implemented the design *DES_ED16x4* on the devices `XC4025E-3-PG223` and `XC4028EX-3-PG299`. Table 8.6 compares the two implementations.

This comparison shows clearly the influence of the wiring resources on the performance of the design. The implementation on the `XC4000EX` family device is more

Design	**Chip**	**Chip Family**	**CLBs**	**Min CLK** in ns	**Data Rate** in Mbit/s
DES_ED16x4	XC4025E-3-PG223	XC4000E	741	61.5	**248.3**
DES_ED16x4	XC4028EX-3-PG299	XC4000EX	741	39.7	**384.0**

Table 8.6: Comparison of different chip families

than 54% faster for our largest design. It is to be noted that both devices provide the same amount of logic resources (CLBs).

Even tough the design *DES_ED16x4* is our largest design, it is not the most routing intensive. The most routing intensive design is *DES_ED4*; the Xilinx tools were not able to place and route this design in the `XC4025E-3-PG223` device.

8.5 Summary and Overview

Table 8.7 summarizes the results of all the implemented designs. Our fastest implementation with loop unrolling is *DES_ED4* with **176.0 Mbit/sec**, the fastest employing pipelines is *DES_ED16x4* with **384.0 Mbit/sec**.

Design	**Chip**	**CLBs**	**CLBs** per CLU	**Min CLK** in ns	**Data Rate** per CLU in Mbit/s	**Data Rate** in Mbit/s
DES16, v1.1	XC4013E-3-PG223	200	200	43.4	88.0	**88.0**
DES16, v1.2	XC4013E-3-PG223	198	198	41.8	91.3	**91.3**
DES_MQP	XC4013E-3-PG223	294	294	40.9	93.3	**93.3**
DES_MQP	XC4013E-2-PG223	294	294	36.5	104.6	**104.6**
DES_MQP	XC4013E-1-PG223	294	294	29.3	130.1	**130.1**
DES_ED16	XC4013E-3-PG223	262	262	41.8	91.2	**91.2**
DES_ED16	XC4025E-3-PG223	262	262	45.5	83.9	**83.9**
DES_ED16	XC4008E-3-PG223	262	262	40.4	94.5	**94.5**
DES_ED8	XC4013E-3-PG223	443	222	54.0	70.6	**141.3**
DES_ED4	XC4028EX-3-PG299	722	241	86.7	44.0	**176.0**
DES_ED16x2	XC4013E-3-PG223	433	217	43.5	87.7	**175.3**
DES_ED16x4	XC4028EX-3-PG299	741	185	39.7	96.0	**384.0**
DES_ED16x4	XC4025E-3-PG223	741	185	61.5	62.1	**248.3**
DES_ED8x2	XC4028EX-3-PG299	733	184	48.0	79.4	**317.6**

Table 8.7: Complete table of all implemented architectures

Chapter 9

Conclusion

This chapter concludes the thesis. It lists some recommendations for the design of DES on FPGAs and presents a summary of the results. Finally some recommendations for future work are given.

9.1 Design Recommendations

During our research and the implementation phase of the designs, we formulated some recommendations for an efficient DES design on Xilinx FPGAs.

- *S-Boxes* should be implemented in ROM for maximum performance; a fast implementation of the S-Boxes is crucial for the over-all performance of the design.
- *Permutations and expansions* are implemented using only wiring resources.
- *Shift registers* can be implemented using only wiring resources, or for decisive and directional shifters a multiplexer.
- *LogiBLOX* ease the design entry and are already well optimized.

We could also show that our technique of *one-round sub-key precomputation* results in a faster design and enables us to generate sub-keys for encryption and decryption at no performance penalty (as opposed to just generate sub-keys for encryption).

The split up of the design into small basic function blocks simplified design modifications. In order to create a new architecture we had to modify only some files. Each new architecture needed a new control logic.

9.2 Summary of Results

We implemented all designs based on devices from Xilinx (see Section 7.1). Here are our most important findings.

- **Maximum speed:** We achieved speeds of up to 384.0 Mbit/sec.
- **Performance Comparison:** If we compare the reported DES speeds for ASICs (1600 Mbit/sec) [12] and Software (12 Mbit/sec) [12], with our best result of 384.0 Mbit/sec we conclude that the speed-up factor from software to FPGAs is 32.0, and from FPGAs to ASICs is 4.3.

We explored the architecture options *loop unrolling* and *pipelining* in detail for FPGAs. Here are our most important results.

- **Loop unrolling:** With the first unrolling we gained 50% higher encryption rate and used 69% more logic resources; with the second unrolling we gained only 25% speed over the first unrolling and used 63% more logic resources. **Conclusion:** the amount of logic resources consumed rises linearly, whereas the speed increases much slower.
- **Pipelining:** With two pipelines we gained 86% more speed at the expense of 65% more logic resources; with four pipelines we gained 120% more speed

over two pipelines and used 71% more logic resources. This speed-up is a little distorted due to the limited amount of wiring resources on the chip we implemented the two pipeline design. **Conclusion:** the amount of logic resources consumed rises linearly and the speed too.

- **Combined Design:** Results in a fast overall design. The result is a mixture of both base designs this is comprised of.

Loop unrolling does not result in the highest speeds but it is can be used in any mode of operation. Pipelined designs are faster but can only be used in modes which are not based on a feedback of the result of DES or a derivation therefrom. A pipelined design can therefore only be used in ciphers that employ ECB or counter mode (e.g., Counter Mode specified for ATM-networks). This holds also for the combined design as it contains a pipeline.

If the pipelines are demultiplexed external to the FPGA a pipelined design comprising two pipelines could be used as two separate DES chips, and then every mode of operation is possible within each pipeline.

The influence of Xilinx chip parameters is summarized below:

- **Chip size:** A bigger chip results in a slower design.
- **Speed grades:** A migration from a speed grade *-3* to *-2* results in a 10% higher performance.
- **Device family:** The amount of routing resources on the chip is crucial for the implementation.

9.3 Recommendations for Future Work

For this thesis we implemented DES just in ECB mode. It can be used in other modes as well but at the expense of additional external hardware. It would be very interesting to explore the issues involved in enhancing this design so that it supports all modes defined for DES within the same FPGA and its final speed.

Future work will also investigate applications for the designs presented here. Interesting areas would be ATM-encrypters and key-search machines. A natural application area for our design would be encryption modules that provide algorithm agility, i.e., encryption algorithm switch on-the-fly. A possible system might be a PC plug-in board with fast bus interface which supports a variety of encryption schemes.

Appendix A

Simulation Script Files

RTL-level simulation requires that the used libraries are analyzed first, then all VHDL source files and behavioural description of the LogiBLOX, and at the end the test bench.

For past place and route simulation also the libraries have to be analyzed first, then the `time_sim.vhd` file which comprises the whole back-annotated design, and at the end the test bench.

A.1 RTL-Level Simulation Script

```
# -----------------------------------------------------------------
# everything to get ready for the rtl-level simulation
#
# Jens-Peter Kaps                               February 23rd, 1998
#
# $Log: make_rtl_sim,v $
# Revision 2.1  1998/02/25  03:09:18  kaps
# updated for encrypt / decrypt des
#
# Revision 1.1  1998/02/23  05:05:17  kaps
# Initial revision
#
# -----------------------------------------------------------------
vhdlan -i ./rtl_sim/mvlutil.vhd \
          ./rtl_sim/mvlarith.vhd \
          ./rtl_sim/logiblox.vhd \
          ./logiblox/sox1.vhd \
          ./logiblox/sox2.vhd \
          ./logiblox/sox3.vhd \
          ./logiblox/sox4.vhd \
          ./logiblox/sox5.vhd \
          ./logiblox/sox6.vhd \
          ./logiblox/sox7.vhd \
          ./logiblox/sox8.vhd \
          ./logiblox/reg32c.vhd \
          ./logiblox/reg16c.vhd \
          ./logiblox/reg8c.vhd \
          ./logiblox/mux32l.vhd \
          ./logiblox/mux16l.vhd \
          ./logiblox/mux8l.vhd \
          ./src/sboxes.vhd \
          ./src/ebox.vhd \
          ./src/ipinv.vhd \
          ./src/ipnorm.vhd \
          ./src/la_rot.vhd \
          ./src/lm_rot.vhd \
          ./src/ra_rot.vhd \
          ./src/rm_rot.vhd \
          ./src/mux56.vhd \
          ./src/mux64.vhd \
          ./src/pbox.vhd \
          ./src/pc1box.vhd \
          ./src/pc2box.vhd \
          ./src/reg56.vhd \
          ./src/reg64.vhd \
          ./src/xormod.vhd \
          ./src/xormod48.vhd \
          ./src/ffunc.vhd \
          ./src/feistel.vhd \
          ./src/keygen.vhd \
          ./src/control.vhd \
          ./src/des.vhd \
          ./rtl_sim/testbench.vhd
```

A.2 Post Place and Route Simulation Script

```
vhdlan -i ./ppr_sim/simprim_Vcomponents.vhd \
          ./ppr_sim/simprim_Vpackage.vhd \
          ./ppr_sim/simprim_VITAL.vhd \
          ./time_sim.vhd \
          ./rtl_sim/testbench.vhd
```

```
# afterwards invoke the simulator with the following command line:
#
#   vhdldbx -sdf_top testbench/uut -sdf time_sim.sdf CFG_TB &
```

Appendix B

Synthesis Script

This is the script file for Synopsys to synthesis the design *DES_ED16* for the device XC4013E-3-PG223.

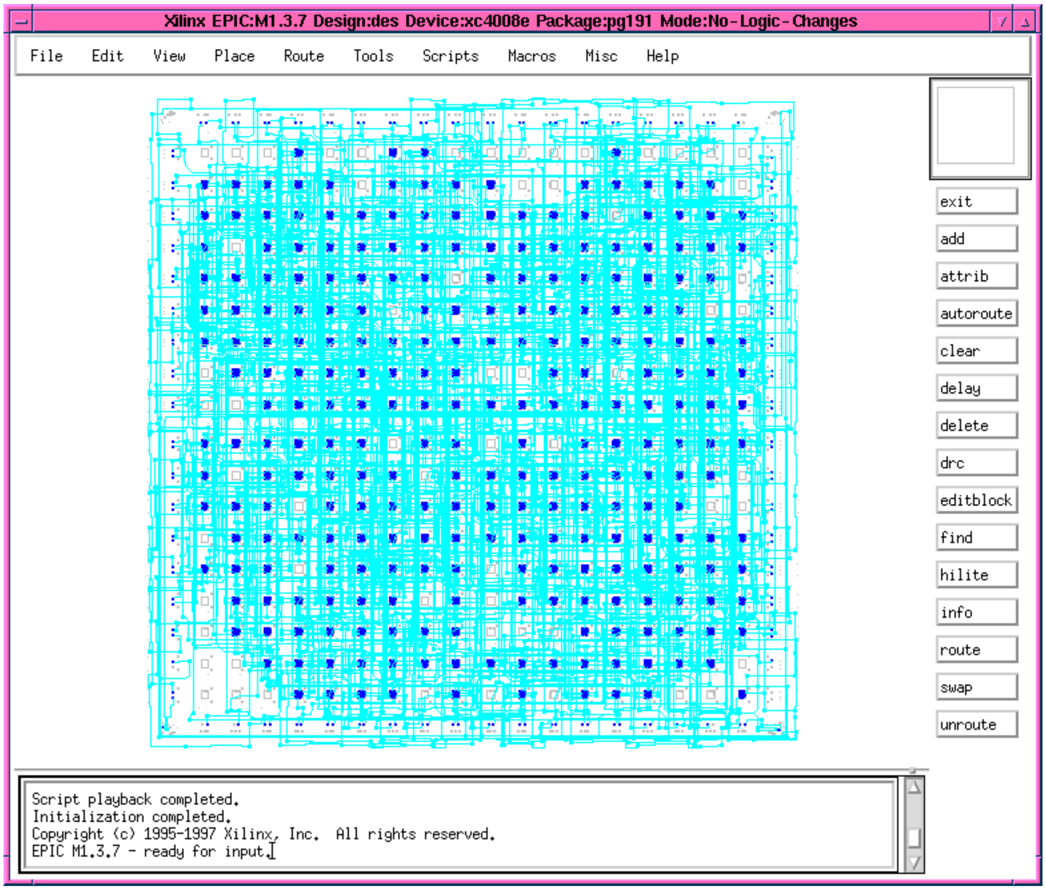
```
/* ------------------------------------------------------------ */
/* Script file for Synopsys FPGA Compiler                       */
/* targeting a XC4013E device using Logiblox for the S-Boxes    */
/* ------------------------------------------------------------ */
/* $Log $
 */

/* ------------------------------------------------------------ */
/* Defining the Paths                                           */
/* ------------------------------------------------------------ */
    SRC_PATH    = "src/"
    DB_PATH     = "db/"
    DC_PATH     = "dc/"
    REPORT_PATH = "reports/"
    SXNF_PATH   = "sxnf/"
    LOGI_PATH   = "logiblox/"

/* ------------------------------------------------------------ */
/* Defining the Logiblox Elements No Need but.....              */
/* ------------------------------------------------------------ */
    SBOX1       = sox1
    SBOX2       = sox2
    SBOX3       = sox3
    SBOX4       = sox4
    SBOX5       = sox5
    SBOX6       = sox6
    SBOX7       = sox7
    SBOX8       = sox8
    REG8C       = reg8c
    REG16C      = reg16c
    REG32C      = reg32c
    MUX8L       = MUX8L
    MUX16L      = MUX16L
    MUX32L      = MUX32L

/* --------------------------------------------------------------- */
/* Name for the design's top-level and other                       */
/* --------------------------------------------------------------- */
    TOP         = des
    MODULS      = module_pack

    CONTROL     = control
    KEY1GEN     = key1gen
    FEISTEL     = feistel
    FFUNC       = ffunc
/* --------------------------------------------------------------- */
/* Name for the design's modules containing Loginlox               */
/* --------------------------------------------------------------- */

    MUX56       = mux56
    MUX64       = mux64
    REG56       = reg56
    REG64       = reg64
    SBOXES      = sboxes

/* --------------------------------------------------------------- */
/* Low level modules (don't contain other modules)                 */
/* --------------------------------------------------------------- */
    EBOX        = ebox
    IPINV       = ipinv
    IPNORM      = ipnorm
    LAROT       = la_rot
    LMROT       = lm_rot
    RMROT       = rm_rot
    PBOX        = pbox
    PC1BOX      = pc1box
    PC2BOX      = pc2box
    XORMOD      = xormod
    XORMOD48    = xormod48

/* --------------------------------------------------------------- */
/* Design Group and Part Number                                    */
/* --------------------------------------------------------------- */
    designer    = "Jens-Peter Kaps"
    company     = "WPI Crypto Group"
    part        = "4013EPG223-3"
```

```
/* ----------------------------------------------------------------- */
/* Analyze the Module Package                                        */
/* ----------------------------------------------------------------- */
    analyze -f vhdl -lib WORK SRC_PATH + MODULS   + ".vhd"

/* ----------------------------------------------------------------- */
/* Analyze and elaborate the low level files first                   */
/* ----------------------------------------------------------------- */
    analyze -f vhdl -lib WORK SRC_PATH + EBOX     + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + IPINV    + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + IPNORM   + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + LAROT    + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + LMROT    + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + RMROT    + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + PBOX     + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + PC1BOX   + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + PC2BOX   + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + XORMOD   + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + XORMOD48 + ".vhd"

    elaborate EBOX
    elaborate IPINV
    elaborate IPNORM
    elaborate LAROT
    elaborate LMROT
    elaborate RMROT
    elaborate PBOX
    elaborate PC1BOX
    elaborate PC2BOX
    elaborate XORMOD
    elaborate XORMOD48

/* ----------------------------------------------------------------- */
/* Analyze and elaborate the design files containing Logiblox        */
/* ----------------------------------------------------------------- */
/* ----------------------------------------------------------------- */
/* SBOXES                                                            */
/* ----------------------------------------------------------------- */
    analyze -f vhdl -lib WORK SRC_PATH + SBOXES   + ".vhd"

    elaborate SBOXES

/* ----------------------------------------------------------------- */
/* set don’t touch on LogiBLOX                                       */
/* ----------------------------------------------------------------- */
    set_dont_touch find(cell, "MY_SBOX1")
    set_dont_touch find(cell, "MY_SBOX2")
    set_dont_touch find(cell, "MY_SBOX3")
    set_dont_touch find(cell, "MY_SBOX4")
    set_dont_touch find(cell, "MY_SBOX5")
    set_dont_touch find(cell, "MY_SBOX6")
    set_dont_touch find(cell, "MY_SBOX7")
    set_dont_touch find(cell, "MY_SBOX8")

/* --------------------------------------------------------------- */
/* REG64                                                           */
/* --------------------------------------------------------------- */
    analyze -f vhdl -lib WORK SRC_PATH + REG64    + ".vhd"

    elaborate REG64

/* --------------------------------------------------------------- */
/* set don't touch on LogiBLOX                                     */
/* --------------------------------------------------------------- */
    set_dont_touch find(cell, "LEFT_REG")
    set_dont_touch find(cell, "RIGHT_REG")

/* --------------------------------------------------------------- */
/* REG56                                                           */
/* --------------------------------------------------------------- */
    analyze -f vhdl -lib WORK SRC_PATH + REG56    + ".vhd"

    elaborate REG56

/* --------------------------------------------------------------- */
/* set don't touch on LogiBLOX                                     */
/* --------------------------------------------------------------- */
    set_dont_touch find(cell, "BUF_8")
    set_dont_touch find(cell, "BUF_16")
    set_dont_touch find(cell, "BUF_32")

/* --------------------------------------------------------------- */
/* MUX64                                                           */
/* --------------------------------------------------------------- */
    analyze -f vhdl -lib WORK SRC_PATH + MUX64    + ".vhd"

    elaborate MUX64

/* --------------------------------------------------------------- */
/* set don't touch on LogiBLOX                                     */
/* --------------------------------------------------------------- */
    set_dont_touch find(cell, "LEFT_MUX")
    set_dont_touch find(cell, "RIGHT_MUX")

/* --------------------------------------------------------------- */
/* MUX56                                                           */
/* --------------------------------------------------------------- */
```

```
    analyze -f vhdl -lib WORK SRC_PATH + MUX56    + ".vhd"

    elaborate MUX56

/* ----------------------------------------------------------------- */
/* set don't touch on LogiBLOX                                        */
/* ----------------------------------------------------------------- */
    set_dont_touch find(cell, "MY_MUX_8")
    set_dont_touch find(cell, "MY_MUX_16")
    set_dont_touch find(cell, "MY_MUX_32")

/* ----------------------------------------------------------------- */
/* Analyze and elaborate some more design files                      */
/* ----------------------------------------------------------------- */
    analyze -f vhdl -lib WORK SRC_PATH + FFUNC     + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + FEISTEL   + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + KEY1GEN   + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + CONTROL   + ".vhd"
    analyze -f vhdl -lib WORK SRC_PATH + TOP       + ".vhd"

    elaborate FFUNC
    elaborate FEISTEL
    elaborate KEY1GEN
    elaborate CONTROL
    elaborate TOP

/* ----------------------------------------------------------------- */
/* Set the current design to the top level.                          */
/* ----------------------------------------------------------------- */
    current_design TOP

/* ----------------------------------------------------------------- */
/* Set the synthesis design constraints                              */
/* ----------------------------------------------------------------- */
    remove_constraint -all


/* ----------------------------------------------------------------- */
/* set don't touch on Startup Block                                  */
/* ----------------------------------------------------------------- */
    set_dont_touch {STARTUPBLK}

/* ----------------------------------------------------------------- */
/* uniquify multiple instances of designs                            */
/* ----------------------------------------------------------------- */
    uniquify

/* ----------------------------------------------------------------- */
/* include timing and timing constraints                             */
/* ---------------------------------------------------------------- */
    create_clock clk -period 40
    set_input_delay 5 -clock clk { all_inputs()}
    set_output_delay 5 -clock clk { all_outputs()}
    set_wire_load "4013e-3_avg"
    set_operating_conditions WCCOM

/* ---------------------------------------------------------------- */
/* Indicate top-level module ports that shoud become i/o pads      */
/* ---------------------------------------------------------------- */
    set_port_is_pad "*"
    set_pad_type -clock clk
    set_pad_type -slewrate HIGH all_outputs()
    insert_pads

/* ---------------------------------------------------------------- */
/* Synthesize and optimize the design.                              */
/* ---------------------------------------------------------------- */
    compile -boundary_optimization

/* ---------------------------------------------------------------- */
/* Write the design report files.                                   */
/* ---------------------------------------------------------------- */
    report_fpga > REPORT_PATH + TOP + ".fpga"
    report_timing > REPORT_PATH + TOP + ".timing"

/* ---------------------------------------------------------------- */
/* Write out an intermediate DB file to save state                  */
/* ---------------------------------------------------------------- */
    write -format db -hierarchy -output DB_PATH + TOP + "_compiled.db"

/* ---------------------------------------------------------------- */
/* Replace CLBs and IOBs primitives (XC4000E/EX/XL only)            */
/* ---------------------------------------------------------------- */
    replace_fpga

/* ---------------------------------------------------------------- */
/* Set the part type for the output netlist.                        */
/* ---------------------------------------------------------------- */
    set_attribute TOP "part" -type string part

/* ---------------------------------------------------------------- */
/* Write out an intermediate DB file to save state                  */
/* ---------------------------------------------------------------- */
    write -format db -hierarchy -output DB_PATH + TOP + ".db"

/* ---------------------------------------------------------------- */
/* Write-out the timing constraints that were applied earlier.      */
/* And flatten the hierarchy                                        */
/* --------------------------------------------------------------- */
    ungroup -all -flatten
    write_script > DC_PATH + TOP + ".dc"

/* Save design in XNF format as <design>.sxnf      */
    write -f xnf -h -o SXNF_PATH + TOP + ".sxnf"

/* --------------------------------------------------------------- */
/* Call synopsys to Xilinx contraints translator DC2NCF            */
/* --------------------------------------------------------------- */
    sh dc2ncf DC_PATH + TOP + ".dc"
```

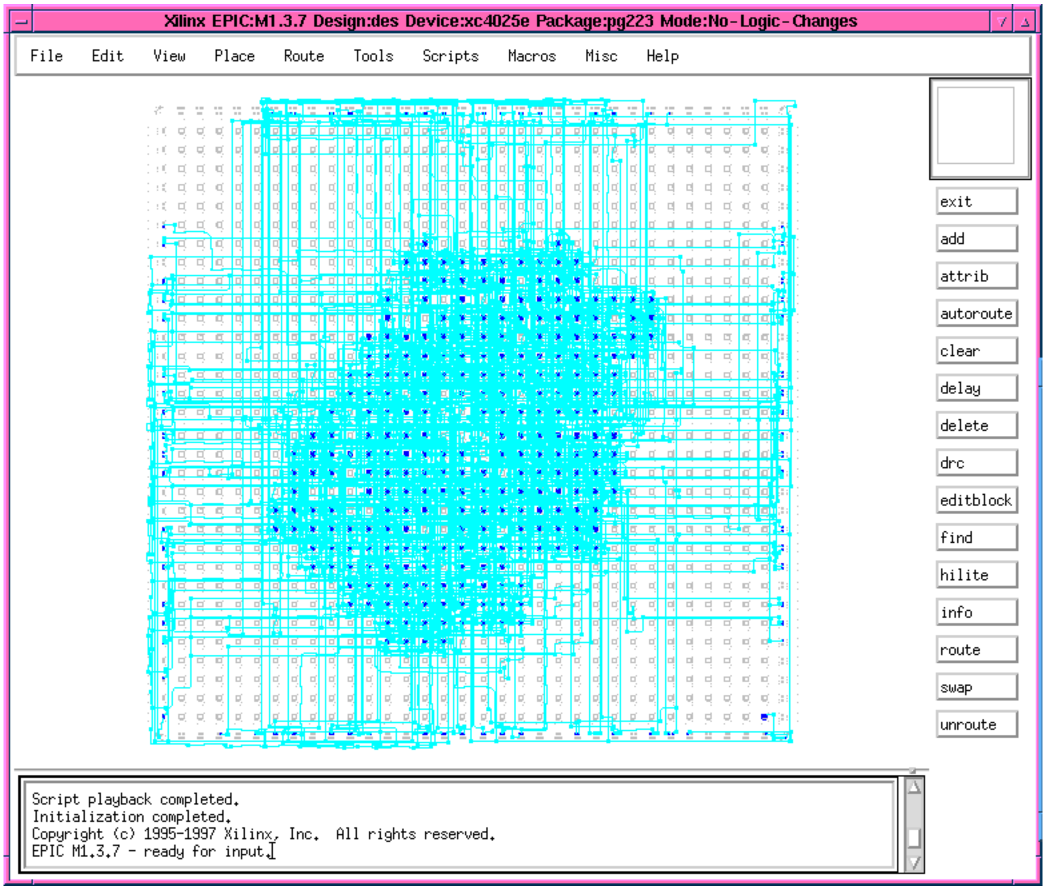
Appendix C

Floor Plans



```
Script playback completed.
Initialization completed.
Copyright (c) 1995-1997 Xilinx, Inc.  All rights reserved.
EPIC M1.3.7 - ready for input.
```

Figure C.1: Floor Plan of *DES_ED16* on the Chip `4008E-3-PG191`



```
Script playback completed.
Initialization completed.
Copyright (c) 1995-1997 Xilinx, Inc.  All rights reserved.
EPIC M1.3.7 - ready for input.
```

Figure C.2: Floor Plan of *DES_ED16* on the Chip `4025E-3-PG223`

Appendix D

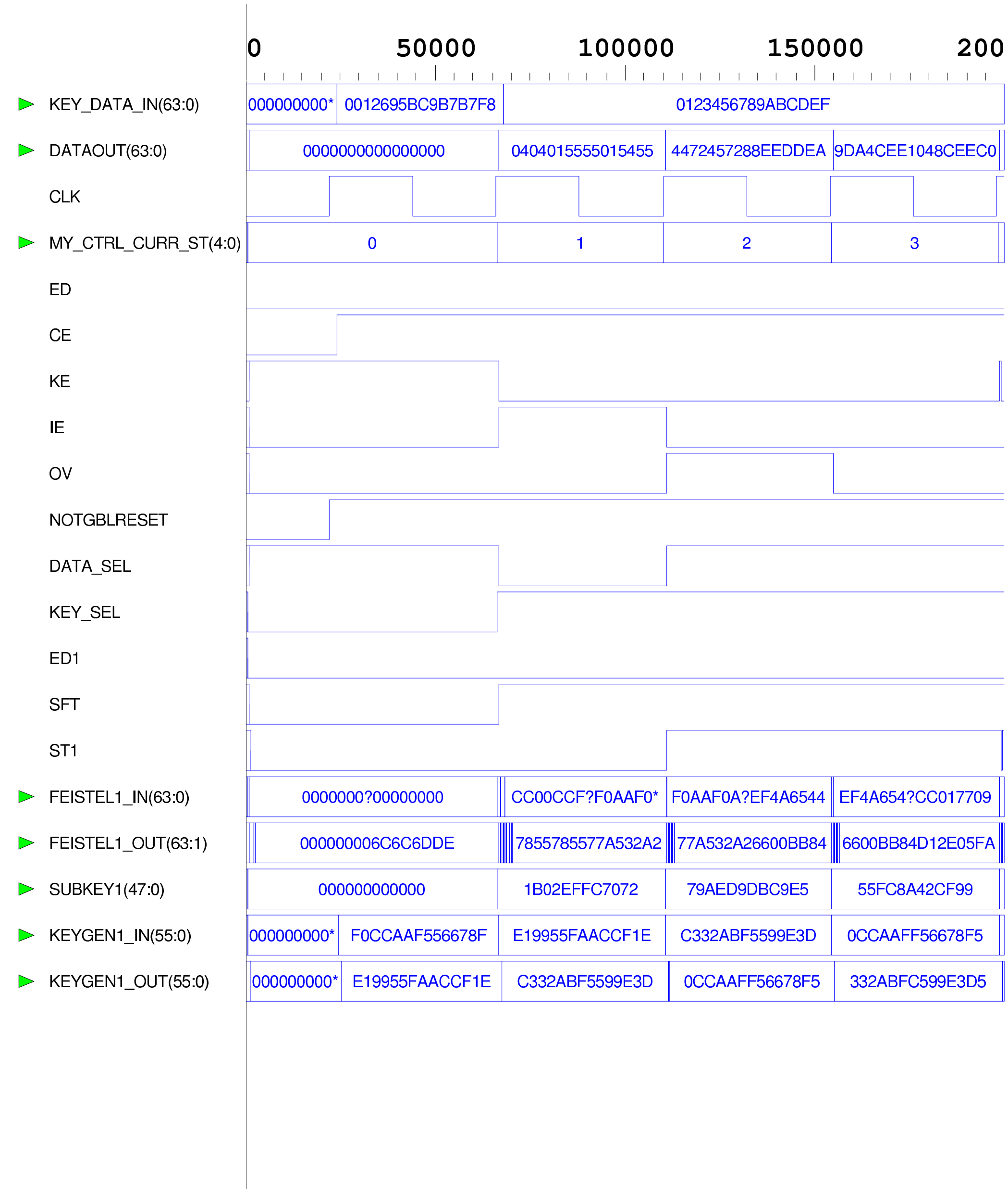
Timing Diagrams

This appendix shows the timing diagram of one full encryption in Appendix D.1 and one full decryption in Appendix D.2. These timing diagrams are *past place and route* and therefore show the actuall delays.

The clock period is set to 44ns. The scale on top of the diagrams is in pico seconds. During state *0* the key gets loaded and during stage *1* the data. Data and key are provided on the `KEY_DATA_IN(63:0)` bus. The result of the operation appears on the `DATAOUT(63:0)` bus during the first stage of the next operation.

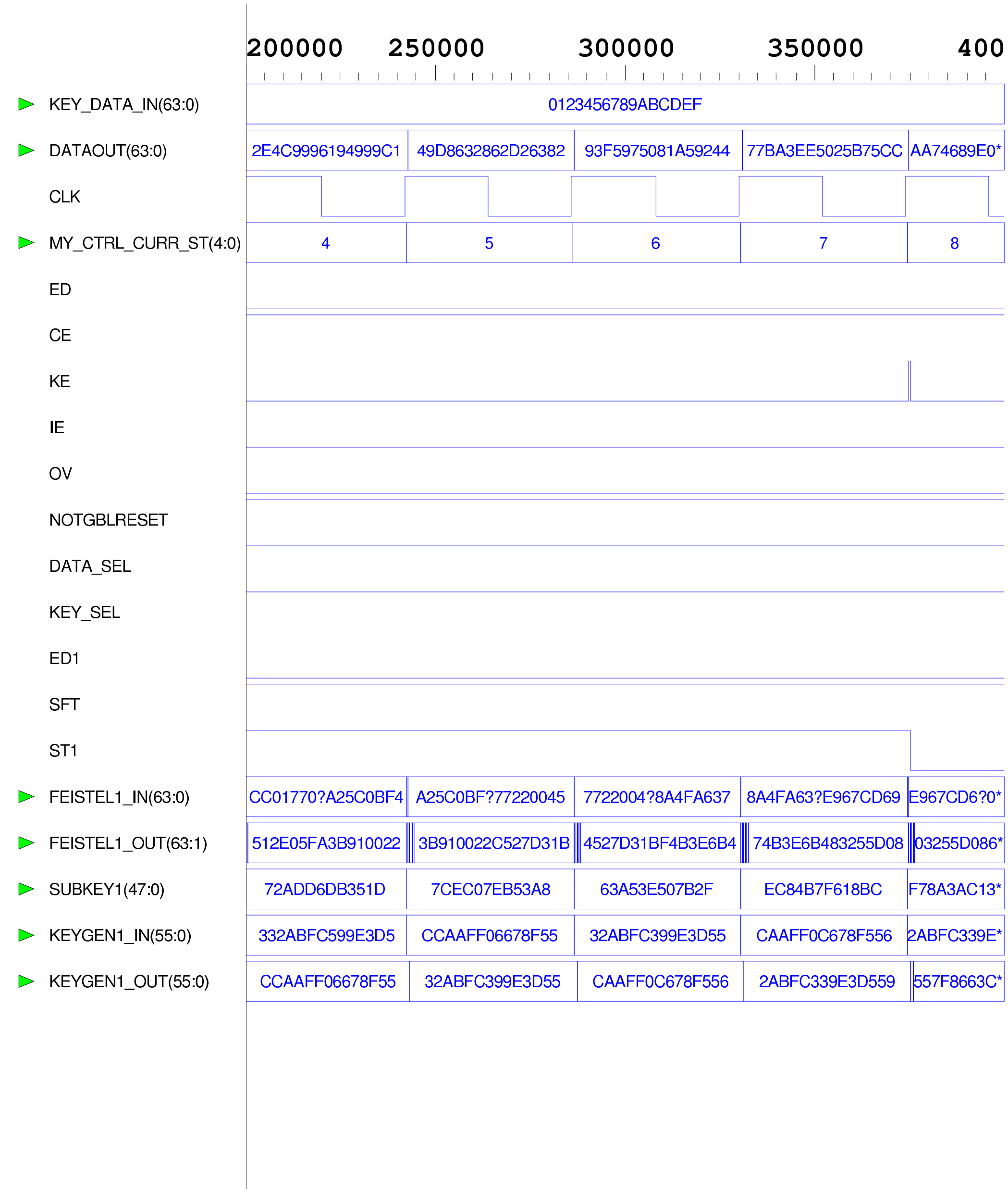
The test bench used to test the design and generate these diagrams is in Appendix E.

D.1 Encryption



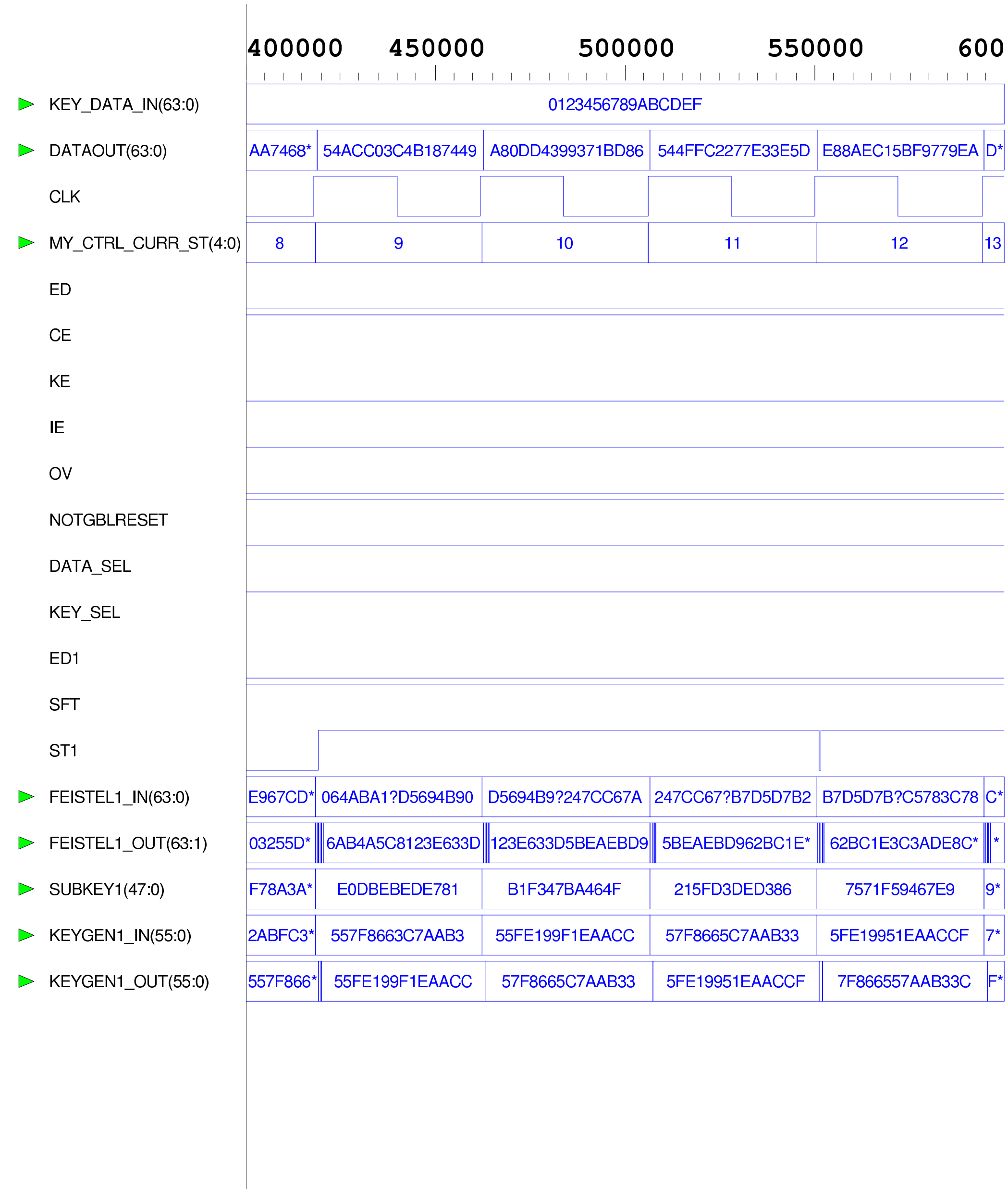
```
   /usr3/kaps/tw/vhdl/development/E.pike.WPI.EDU.21188.ow
19/4/1998      2:19:55                          Page 1,1 of 1,1
```

Figure D.1: Encryption with *DES_ED16* on the Chip 4008E-3-PG191



/usr3/kaps/tw/vhdl/development/E.pike.WPI.EDU.21188.ow
19/4/1998 2:21:39 Page 1,1 of 1,1

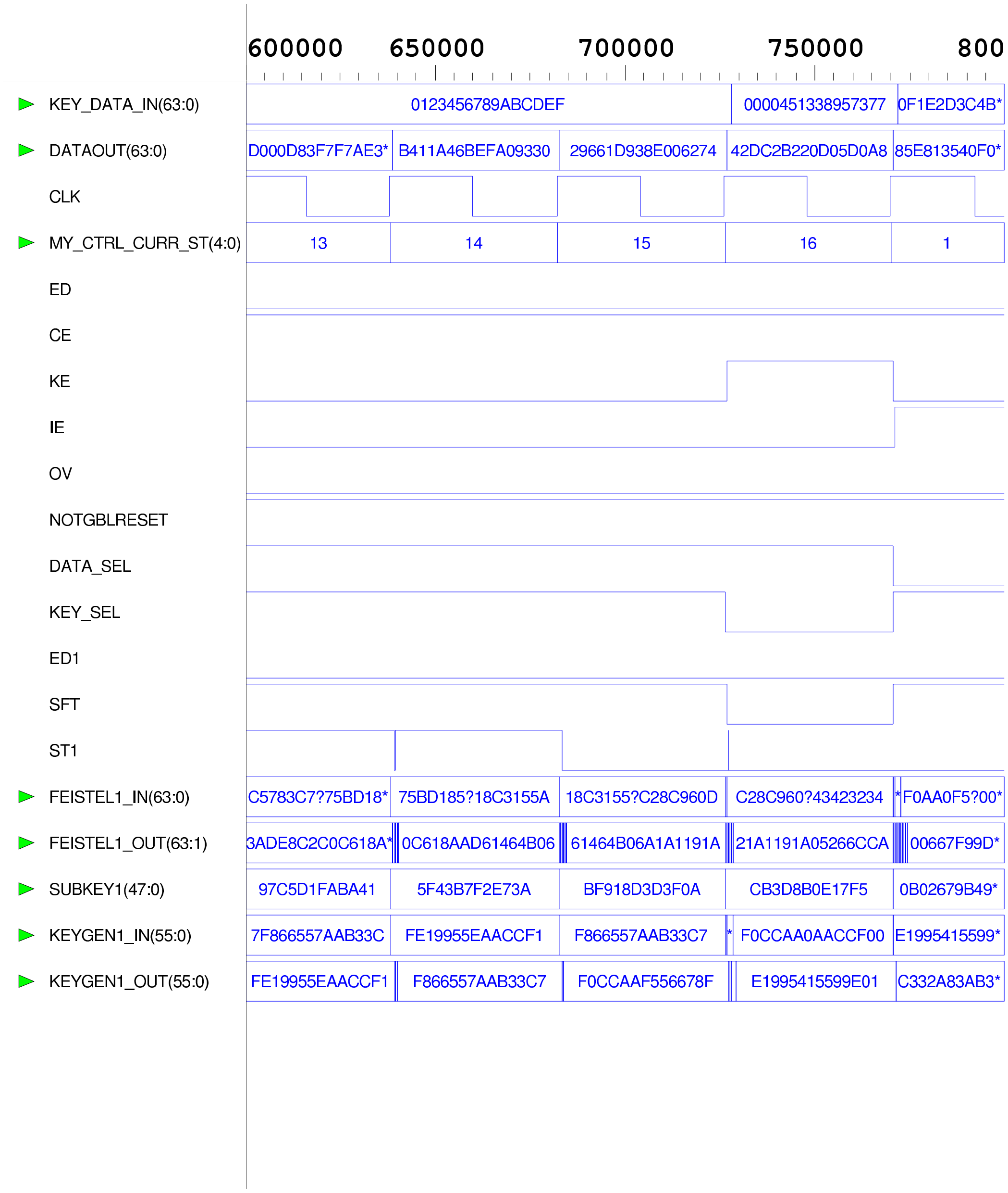
Figure D.2: Encryption with *DES_ED16* on the Chip 4008E-3-PG191



/usr3/kaps/tw/vhdl/development/E.pike.WPI.EDU.21188.ow

19/4/1998 2:22:36 Page 1,1 of 1,1

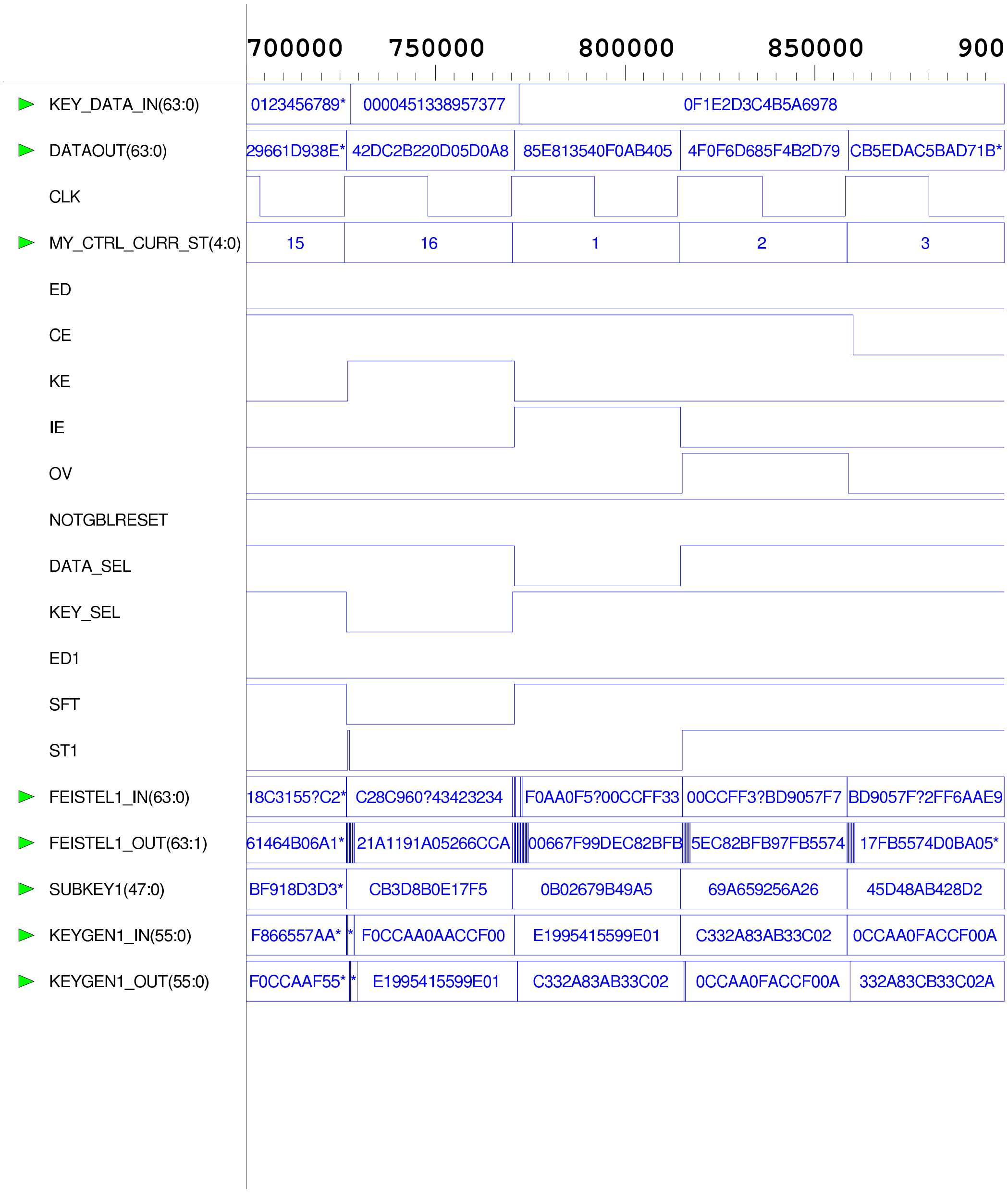
Figure D.3: Encryption with *DES_ED16* on the Chip 4008E-3-PG191



/usr3/kaps/tw/vhdl/development/E.pike.WPI.EDU.21188.ow

19/4/1998 2:31:24 Page 1,1 of 1,1

Figure D.4: Encryption with *DES_ED16* on the Chip 4008E-3-PG191

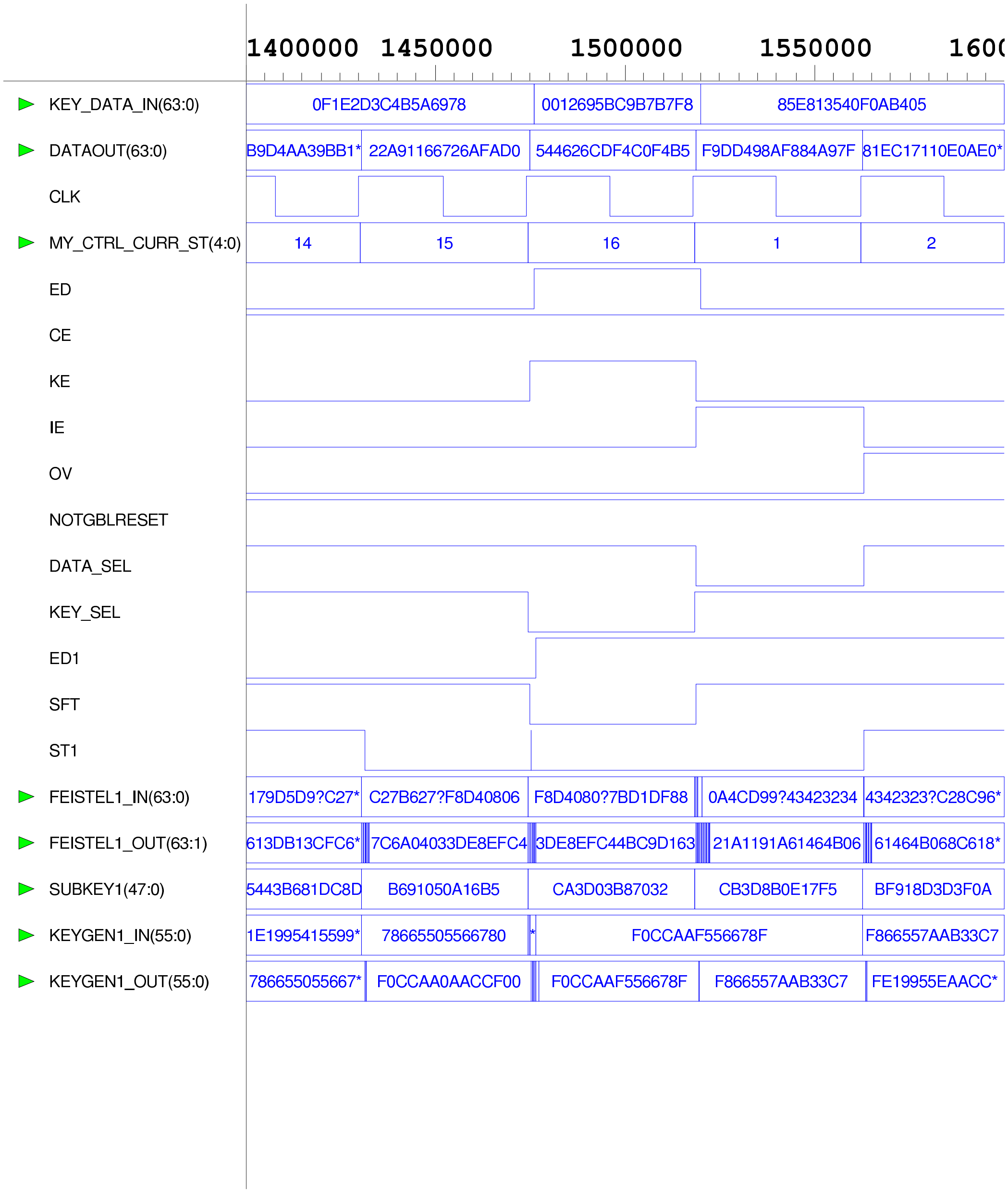


/usr3/kaps/tw/vhdl/development/E.pike.WPI.EDU.21188.ow

19/4/1998 2:32:6 Page 1,1 of 1,1

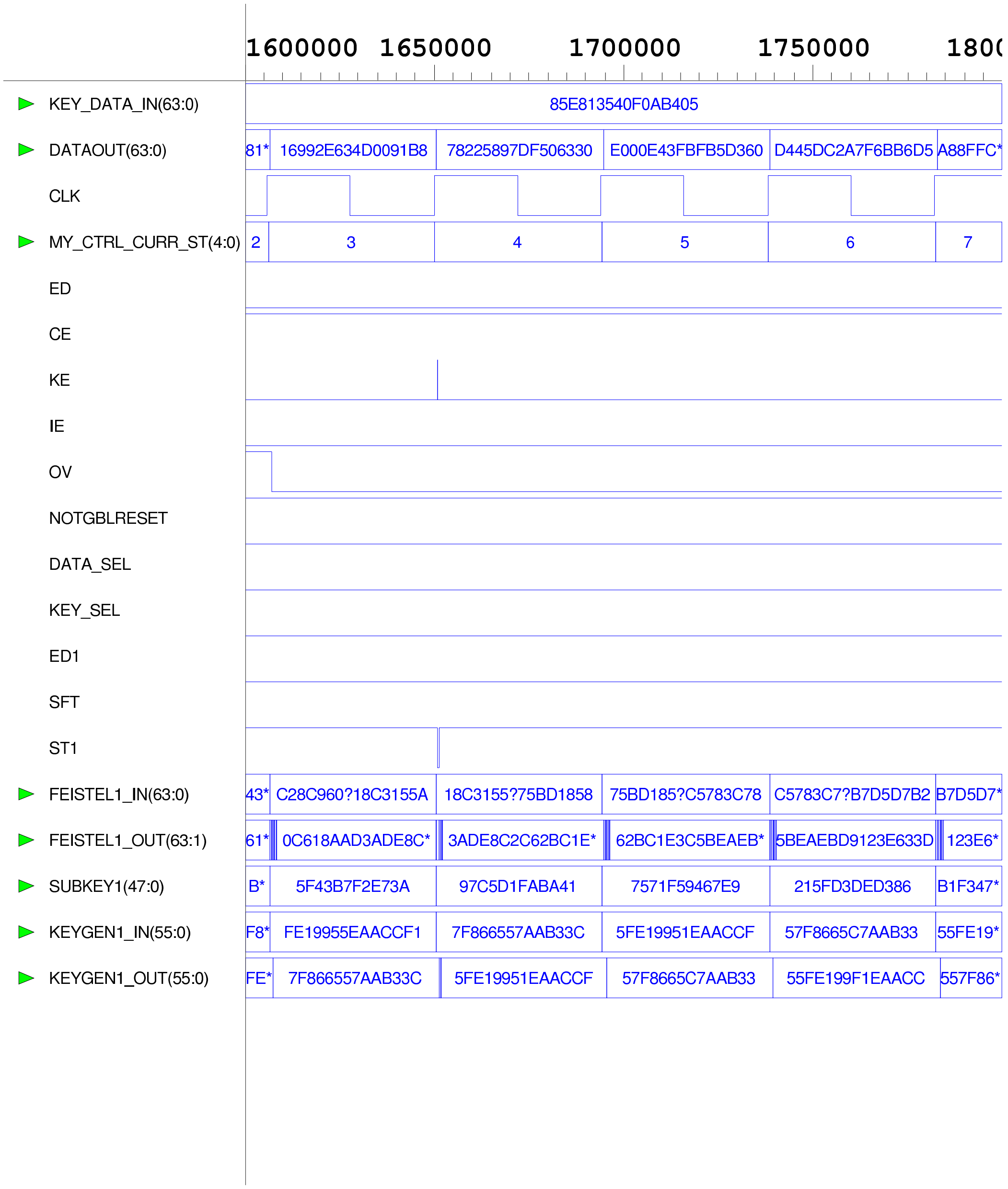
Figure D.5: Enrcyption with *DES_ED16* on the Chip 4008E-3-PG191

D.2 Encryption



/usr3/kaps/tw/vhdl/development/E.pike.WPI.EDU.21188.ow
19/4/1998 4:41:1 Page 1,1 of 1,1

Figure D.6: Decryption with *DES_ED16* on the Chip 4008E-3-PG191



/usr3/kaps/tw/vhdl/development/E.pike.WPI.EDU.21188.ow

19/4/1998 4:43:29 Page 1,1 of 1,1

Figure D.7: Decryption with *DES_ED16* on the Chip `4008E-3-PG191`

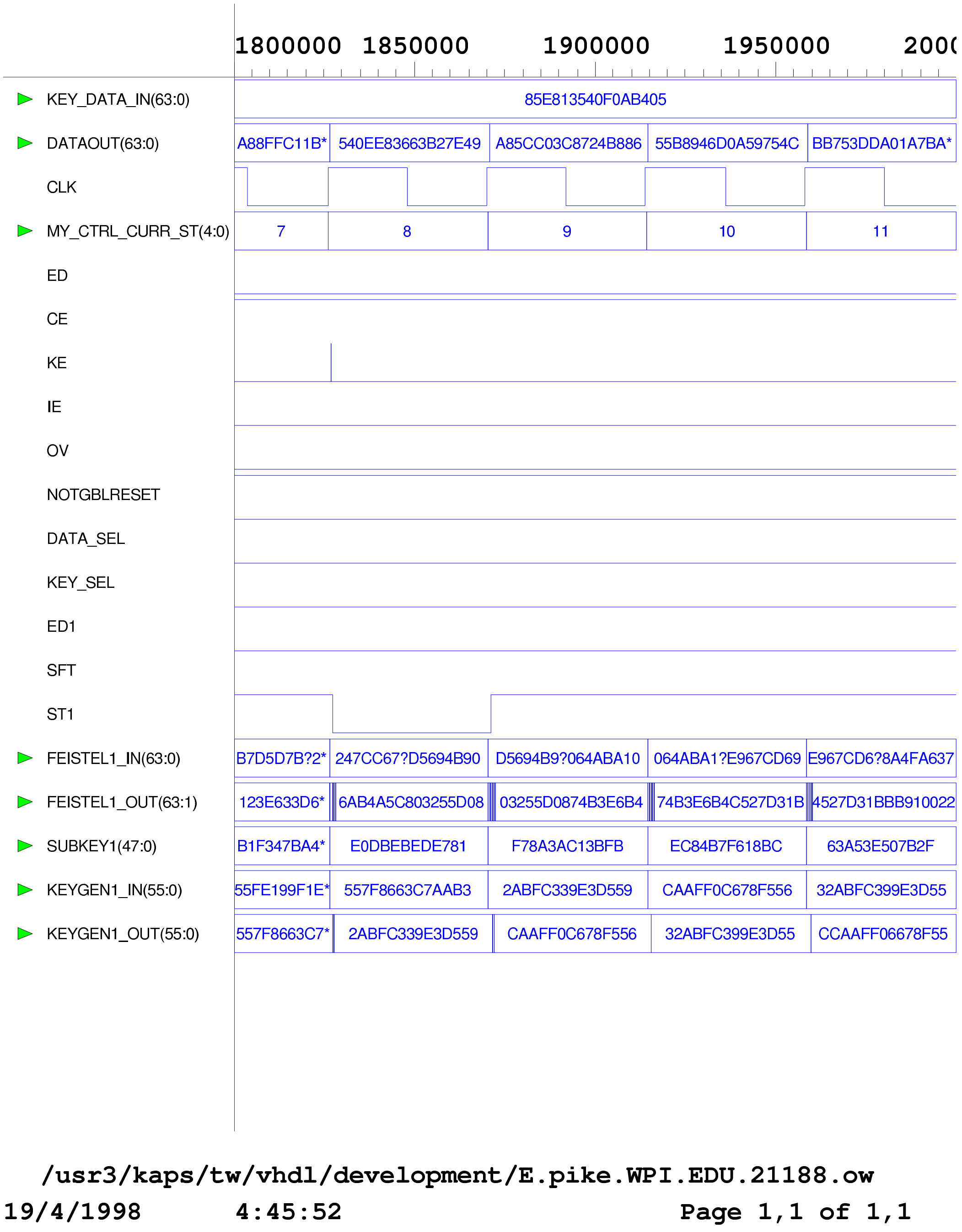
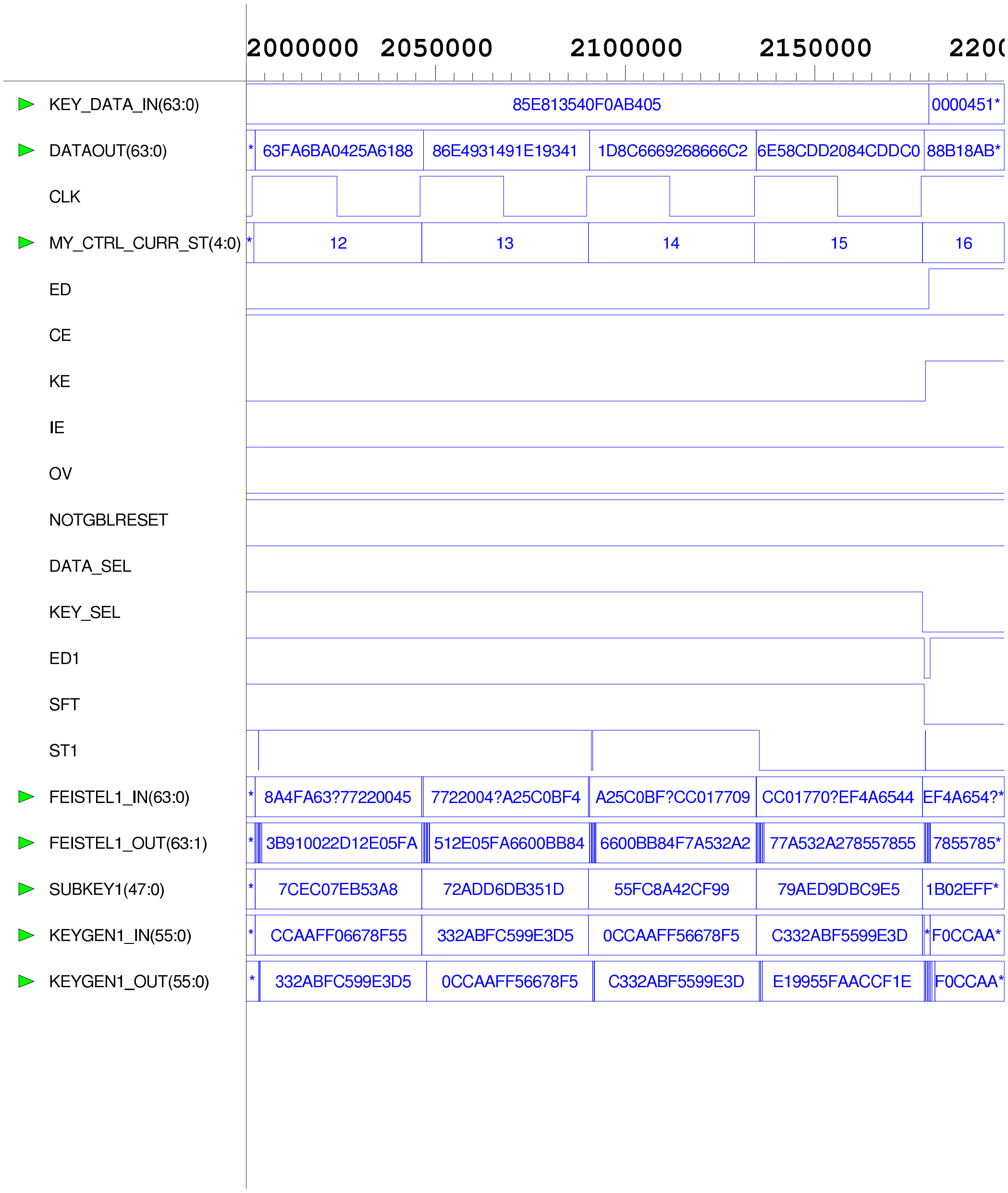


Figure D.8: Decryption with *DES_ED16* on the Chip 4008E-3-PG191



/usr3/kaps/tw/vhdl/development/E.pike.WPI.EDU.21188.ow

19/4/1998 4:47:39 Page 1,1 of 1,1

Figure D.9: Decryption with *DES_ED16* on the Chip 4008E-3-PG191

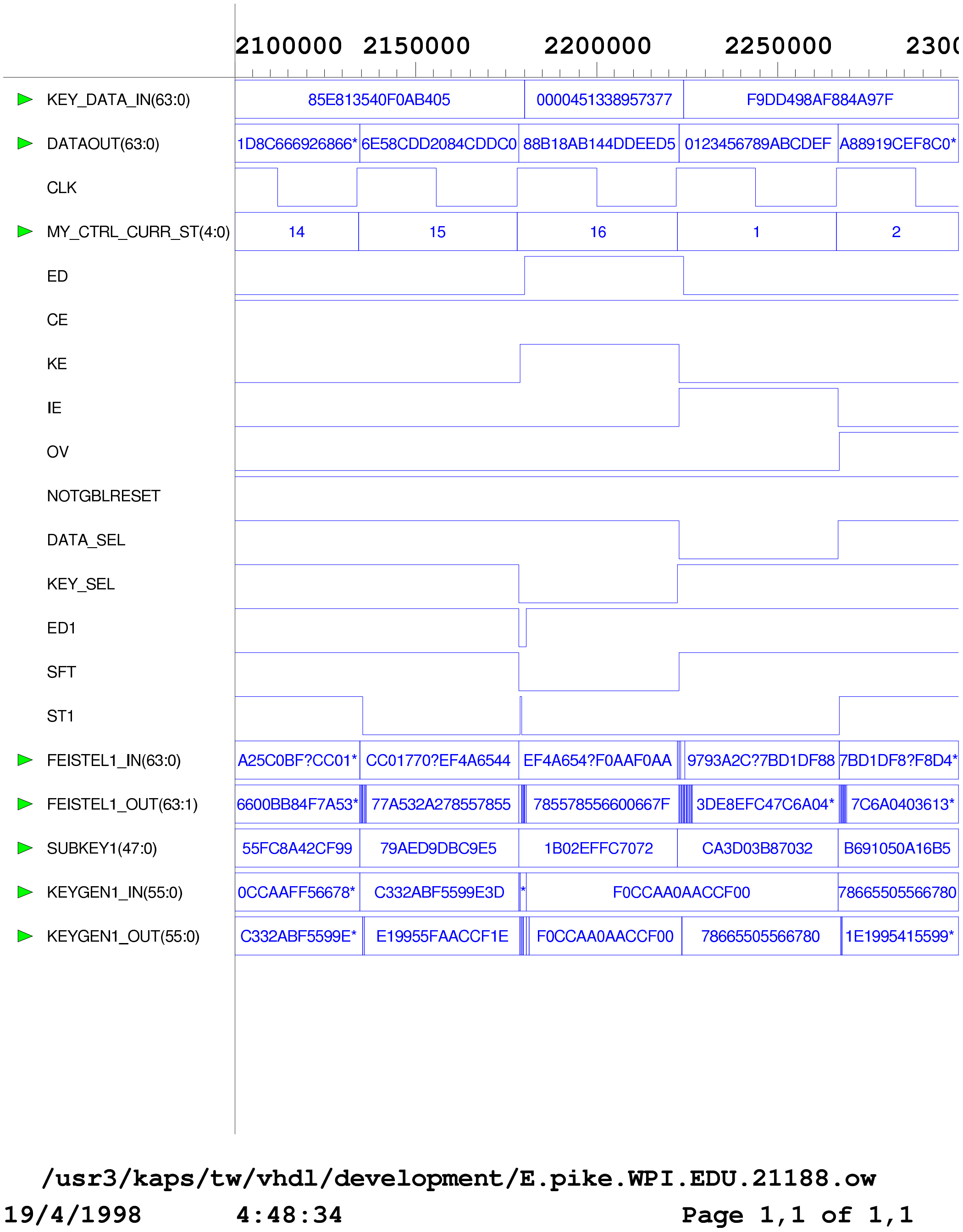


Figure D.10: Decryption with *DES_ED16* on the Chip 4008E-3-PG191

Appendix E

Test Bench

```
-- =============================================================== --
-- FPGA SIMULATOR     Testbench for Design with LogibloX           --
-- FOR XC4000e PARTYPES using Xilinx M1.3                          --
-- DES 16                                                          --
-- Jens-Peter Kaps                                         1/17/98 --
-- $Log: testbench.vhd,v $
-- Revision 2.1  1998/02/25  03:25:38  kaps
-- modified for des encryption/decryption,  full test
--
-- Revision 1.2  1998/0223  04:45:06  kaps
-- *** empty log message ***
--
-- =============================================================== --

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_textio.all;

-- =============================================================== --
-- Testbench Name is       E                                       --
-- =============================================================== --

ENTITY E IS
END E;

-- =============================================================== --
-- Define Architecture     AR                                      --
-- =============================================================== --
ARCHITECTURE AR OF E IS
```

```
-- =============================================================== --
-- Component Description                                           --
-- =============================================================== --

COMPONENT des PORT
    ( key_data_in : IN  std_logic_vector(63 downto 0);
      dataout     : OUT std_logic_vector(63 downto 0);
      clk         : IN  std_logic;
      ed          : IN  std_logic;      -- encryption / decryption
      ce          : IN  std_logic;      -- clock enable
      ke          : OUT std_logic;      -- key exspected
      ie          : OUT std_logic;      -- input exspected
      ov          : OUT std_logic;      -- output valid
      NOTGBLRESET : IN std_logic );
END COMPONENT;

-- =============================================================== --
-- Define the Signals                                              --
-- =============================================================== --

    SIGNAL key_data_in : std_logic_vector (63 downto 0);
    SIGNAL dataout     : std_logic_vector (63 downto 0);
    SIGNAL clk         : std_logic;
    SIGNAL ed          : std_logic;
    SIGNAL ce          : std_logic;
    SIGNAL ke          : std_logic;
    SIGNAL ie          : std_logic;
    SIGNAL ov          : std_logic;
    SIGNAL NOTGBLRESET : std_logic;

-- =============================================================== --
-- Instantiate the design for simulation                           --
-- =============================================================== --

BEGIN
    UUT : des PORT MAP (
            key_data_in => key_data_in,
            dataout     => dataout,
            clk         => clk,
            ed          => ed,   -- 0 = encryption,      1 = decryption
            ce          => ce,   -- 0 = disabled (stop), 1 = enabled (run)
            ke          => ke,
            ie          => ie,
            ov          => ov,
            NOTGBLRESET => NOTGBLRESET );

-- =============================================================== --
-- Start the simulation                                            --
-- =============================================================== --
```

```
    flow_process: PROCESS

    BEGIN

-- ================================================================ --
-- Start Values                                                     --
-- ================================================================ --

key_data_in <= "0000000000000000000000000000000000000000000000000000000000000000";
ed          <= '0';
ce          <= '0';
clk         <= '0';
NOTGBLRESET <= '0';
wait for 22 NS;
NOTGBLRESET <= '1';

-- ================================================================ --
-- Round  INIT      S T A R T   E N C R Y P T I O N                 --
-- ================================================================ --
clk         <= '1';
wait for 2 NS;
key_data_in <= "0000000000010010011010010101101111001001101101111011011111111000";
ce          <= '1';
ed          <= '0';
wait for 20 NS;
clk         <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  1                                                         --
-- ================================================================ --
clk         <= '1';
wait for 2 NS;
key_data_in <= "0000000100100011010001010110011110001001101010111100110111101111";
ed          <= '0';
wait for 20 NS;
clk         <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  2                                                         --
-- ================================================================ --
clk     <= '1';
wait for 22 NS;
clk     <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  3                                                         --
-- ================================================================ --
clk     <= '1';
```

```
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  4                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  5                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  6                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  7                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  8                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  9                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  10                                                        --
-- ================================================================ --
clk     <= ’1’;
```

```
wait for 22 NS;
clk     <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  11                                                        --
-- ================================================================ --
clk     <= '1';
wait for 22 NS;
clk     <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  12                                                        --
-- ================================================================ --
clk     <= '1';
wait for 22 NS;
clk     <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  13                                                        --
-- ================================================================ --
clk     <= '1';
wait for 22 NS;
clk     <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  14                                                        --
-- ================================================================ --
clk     <= '1';
wait for 22 NS;
clk     <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  15                                                        --
-- ================================================================ --
clk         <= '1';
wait for 22 NS;
clk         <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  16         S T A R T  2nd  E N C R Y P T I O N            --
-- ================================================================ --
clk         <= '1';
wait for 2 NS;
key_data_in <= "0000000000000000010001010001001100111000100101010111001101110111";
ed          <= '0';
wait for 20 NS;
clk         <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  1                                                         --
-- ================================================================ --
clk         <= '1';
wait for 2 NS;
key_data_in <= "0000111100011110001011010011110001001011010110100110100101111000";
wait for 20 NS;
clk         <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  2                                                         --
-- ================================================================ --
clk         <= '1';
wait for 22 NS;
clk         <= '0';
wait for 22 NS;
-- ================================================================ --
-- Stop Machine for one clock cycle                                 --
-- ================================================================ --
clk         <= '1';
wait for 2 NS;
ce          <= '0';
wait for 20 NS;
clk         <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  3                                                         --
-- ================================================================ --
clk         <= '1';
wait for 2 NS;
ce          <= '1';
wait for 20 NS;
clk         <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  4                                                         --
-- ================================================================ --
clk         <= '1';
wait for 22 NS;
clk         <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  5                                                         --
-- ================================================================ --
clk     <= '1';
wait for 22 NS;
clk     <= '0';
wait for 22 NS;
-- ================================================================ --
-- Round  6                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  7                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  8                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  9                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  10                                                        --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  11                                                        --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  12                                                        --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  13                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  14                                                       --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  15                                                       --
-- =============================================================== --
clk         <= ’1’;
wait for 22 NS;
clk         <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  16        S T A R T   D E C R Y P T I O N                --
-- =============================================================== --
clk         <= ’1’;
wait for 2 NS;
key_data_in <= "0000000000010010011010010101101111001001101101111011011111111000";
ce          <= ’1’;
ed          <= ’1’;
wait for 20 NS;
clk         <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  1                                                        --
-- =============================================================== --
clk         <= ’1’;
wait for 2 NS;
key_data_in <= "1000010111101000000100110101010000001111000010101011010000000101";
ed          <= ’0’;
wait for 20 NS;
clk         <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  2                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  3                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  4                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  5                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  6                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  7                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  8                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  9                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  10                                                       --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  11                                                       --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  12                                                       --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  13                                                       --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  14                                                       --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  15                                                       --
-- =============================================================== --
clk         <= ’1’;
wait for 22 NS;
clk         <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  16     S T A R T  2nd  D E C R Y P T I O N               --
-- =============================================================== --
clk         <= ’1’;
wait for 2 NS;
key_data_in <= "0000000000000000010001010001001100111000100101010111001101110111";
ed          <= ’1’;
wait for 20 NS;
clk         <= ’0’;
```

```
wait for 22 NS;
-- ================================================================ --
-- Round  1                                                         --
-- ================================================================ --
clk         <= ’1’;
wait for 2 NS;
key_data_in <= "1111100111011101010010011000101011111000100001001010100101111111";
ed          <= ’0’;
wait for 20 NS;
clk         <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  2                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Stop Machine for one clock cycle                                 --
-- ================================================================ --
clk         <= ’1’;
wait for 2 NS;
ce          <= ’0’;
wait for 20 NS;
clk         <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  3                                                         --
-- ================================================================ --
clk         <= ’1’;
wait for 2 NS;
ce          <= ’1’;
wait for 20 NS;
clk         <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  4                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  5                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
```

```
wait for 22 NS;
-- ================================================================ --
-- Round  6                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  7                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  8                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  9                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  10                                                        --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  11                                                        --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  12                                                        --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
```

```
wait for 22 NS;
-- ================================================================ --
-- Round  13                                                        --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  14                                                        --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  15                                                        --
-- ================================================================ --
clk         <= ’1’;
wait for 22 NS;
clk         <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  16         E N C R Y P T I O N   A G A I N                --
-- ================================================================ --
clk         <= ’1’;
wait for 2 NS;
key_data_in <= "0000000000010010011010010101101111001001101101111011011111111000";
ce          <= ’1’;
ed          <= ’0’;
wait for 20 NS;
clk         <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  1                                                         --
-- ================================================================ --
clk         <= ’1’;
wait for 2 NS;
key_data_in <= "0000000100100011010001010110011110001001101010111100110111101111";
ed          <= ’0’;
wait for 20 NS;
clk         <= ’0’;
wait for 22 NS;
-- ================================================================ --
-- Round  2                                                         --
-- ================================================================ --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
```

```
wait for 22 NS;
-- =============================================================== --
-- Round  3                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  4                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  5                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  6                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  7                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  8                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =============================================================== --
-- Round  9                                                        --
-- =============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
```

```
wait for 22 NS;
-- =================================================================== --
-- Round  10                                                           --
-- =================================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =================================================================== --
-- Round  11                                                           --
-- =================================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =================================================================== --
-- Round  12                                                           --
-- =================================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =================================================================== --
-- Round  13                                                           --
-- =================================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =================================================================== --
-- Round  14                                                           --
-- =================================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- =================================================================== --
-- Round  15                                                           --
-- =================================================================== --
clk         <= ’1’;
wait for 22 NS;
clk         <= ’0’;
wait for 22 NS;
-- =================================================================== --
-- Round  16        E N C R Y P T    A G A I N    A N D    R E S E T   --
-- =================================================================== --
clk         <= ’1’;
wait for 2 NS;
key_data_in <= "0000000000010010011010010101101111001001101101111011011111111000";
```

```
ce          <= ’1’;
ed          <= ’0’;
wait for 20 NS;
clk         <= ’0’;
wait for 22 NS;
-- ============================================================== --
-- Round  1                                                        --
-- ============================================================== --
clk         <= ’1’;
wait for 2 NS;
key_data_in <= "0000000100100011010001010110011110001001101010111100110111101111";
ed          <= ’0’;
wait for 20 NS;
clk         <= ’0’;
wait for 22 NS;
-- ============================================================== --
-- Round  2                                                        --
-- ============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ============================================================== --
-- Round  3                                                        --
-- ============================================================== --
clk     <= ’1’;
wait for 22 NS;
clk     <= ’0’;
wait for 22 NS;
-- ============================================================== --
-- Reset in the middle and decrypt this time                       --
-- ============================================================== --
clk         <= ’0’;
NOTGBLRESET <= ’0’;
wait for 22 NS;
NOTGBLRESET <= ’1’;
wait for 22 NS;
-- ============================================================== --
-- Round  INIT       S T A R T   E N C R Y P T I O N               --
-- ============================================================== --
clk         <= ’1’;
wait for 2 NS;
key_data_in <= "0000000000010010011010010101101111001001101101111011011111111000";
ce          <= ’1’;
ed          <= ’0’;
wait for 20 NS;
clk         <= ’0’;
wait for 22 NS;
-- ============================================================== --
-- Round  1                                                       --
-- ============================================================== --
clk         <= '1';
wait for 2 NS;
key_data_in <= "0000000100100011010001010110011110001001101010111100110111101111";
ed          <= '0';
wait for 20 NS;
clk         <= '0';
wait for 22 NS;
-- ============================================================== --
-- Round  2                                                       --
-- ============================================================== --
clk     <= '1';
wait for 22 NS;
clk     <= '0';
wait for 22 NS;
-- ============================================================== --
-- Round  3                                                       --
-- ============================================================== --
clk     <= '1';
wait for 22 NS;
clk     <= '0';
wait for 22 NS;

    END PROCESS flow_process;

END AR;

-- ============================================================== --
-- Configuration Statement                                        --
-- ============================================================== --

configuration CFG_TB of E is
    for AR
--        for UUT : LOGITEST
--            use configuration WORK.CFG_LOGITEST_BEHAVIORAL;
--        end for;
    end for;
end CFG_TB;
```

Bibliography

[1] S.A. Vanstone A.J. Menezes and P.C Van Oorschot. *Handbook of applied cryptography.* Discrete Mathematics and its Application. CRC Press, Florida, USA, 1997.

[2] H. Eberle amd C.P. Thacker. A 1 Gbit/second GaAs DES chip. In *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference*, pages 19.7/1–4, New York, NY, USA, 1992. IEEE, IEEE.

[3] A.G. Broscius and J.M. Smith. Exploiting parallelism in hardware implementation of the DES. In J. Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91. Proceedings*, number 576 in Lecture Notes in Computer Science, pages 367–376, Berlin, Germany, 1992. Int. Assoc. Cryptologic Res, Springer-Verlag.

[4] I. Eslick J. Brown E. Tau, D. Chen. A first generation DPGA implementation. In *FPD'95 – Third Canadian Workshop of Field-Programmable Devices*, page ?, 1995.

[5] H. Eberle. A high-speed DES implementation for network applications. In E.F. Brickell, editor, *Advances in Cryptology - CRYPTO '92. 12th Anual International Cryptology Conference Proceedings*, Lecture Notes in Computer Science, pages 521–539, Berlin, Germany, 1993. Springer-Verlag.

[6] J. Goubert F. Hoornaert and Y. Desmedt. Efficient hardware implementation of the DES. In G.R. Blakley and D. Chaum, editors, *Advances in Cryptology: Proceedings of CRYPTO'84*, number 196 in Lecture Notes in Computer Science, pages 147–173, Berlin, Germany, 1985. International Association for Cryptologic Research, Springer-Verlag.

[7] G.M. Haskins. Securing asynchronous tranfer mode networks. Masters thesis, WPI, Worcester, Massachusetts, USA, May 1997.

[8] J. Vandewalle I. Verbauwhede, F. Hoornaert and H.J. De Man. Security and performance optimization of a new DES data encryption chip. *IEEE Journal of Solid-State Circuits*, 23(3):647–656, June 1988.

[9] J. Leonard and W.H. Magione-Smith. A case study of partially evaluated hardware circuits: keyspecific DES. In P.Y.K. Cheung W. Luk and M. Glesner, editors, *Field-programmable Logic and Applications. 7th International Workshop, FPL '97*, Berlin, Germany, 1997. Springer-Verlag.

[10] T. McCall. Dataquest reports 82 million computers will be connected to the internet this year. http://www.dataquest.com, August 1997.

[11] A. Matusevich R.C. Fairfield and J. Plany. An LSI digital encryption processor. In G.R. Blakley and D. Chaum, editors, *Advances in Cryptology: Proceedings of CRYPTO'84*, number 196 in Lecture Notes in Computer Science, pages 115–143, Berlin, Germany, 1985. International Association for Cryptologic Research, Springer-Verlag.

[12] B. Schneier. *Applied Cryptography Second Edition: protocols, algorithms, and source code in C.* Wiley & Sons, New York, USA, 2nd edition, 1996.

[13] D.R. Stinson. *Cryptography: Theory and Practice.* Discrete Mathematics and its Applications. CRC Press, Florida, USA, 1995.

[14] Xilinx, San Jose, California, USA. *The Programmable Logic Data Book*, 1996.

[15] Xilinx, San Jose, California, USA. *Xilinx University Program Workshops*, 1997.