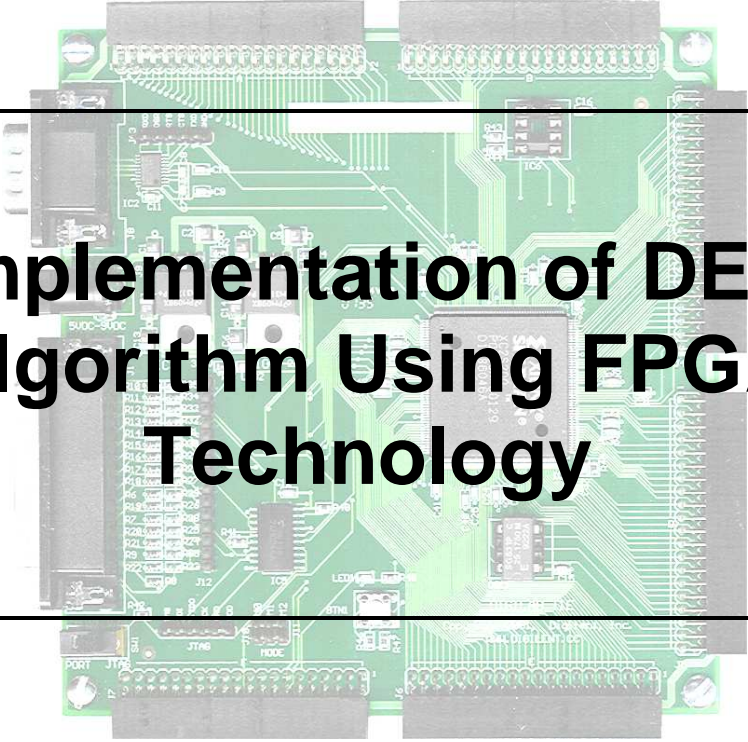

Microelectronic Systems Laboratory



2002 Winter Semester Project

A photograph of a green printed circuit board (PCB) populated with various electronic components, including integrated circuits, resistors, and capacitors. The board is shown from a top-down perspective, with several connectors visible along the edges.

**Implementation of DES
Algorithm Using FPGA
Technology**

Student: Arnaud Lager
Assistant: Ilhan Hatirnaz
Professor: Yusuf Leblebici

INTRODUCTION	4
OVERVIEW OF CRYPTOGRAPHY	5
SYMMETRIC-KEY ENCRYPTION	5
DATA ENCRYPTION STANDARD (DES)	6
<i>Examples of modern equipments which use DES encryption</i>	11
METHOD	12
TOOLS	12
<i>Software</i>	12
<i>Hardware</i>	12
DESIGN FLOW	14
UNDERSTANDING THE CODE	16
COMPONENTS	17
<i>desenc</i>	17
<i>keysched</i>	17
<i>IP</i>	17
<i>roundfunc</i>	18
<i>FP</i>	18
FAST DESIGN	19
OPTIMIZATIONS	19
SIMULATION PROCESS	21
SYNTHESIS PROCESS	22
PLACE & ROUTE	25
FPGA	26
SMALL DESIGN	27
SIMULATION PROCESS	30
SYNTHESIS PROCESS	31
PLACE & ROUTE	32
FPGA	33
CONCLUSION	34
ACKNOWLEDGEMENTS	35
REFERENCES	36
BOOKS AND ARTICLES	36
WEB SITES	36
APPENDIXES	37
FIGURES FOR FAST DESIGN	37
<i>Figure 1a</i>	37
<i>Figure 2a</i>	38
<i>Figure 3a</i>	39
<i>Figure 4a</i>	40
<i>Figure 5a</i>	41
<i>Figure 6a</i>	42
FIGURES FOR SMALL DESIGN	43
<i>Figure 1b</i>	43
<i>Figure 2b</i>	44
<i>Figure 3b</i>	45
<i>Figure 4b</i>	46
<i>Figure 5b</i>	47
EXCERPTS OF THE VHDL CODE	48

Fast design
Small design

48
66

Introduction

DES is probably one of the best known cryptographic algorithms, and has been widely used since its introduction in 1976 (and is still used today despite the fact that he doesn't offer a sufficient level of security).

The goal of this project is to continue the work of a student who worked on a pipelined VHDL implementation of the DES algorithm. Two architectures are studied for this project: one which is the fastest possible and another one which results in the less area than the first architecture on the FPGA. The meaning of speed for this project is the throughput (number of bits processed per second) and the meaning of area is number of CLB's (Configurable Logic Block).

Before building our design, we need an overview of cryptography, followed by a description of the DES algorithm. We will see then what and how will realise this project. The next step will be to analyze the components used in both design. So at this point, we will be ready to build the fast design going through the design flow (simulation, synthesis, place & route and tests on FPGA). After that, the small design will be built.

The appendixes at the end of the report contain all the figures and the main part of the VHDL code.

Overview of cryptography

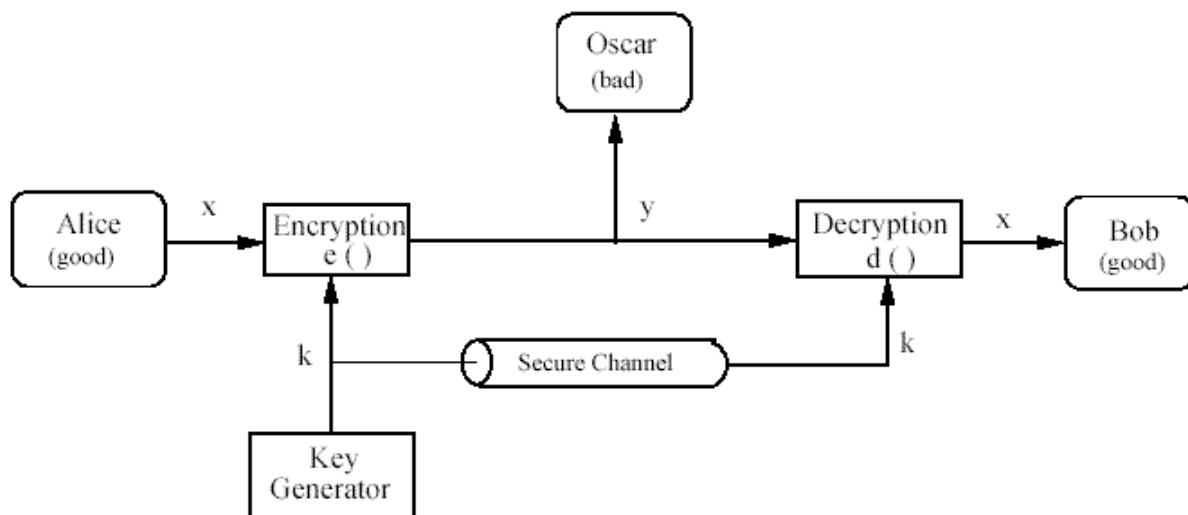
Before going into depth about this project, we shall explain some basic concepts about cryptography and particularly about DES, because we will use this algorithm.

Remark: all the figures in this section are taken from the excellent course of Prof. Christof Paar (see [1] for more details).

Symmetric-key encryption

Symmetric-key (or private-key) encryption can be simply illustrated with the schematic below.

Alice and Bob want to communicate over an un-secure channel, but Oscar is trying to read the message. So Alice and Bob must use a crypto system to prevent Oscar from reading the message.



Symmetric-key schematic

x is called the plaintext, y the ciphertext and k the key. The function e performs the encryption on the plaintext using the key and the function d decrypts the ciphertext using the same key. Thus, Alice and Bob must own the same key. They should not use the un-secure, since Oscar is able to retrieve all the data through this channel (the functions e and f are also known by Oscar). So the key must be exchanged through a secure channel.

We can also introduce some others definitions that will characterize the algorithm used:

- $\mathcal{P} = \{x_1, x_2, \dots, x_p\}$ is the plaintext space
- $\mathcal{C} = \{y_1, y_2, \dots, y_p\}$ is the ciphertext space

- $\mathcal{K} = \{k_1, k_2, \dots, k_p\}$ is the key space
- The encryption function $e_{k_i} : \mathcal{P} \rightarrow \mathcal{C}$ ($e_{k_i}(x)=y$)
- The decryption function $d_{k_i} : \mathcal{C} \rightarrow \mathcal{P}$ ($d_{k_i}(y)=x$)

The functions e_{k_i} and d_{k_i} are inverse function for the same key: $d_{k_i}(y) = d_{k_i}(e_{k_i}(x)) = x$

Data Encryption Standard (DES)

DES is the most popular symmetric-key algorithm. It was standardized in 1977 but expired in 1998.

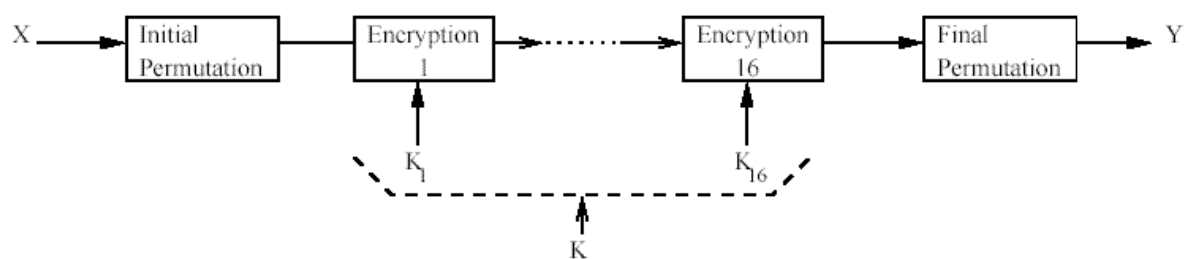
DES is a block cipher, which means that during the encryption process, the plaintext is broken into fixed length blocks and each block is encrypted at the same time. One block is 64 bits and the key is 64 bits wide (but only 56 bits are used).

So we can in a more formally manner describe the algorithm like this:

- $\mathcal{P} = \mathcal{C} = \{0, 1, 2, \dots, 2^{64}-1\}$
- $\mathcal{K} = \{0, 1, 2, \dots, 2^{56}-1\}$
- each x_i has 64 bits
- each k_i has 56 bits

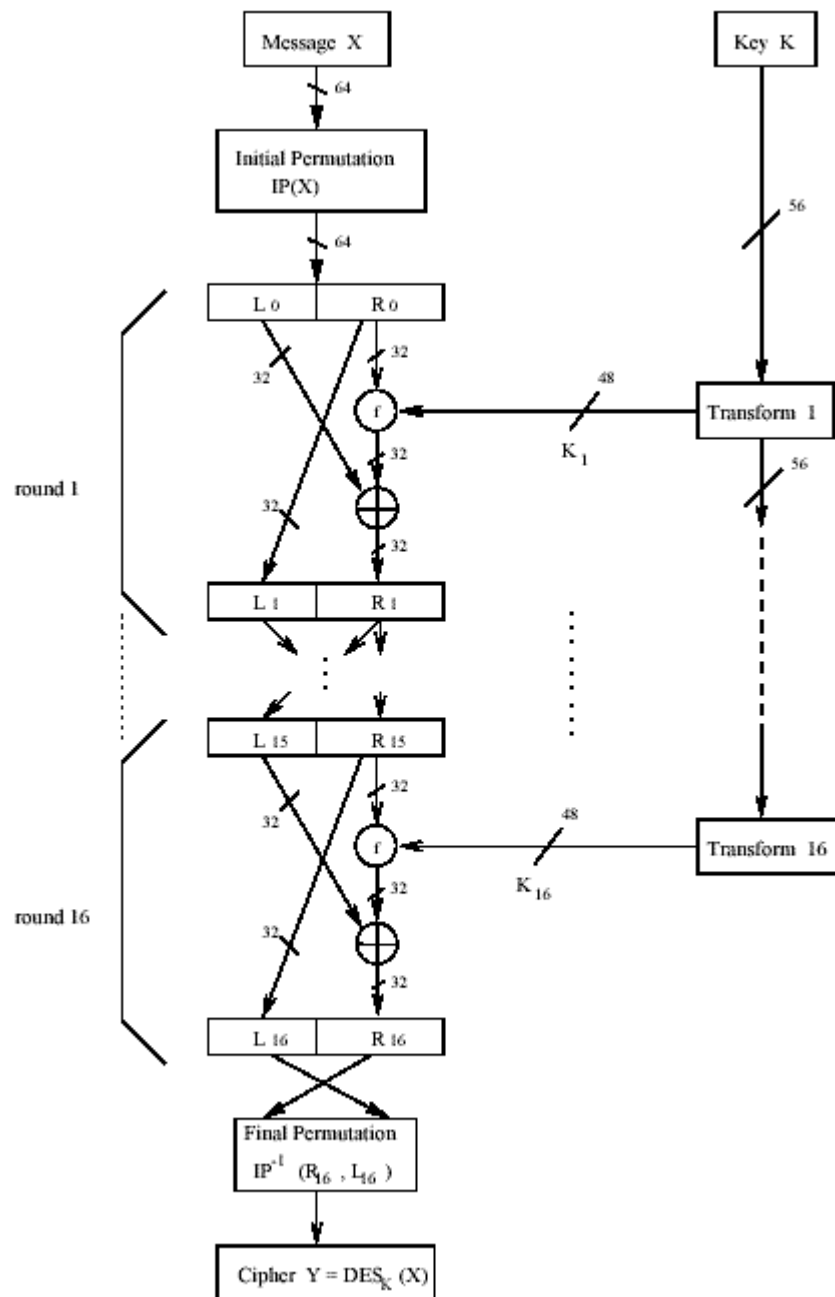
This description is not complete without the encryption function (e_k) and the decryption function (d_k).

The principle of DES encryption is made of an initial permutation, followed by 16 rounds and ended by a final permutation, as we can see in the next figure:



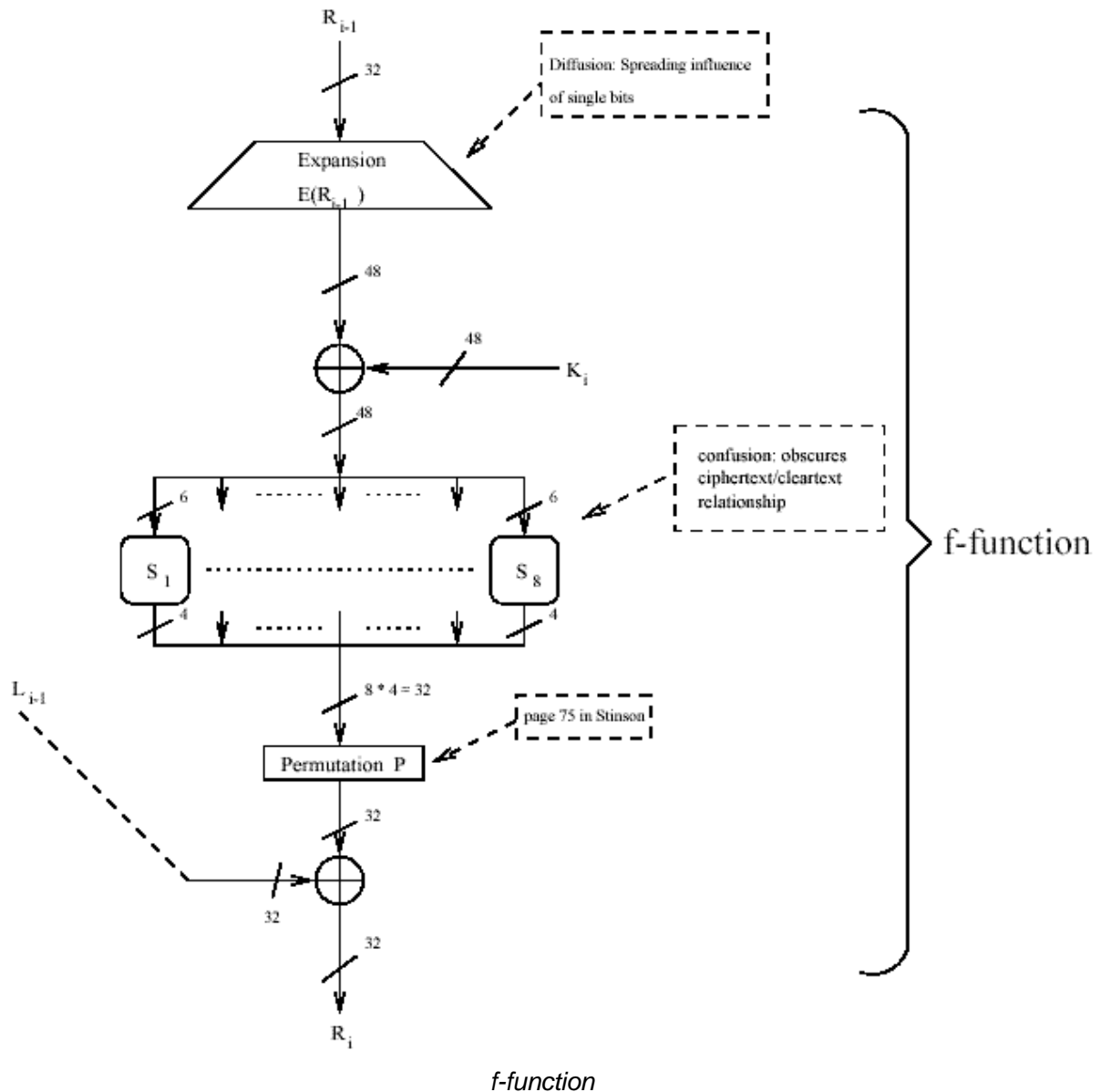
DES block schematic

Here is a more detailed version of this illustration:



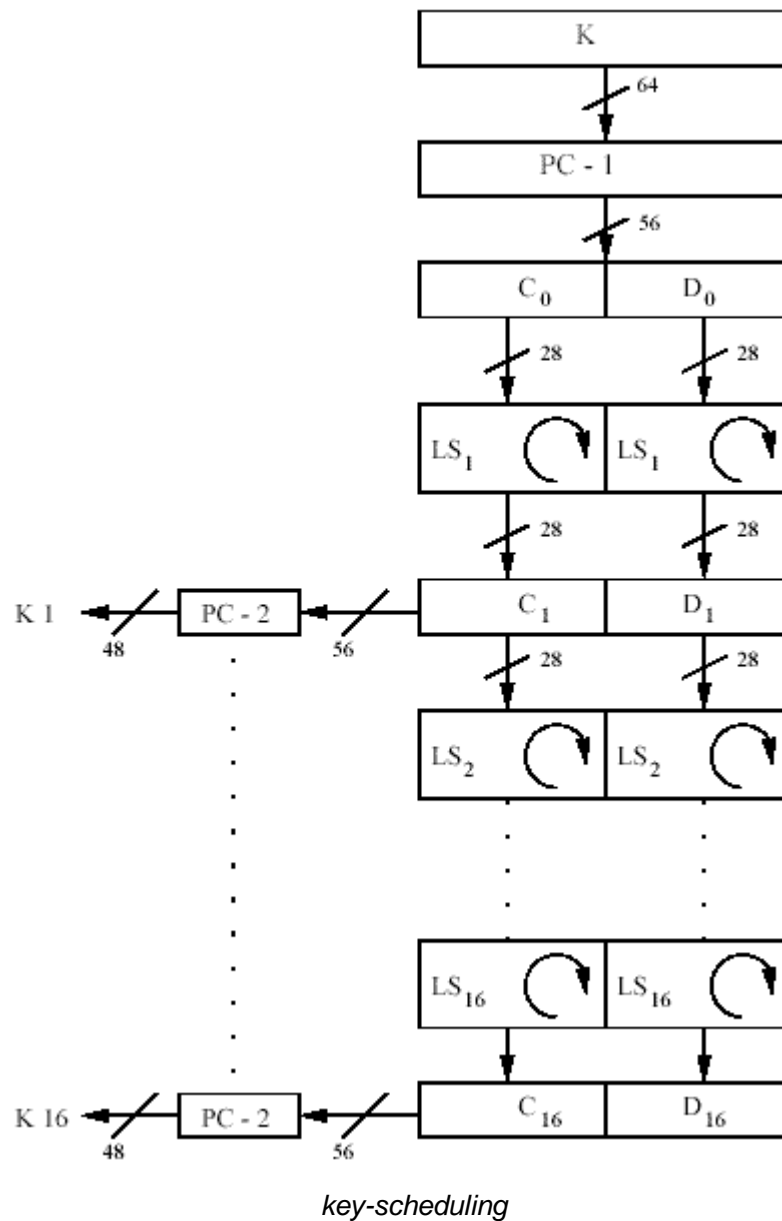
Feistel Network

The above figure is called the Feistel network. We can see the key-scheduling part at the right which is responsible to give a new 48 bits sub key for each round. Inside each round, the right part of the data is simply swapped while the left part is xored with the result of the f-function applied to the left part. Now take a closer look at the core of the DES algorithm: the f-function.



As we can see, the data coming to the f-function goes first through an expansion block and is then xored with the sub key. After that, the data arrives at the S-boxes, which are look-up tables. The next step is a simple permutation and finally, the resulting data is xored with the left part.

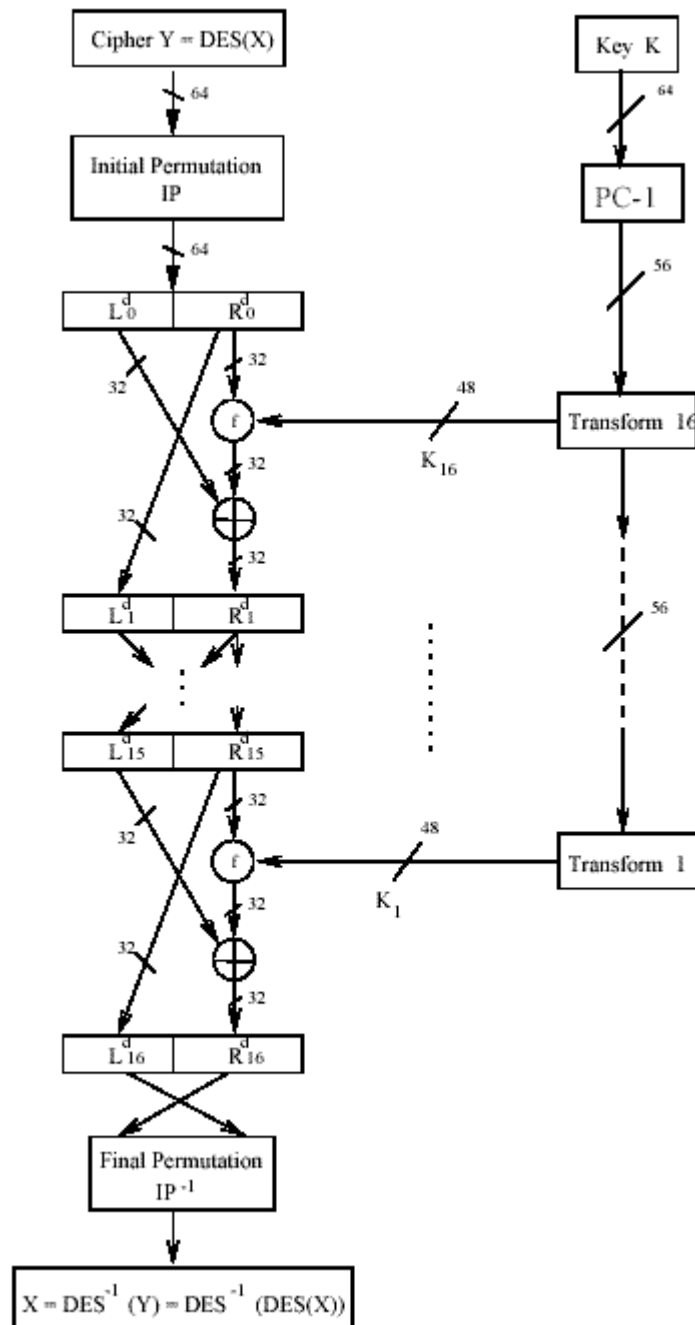
At this moment, we must find out how the key-scheduling achieves to build the sub keys.



We saw before that the key is initially 64 bits wide. But in the algorithm, only 56 bits are really used. So we can notice in the above figure that the component $PC-1$ removes these 8 bits to have the correct size. $PC-1$ also permutes the other bits. Then, at each step the key is shifted on the left one or two times. Before delivering the sub key, the component $PC-2$ reduce and permutes the 56 bits shifted key.

Now that we had a good overview of the encryption function (e_k), we can take look at the decryption function (d_k).

The decryption process is very similar as the encryption one, as we observe in the next schematic:



decryption

Obviously, the decryption is very similar to the encryption. Only the key-scheduling is reversed.

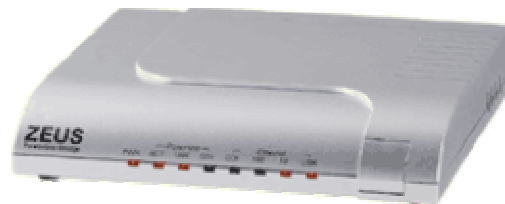
Additionally, we must highlight that there are four standardized modes of operation of DES: *ECB* (**E**lectronic **C**ode**b**ook mode), *CBC* (**C**ipher **B**lock **C**haining mode), *CFB* (**C**ipher **F**eed**b**ack mode) and *OFB* (**O**utput **F**eed**b**ack mode). We won't detail all the modes of operation; we just need to know that in our project, we will use the *ECB* mode (for a detailed description of DES, see [2]). In *ECB* mode, each plaintext block is encrypted independently with the block cipher.

Before going straight into the project, let's take a look at today's applications of DES.

Examples of modern equipments which use DES encryption

DES has expired in 1998 and has thus been replaced by stronger encryption algorithms, like AES. But DES is although still widely used if we don't need a high level of security.

Modern applications of DES cover a wide variety of applications, such as secure internet (ssl), electronic financial transactions, remote access servers, cable modems, secure video surveillance and encrypted data storage.



Ethernet Bridge 10/100 Mbps which use DES

Despite the fact that DES is considered un-secure, there is a way to enhance it. By encrypting three times successively, we get an adequate level of security, even for sensitive data. This method is called Triple DES.

The major drawback of Triple DES is the speed which is much lower than other modern algorithm like AES.

Method

Tools

Software

ModelSim

ModelSim 5.5 is a tool to simulate VHDL file and is used to verify the behaviour of the code. A test bench file is very useful to check immediately and automatically if the design is working as expected.

FPGA Compiler II

The next step is to synthesize the code using FPGA Compiler II. In fact, Synopsys (FPGA analyzer) was originally utilised, but was replaced by FPGA Compiler II. The reason of this change is that FPGA Compiler II is a much more recent software than Synopsys, therefore we can expect better performances.

Leonardo Spectrum 2002

In complement of FPGA Compiler II, I decided to try another synthesizer in parallel to compare.

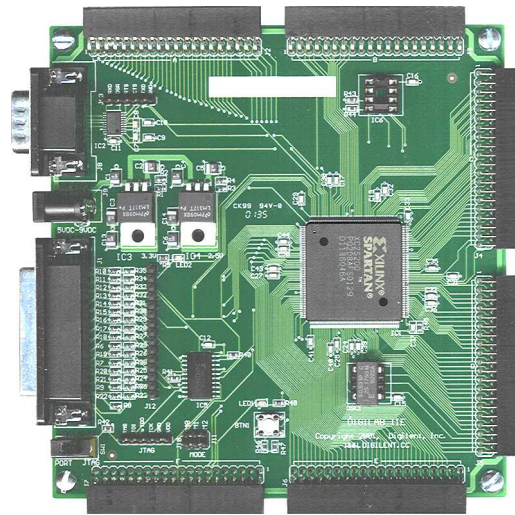
Xilinx ISE

The final step is to place and route the synthesized code with Xilinx ISE. ISE means Integrated Software Environment. It provides a lot of tools to accomplish each step of the design process from design entry to download the design to the FPGA. A synthesizer (XST) is part of these tools. One of the others ISE's tools is called Impact and is used to download the bit generated file directly to the FPGA.

Hardware

Spartan II

The FPGA we used in our project is a Spartan II XC2S200 package PQ208. This FPGA is integrated into the board Digilab 2 (see the picture below).



Digilab 2

Here are some main characteristics about our FPGA:

characteristic	value
System Gates	200'000
Logic Cells	5'292
Slices	2352
I/O	140

As we observe these results, we can imagine that this FPGA will provide enough space for this project.

The original implementation was tested with a Xilinx 40150XVPG559 which provides more space and I/O.

Logic analyzer

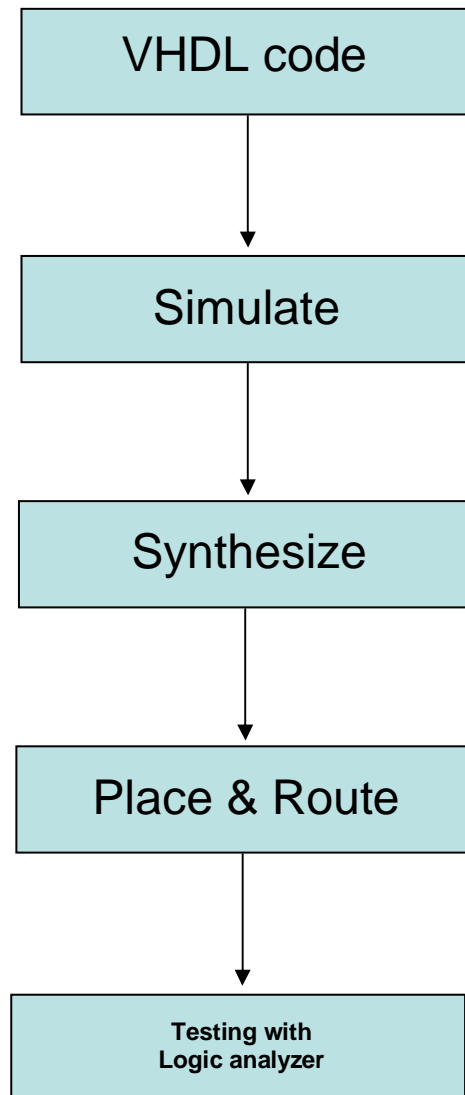
The last step in this project consists of testing the FPGA itself. As the numbers of pins used will be too high to test with an oscilloscope, we will use a special test equipment called a Logic analyzer. A logic analyzer is a device that can generate the input we want (with the pattern generator) and retrieve the output and display them (the analyzer part).

Design flow

The original files were given in VHDL. These files were provided by a student who wanted to improve (and test) his knowledge in VHDL and cryptography.

As we will see later, initially some mistakes had to be corrected and some improvements were needed.

The next steps are summarized in the design flow chart below, using the tools we mentioned above:



Design flow chart

The goal of this project is to optimize in 2 ways: area (less gates as possible) or speed (maximum throughput). Then, we will have two designs: one called fast design and another one called small design.

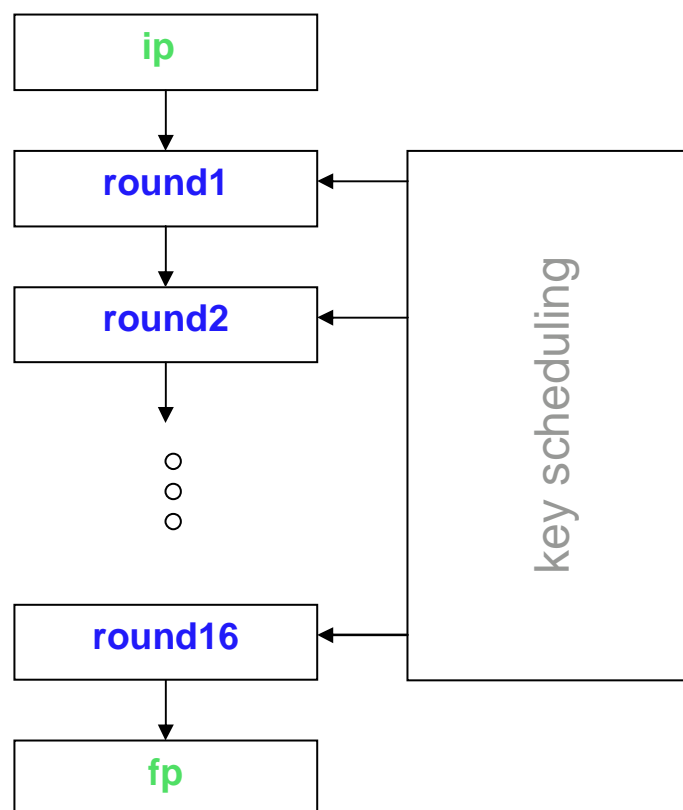
But before starting working on those designs, we should understand the supplied code which will be useful for both designs.

Understanding the code

Before going through the design flow, a verification on the supplied code is required. There were 2 files given for this project. The first one, called “pipelined-des.vhdl”, contains the implementation of the DES processor. The other one, called “testbench1.vhdl”, contains a benchmark which will be used in the simulation process.

As the name says, “pipelined-des.vhdl” is a 16 stages pipelined version of a DES processor. This is a structural code which uses 16 rounds.

By reading and analyzing the file, we can write a first schematic about how the code works.



Structure of the code

This figure corresponds to what we learnt in the description of the DES algorithm. The components at the beginning and at the end are respectively, initial permutation and final permutation (as we saw in the overview of DES).

If we compile the code immediately as-is, we notice that there are several errors. Putting all the components in one file was not a good idea and ModelSim refuses to compile the code like this.

So the first action was to split the supplied code in the different components. We also had to do some modifications (add the library and the declaration of the components used) to make the code work.

Furthermore, the file was not written in standard VHDL. For instance, the operator “rol” used in the key scheduling doesn’t work with `std_logic_vector` and there were some type ambiguities in the S-boxes files.

Now we can more easily describe the function of each component.

Components

desenc

This is the top design. It comprises 4 components:

- `keysched` (key scheduling)
- `ip` (initial permutation)
- `roundfunc` (round function)
- `fp` (final permutation)

We will know detail the function of these components (and their subcomponents).

keysched

This is the key scheduling part. It includes 2 components:

- `pc1` (permuted choice 1)
- `pc2` (permuted choice 2)

PC1 and PC2 both are permuting bits components. PC1 discards 8 bits from the key (which is originally 64 bits wide). In practice, these 8 bits are used to check if the key has not been altered (with a parity check).

PC2 also discards some bits to reduce the number of bits from 56 to 48.

The `keysched` component applies first PC1 then shifts one or two times one sub key after one sub key. Then at the end PC2 is applied at the end.

This leads to this observation: all the sub keys are created in a row.

The component `keysched` (and his subcomponents) is using only wiring resources, because it is made of permuting and shifting operations only. So this part will be executed very fast and no optimizations would have any effect.

IP

The initial permutation is only a matter of swapping the bits of the input (the plaintext). So we can already affirm that this component will not require logical resource (only wiring).

roundfunc

This is the round function, repeated 16 times as stated in the top design. It is in fact a structural design which connects the following components together:

- xp (expansion)
- desxor1 (xor 1)
- s1, s2, s3, s4, s5, s6, s7, s8 (S-boxes)
- pp (p-permutation)
- desxor2 (xor 2)

xp stands for expansion, since its behaviour is to expand the number of bits from 32 to 48 bits (so this is implemented with only wiring resources). desxor1 is a giant 48 bits xor gate which xor the sub key and the expanded input of the round function.

Then we see the 8 S-boxes that are in fact look-up tables. The registers necessary for the pipeline are also integrated in the S-boxes.

pp is P permutation, so bits swapping.

Finally, another xor (desxor2) is responsible to xor the result of the P permutation with the left part of the preceding round.

FP

The final permutation is the inverse of the initial permutation.

So we can conclude that a lot of the components are made only of wiring resources. So there won't be any effort in logic optimization on these components.

The only components that will use logical resources:

- desxor1
- desxor2
- the S-boxes

Fast design

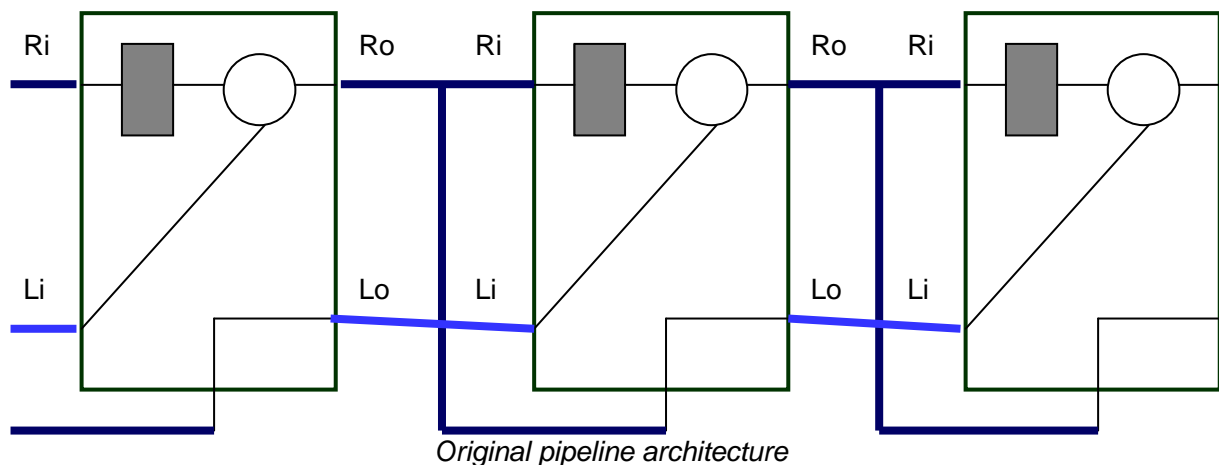
We studied all the components in details so we can build first our fast design. But before doing that, we will add optimizations to the VHDL code to reach an even higher speed.

Optimizations

desxor1 and desxor2 can not be logically optimized because they are using the basic xor functions.

Now let's focus on the S-boxes. They correspond to a large case in VHDL, which will be implemented in look-up tables in the FPGA.

We can use the following figure to explain how the pipeline architecture works:



The S-boxes which contain the registers are highlighted in grey on the figure above. As the registers are 4 bits wide (size of the output that needs to be saved) and as we have 8 S-boxes, we get 32 bits memorized in each stage (one stage = one round). It may look a bit strange because we have 64 bits to transmit between each stage. But if we consider the DES algorithm, the left part of the next round is the right part of the preceding one. So only half of the information needs to be stored. By doing this way, the author of the code privileged an architecture optimized for area, as we store only 32 bits instead of 64 bits. But this approach leads to a less efficient design in term of speed. To prove this affirmation, we need to examine the critical path (which determines actually the clock rate).

The critical path in a pipelined design is the longest path between two stages (that is, between two registers). We will characterize it using the figure above. Subsequently we have two paths to consider:

1. Start from the register, go through the f-function, go from *ro* to *ri* and arrive at the next register.

2. Start from the register, go through the f-function, go from ro to li , go to lo , then to li from the next round, go through the f-function, go from ro to ri and finally arrive at the next register.

We conclude that the second path is the longest one and can be reduced a lot. Indeed, the second path has the longest distance to run, but above all, it's going twice through the f-function (and the xor that has been merged in the f-function in the above figure for sake of simplicity).

Here we introduce a second register that will reduce the critical path to one stage (that is, one round). We will put the two registers at the end of the round. Therefore we expect to have a great boost of speed.

An useful modification to the design is to add a reset pin. This is really easy to implement and this won't have a big impact on final performances.

Another point which needs to be studied is the problem of the I/Os. As we saw before, the FPGA (Spartan-2) has only 140 I/Os. And we have:

- 64 bits for the plain text
- 64 bits for the key
- 1 bit for the reset
- 1 bit for the clock
- 64 bits for the cipher

The sum is 194 I/Os! So we need to reduce the numbers of I/Os. One of the way to do this is to use a converter which will load a portion of the data during some clock cycles and then deliver the full data (and so on repeatedly). We can thus split the inputs (key and plain text) like this:

- 16 bits for the plain text
- 16 bits for the key
- 1 bit for the reset
- 1 bit for the clock
- 64 bits for the cipher

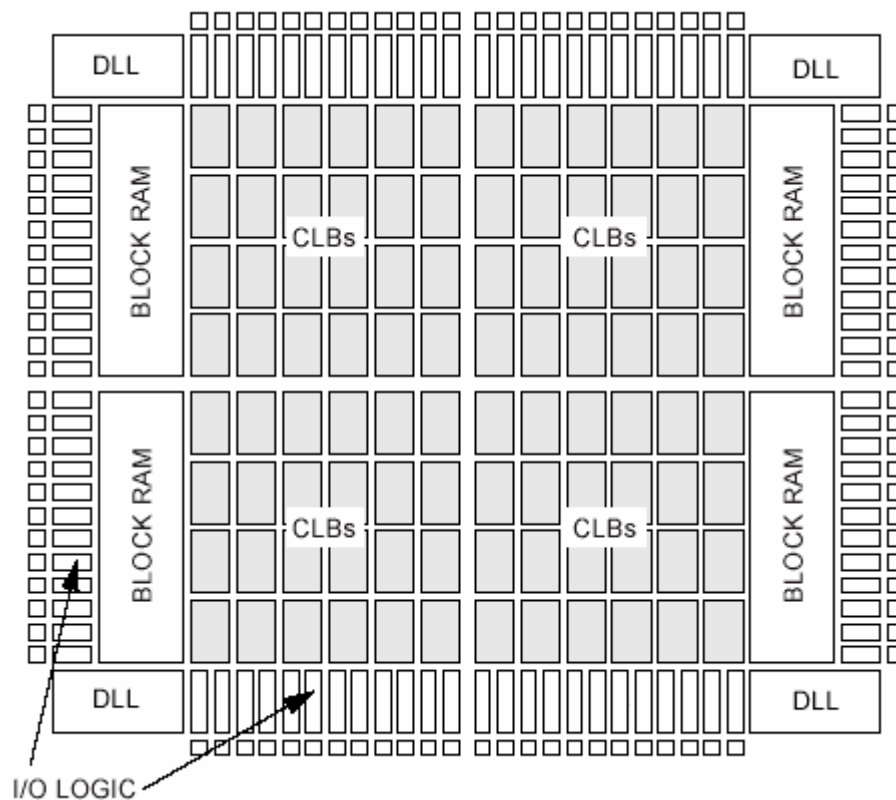
The sum is now 98 so our design fit in the FPGA (for the I/Os).

Therefore, two converters will be added to the design which will take 4 clock cycles to gather the required 64 bits for the plain text and the key. The behaviour of a converter is very similar to a serial to parallel converter except that in our case, we convert block of several bits to a larger block. In fact only 3 clock cycles are needed for this operation, because at the third clock cycle, the incoming data is concatenated with the stored data.

A signal added to the design called "load_data" informs the converters that a new key/plain text is passed at the inputs of the design.

The addition of these converters implies that we need to consider two designs: one with the converters that will be implemented on our FPGA and the other one without the converters which will not be implemented on a FPGA, but compared with the original design.

A further point to explore would be to implement the S-boxes with ROM elements. The FPGA has some areas reserved for those elements, as seen on the next figure:



Block RAM on the Spartan II

Rom would be better implemented in the Block RAM instead of using the CLB's, but in this project we won't go that far. It would be interesting to use this technique if we want a fully optimized design. This would of course reduce the number of CLB's used but it may also speed-up the implementation. So for this project, we will trust the synthesizer to have the best implementation of the S-boxes.

We are now ready to test our optimized design.

Simulation process

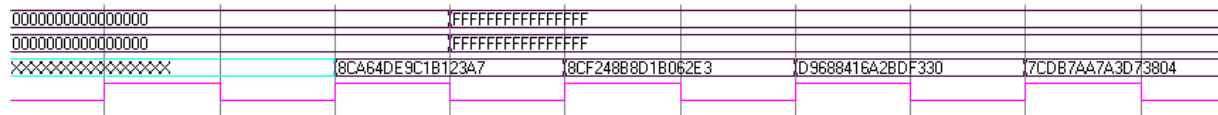
To make the simulation of the design easier, one file was provided: testbench1.vhd. The principle of this test bench is a black box that uses the top design (desenc), give it some predefined simulation vectors (the plain text, the key and the clock) and verify if the output (the cipher) corresponds to the valid cipher.

The test bench gives in fact a new plain text and a new key every 16 clock cycles and compare the output at the end with the associated cipher.

Our pipeline architecture gives one block of cipher (64 bits) every clock cycle, except at the beginning when the pipeline need to be filled (during 16 clock cycles because we have 16 stages), so there is a latency of 16 clock cycles.

We could therefore write another test bench that gives one plain text during 16 clock cycle and then one plain text each clock cycle, but this wouldn't be handy. Indeed with the supplied test bench, we can immediately compare the key and the plain text that have still the same value with the computed cipher (as we will see in the next figures).

After that, we obtain this waveform:

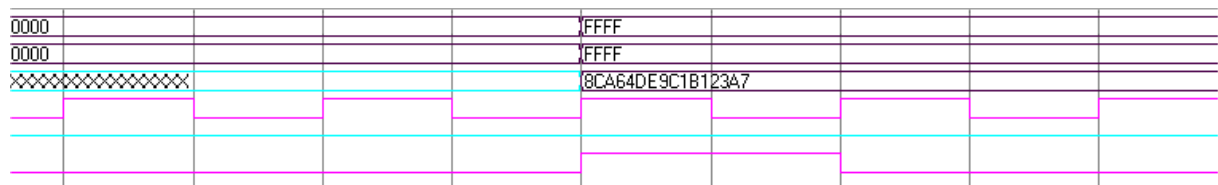


The first line in the figure is the plain text, the second one is the key, the third one is the cipher, the fourth one is the clock and the last one is the reset.

The cipher is the one we expected. We can see that during the next cycles after the first cipher, other ciphers are present, but they don't mean anything. Normally we will give one cipher each clock cycle, so there wouldn't be those "parasites cipher".

Considering now the version with the converters, we must write another test bench which gives 16 bits for the key and the plain text and which handle the signal load_data.

This new test bench gives those results:



Another line (the last one) was added which represents the load_data signal.

The given cipher is correct.

Since we verified the operation of our design, we can now go through the synthesis process.

Synthesis process

Considering that we must compare the speed of our design with the original one but that we will implement it on a different FPGA, we will synthesize twice, targeting the two FPGAs separately.

The choice of using two synthesizers allows us to compare the result and choose the best one.

Let's compare now our result with the original one. In fact, we know the results because they were given in the header of the VHDL file as a comment.

- FPGA: Xilinx XC40150XV-PG559-09
- Max. predicted clock rate: 11.7MHz
- Size: about 4700 CLB's

Now let's run both synthesizer with our implementation and compare the results in the next table:

	Estimated Clock rate		Area (CLB's)	
Original design	11.7 MHz		4700	
	Leonardo		FPGA Compiler II	
	Estimated Clock rate	Area (CLB's)	Estimated Clock rate	Area (CLB's)
New design	32.7 MHz	3840	52 MHz	4864

So, the new design has a much higher speed. We can explain the difference of speed with the two registers which almost multiply the clock rate by 2.

During the synthesis process, Leonardo gives some information. One of these is that he infers ROM elements for the S-Boxes (see appendixes, figure 2a). It means that he will implement a ROM, but using CLB's. Perhaps in the original design, the S-boxes have been implemented with multiplexers. That would also explain the different amount of CLB's and speed.

FPGA Compiler II produces a better result than Leonardo, achieving a higher frequency, but uses more CLB's (even more CLB's than the original design). To understand why, we must compare the two implementations (see figure 5a and 6a of the appendixes). We immediately notice that the S-boxes are different: the version of FPGA Compiler II uses more CLB's than Leonardo (the other components are implemented identically). To be precise, Leonardo uses 20 CLB's while FPGA compiler uses only 28 CLB's. Because we have 8 S-boxes per round and 16 rounds, the difference is $8 \cdot (28 - 20) \cdot 16 = 1024$ CLB's. It explains exactly the difference of area between both implementations: $4864 - 3840 = 1024$.

We can thus conclude that FPGA compiler obtains a better speed because he found a better way to implement the S-boxes. He used more CLB's than Leonardo, but we don't care since we are optimizing for speed.

Another conclusion is that the implementation of the S-boxes are decisive to achieve a high speed.

Finally, no logical resources are used in the components that needed only wiring resources and no latches can be found. We can check too that there are 1024 registers, as expected ($64 \text{ bits} \cdot 16 \text{ rounds} = 1024 \text{ bits to store}$).

The throughput is computed like this: frequency * 64 bits (because we have a block cipher of 64 bits each clock cycle with this pipeline implementation). So in our case we have:

- 748.8 Mbits/s for the original design
- 2092.8 Mbits/s for the new design synthesized by Leonardo
- 3328 Mbits/s for the new design synthesized by FPGA Compiler II

So we get a 344% faster design than the original one, with our design synthesized by FPGA Compiler II !

Now let's focus on the design we will use on the Spartan II (which has 2 converters).

	Leonardo		FPGA Compiler II	
	Estimated Clock rate	Area (CLB's)	Estimated Clock rate	Area (CLB's)
Fast design	83.1 MHz	3336 (=1668 slices)	156 MHz	3330

(Remark: we choose speed grade -5 and maximum effort for delay)

Two types of converters were tested: one which is state machine of 4 states and another one which have a counter. Here is the VHDL code of the second one (the main part of the code):

```
process(clk)
variable counter : integer :=0;
begin

if (clk'event and clk = '1') then

    if(load_data='1') then
        counter := 0;
        memory <= input & memory(1 to 32);
    end if;

    if(counter > 2) then
        output <= memory & input;
    else
        memory <= input & memory(1 to 32);
    end if;
    counter := counter +1;
end if;

end process;
```

If we choose the state machine, we get a clock rate 5% faster than the one with the counter (with less area). Obviously, we will prefer the converter with the state machine.

This time Leonardo and FPGA Compiler II got exactly the same implementation of the round (S-boxes with 16 CLB's). The only difference of amount is due to the converters (FPGA compiler uses 1 CLB while Leonardo uses 4).

We can remark that the implementation is better on the Spartan II because it requires much less area than with the Xilinx XC40150XV.

But the difference between the two clock rates is very surprising. We should be able to understand why they are so different during the place & route (where we will get the final clock rate).

The throughputs are:

- 5318.4 Mbits/s for the design synthesized by Leonardo
- 9984 Mbits/s for the design synthesized by FPGA Compiler II

This is really impressive (several gigabits per second) and is a lot more than the implementation with the other design.

We have now an optimized design, which functions correctly. We're therefore ready to go through the place & route process.

Place & Route

Xilinx XC40150XV is not supported in the current version of Xilinx ISE, so we won't be able to place & route (and retrieve the statistics about clock rate, area).

The first thing to do before running ISE is to build a constraint file. We need this file to have the inputs (and outputs) all together at one socket of the board, so that the connections from the logic analyzer to the board will be easier. This constraint file contains directive to assign the I/Os of our design (pt, key, clk, reset and ct) to the FPGA's pins. We accomplish this by reading the reference manual of the FPGA (Digilab 2) available on the manufacturer's website: http://www.digilentinc.com/assets/documents/d2_rm.pdf

At the end of the place & route, we get one bit generated file which will be downloaded to the FPGA and also some information about our design. Here are the results for the design synthesized by Leonardo (targeting the Spartan 2 device):

	Leonardo		FPGA Compiler II	
	Clock rate	Slices	Clock rate	Slices
Fast design	68.2 MHz	2025	62.3	2030

The final frequency has dropped a lot and Leonardo has this time synthesized a better design. Although the difference between both clock rates is tiny. So the clock rate given by FPGA Compiler II at the end of the synthesis was too optimistic.

The final throughputs are:

- 4364.8 Mbits/s for the design synthesized by Leonardo
- 3987.2 Mbits/s for the design synthesized by FPGA Compiler II

We can also see that we are using about 86% of the total area of the FPGA.

FPGA

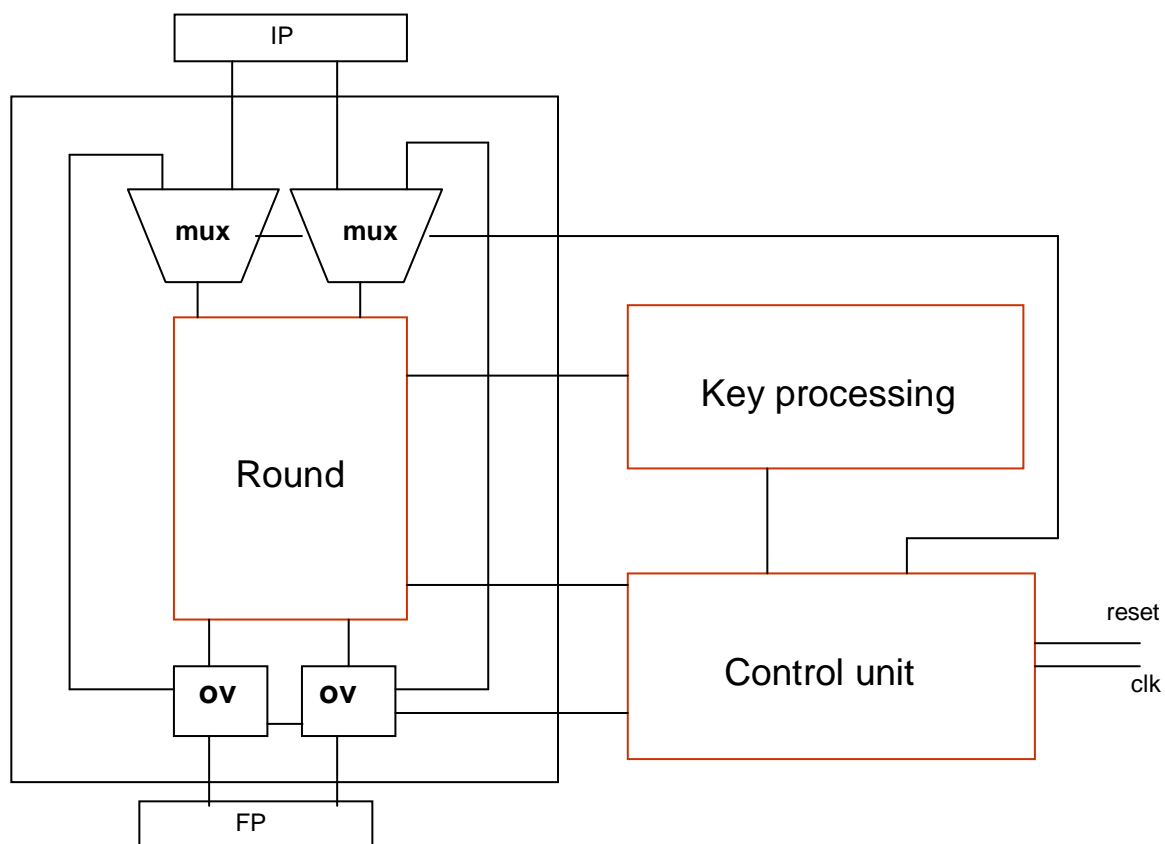
After having downloaded the bit generated-file to the FPGA, the next step is to use logic analyzer to test directly on the FPGA. Unfortunately this step couldn't be completed because a lot of problems with connections have occurred and time was missing at the end.

Small design

In order to obtain a design with the smallest possible area, we must change our architecture. We had a pipelined architecture which is very efficient to achieve a high-speed but now we want to use as low area as possible.

Obviously, the pipelined architecture is wasting a lot of area, as it is using 16 times the same round. So the challenge is to find a way to use only one round and make the data loop 16 times.

This idea can be implemented with a state machine. This leads to the following figure:



By doing this, we can reuse the same round components we had with the pipelined architecture. The key scheduling must however be rewritten in a key processing unit (we call this key processor, or key processing unit). Naturally, a control unit must be included in the design. All these additional components will not cost a lot of area.

But we have also to include two multiplexers at the inputs of the round. As the inputs are 64 bits wide, we will have two 32 bits multiplexers, which may cost a lot of area. These two multiplexers either let the input of the design go through the round (load new data), or take the output of the round to reinsert them in the round (loop).

Just after the round and before the final permutation, we have two output validators (ov on the figure above). These components will keep the same output between each

new data/key (every 16 clock cycles). We could use different methods like frequency divider but the output validers are most likely the easiest solution with a low cost in term of area.

We can imagine that we have a state machine of 16 states, but in reality we will have 17 (because we need one state to load the key). We will see that in detail later.

With this new architecture, we have a throughput divided by 17 (one cipher every 17 clock cycles instead of one cipher each clock cycle with the pipelined architecture). It means that the performance of this architecture will be very low (but it isn't important since we privilege area).

Let's go now deeper into the description of this new design.

Control unit

As mentioned before, we have a state machine of 16 states.

The control unit is in charge of those tasks:

- request a new sub key each clock cycle (signal *shift*)
- tell the multiplexers if they must load new data or take the outputs of the round (signal *load_new_pt*)
- tell the output validers to enable their outputs to the round's current outputs every 16 clock cycles (signal *output_ok*)

In reality, we need to give two pieces of information to the key processor: if we need a new key or if we need to shift once, twice or none. Everything can be coded in a single 3 bits vector (called *shift*) which can be realized as seen in this table below:

value	action
000	no shift, no new key
001	no shift, new key
010	shift once, no new key
011	shift once, new key
100	shift twice, no new key
101	shift twice, new key
"others"	error => no shift, no new key

The least significant bit (bit located in the far right) indicates if a new key is needed, while the middle bit tells if we want to shift once (1=yes, 0=no) and finally the most significant bit tells to shift twice. Values like 111, 110 are impossible (there aren't states coded with 111 or 110).

The signal *shift* will be easily decoded using a case statement in the VHDL.

So here's a rough description of the behaviour of the control component:

Init:	load a new key, shift once
Key loading:	load a new key, shift once, give the output (<i>ct</i>)
State 1:	shift once
State 2:	shift twice
State 3:	shift twice
State 4:	shift twice
State 5:	shift twice
State 6:	shift twice
State 7:	shift twice
State 8:	shift once
State 9:	shift twice
State 10:	shift twice
State 11:	shift twice
State 12:	shift twice
State 13:	shift twice
State 14:	shift twice
State 15:	shift once
State 16:	none

All the related instructions to the state are executed the next state (future state).

The state “key loading” is necessary, as the name says, to load the key before beginning the loop. It may be possible to remove it but we had to modify the key scheduling, so that it will use more logical resources. As the goal is to minimize the amount of area, we won't remove this state.

The behaviour of the control unit has been explained and we will now go on with the key processing unit.

Key processing unit

We remember that in the pipelined architecture, we had a key scheduling implemented with only wiring. This would not be the case with our small design. Actually, the key processing unit give a new sub key each clock cycle, so it needs some registers.

However, the components PC1 and PC2 remain being only wiring resources. So a new component appears: shifter. The shifter simply shifts once or twice the stored sub key. Consequently, we will have a 56 bits registers to store this key.

As we can see on the RTL schematic of the shifter on the figure 1b of the appendixes, the logical components required are:

- multiplexers
- a decoder (to decode the *shift* signal)
- registers to store the key

Fullround unit

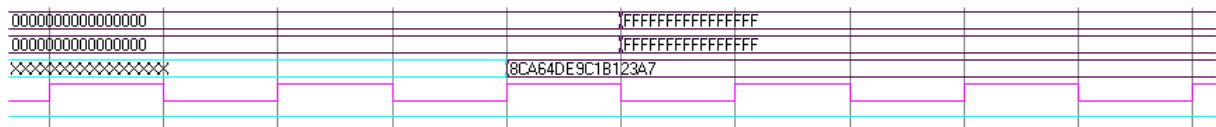
Full round is basically the name of the component which contains the round components (the same we had in the fast design) plus the two output validers. As we

already described the round component in the fast design, no further explanations are necessary.
The output validers (*ov32*) are realized using registers only.

The design of our new architecture is now ready to be tested.

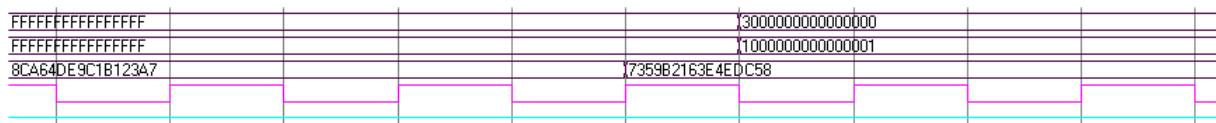
Simulation process

For the simulation, we can use the original supplied test bench without modifications, except that we need 17 clock cycles instead of 16.
After running the process, we get these results:



The first line in the figure is the plain text, the second one is the key, the third one is the cipher, the fourth one is the clock and the last one is the reset.

And at the next data:



So our design works like a charm!

We saw in the fast design that the amount of pins on our FPGA forces us to reduce the number of inputs/outputs of the design. The same remark applies here and we must reuse the converter to reduce the *pt* and the *key* to 16 bits each.

After integrating the converters, we can re-simulate our modified design. Normally, we need the required 17 clock cycles but we need now an additional 4 clock cycles but we handle this situation differently with the test bench. Of course, the first data will need 4 clock cycles to load, plus 1 for the INIT state, plus 17 for the design to work. But we can load the next data while the design is computing the cipher. So we can eliminate those 4 clock cycles and we get only 17 clock cycles instead of 21.

After having rewritten the test bench with these new considerations, we get those results:



We see clearly that the new data loading 4 clock cycles before the output (ct) is given. Whereas the fast design suffers of the additional 4 clock cycles to load the data, the small design can eliminate this problem.

Now that the design works, we are ready to synthesize.

Synthesis process

We already know how the round function will be implemented as we have the same than the fast design.

However, we must check if the key processing unit, the control unit and the outputs validers (integrated in the fullround) are implemented efficiently.

As we notice in the figure 3b in the annexes, the shifter (which is the only component in the key scheduling using logical elements) is composed of registers (called FDE) and of look-up tables (LUT).

Looking at the figure 5b we see that the control unit is made with registers (for the states) and look-up tables.

Concerning the output validers, we learn from figure 4b that these components contain only registers, as expected.

So the behaviour of our design has been efficiently implemented.

Let's take a look at the results:

	Leonardo		FPGA Compiler II	
	Estimated Clock rate	Area (CLB's)	Estimated Clock rate	Area (CLB's)
Fast design	60.7 MHz	452 (=226 slices)	156 MHz	457

We saw that the clock rate is a little less than with the fast design. This is normal, since we have a control unit, a key processing unit, output validers and some multiplexers added. The throughput is dramatically reduced:

- 228.5 Mbits/s for the design synthesized by Leonardo
- 591 Mbits/s for the design synthesized by FPGA Compiler II

There is an enormous difference with the clock rate between Leonardo and FPGA Compiler II (as with the fast design targeting Spartan II). And as previously, we will need to check the results at the place & route to have a real idea of the frequency.

Since we are focused on area with this design, here is a summary of the area used by the components.

component	Leonardo		FPGA Compiler II	
	CLB's	%	CLB's	%
Control unit	8	2%	9	2%
converter	0	0%	1	<1%
fullround	272	60%	272	60%
Key processor	172	38%	174	38%

There are some small differences of CLB's between the two synthesized designs. We also see that the key processor is using a lot of area, because of its shifter (which require storing the sub key and some multiplexers to decide how to load/unload the key).

The fullround is implemented exactly the same between the two synthesizers.

Place & Route

We can actually keep the constraint file (for allocating the pins) we wrote for the fast design.

By running the place & route (with the design synthesized by Leonardo), we can extract those final results:

	Leonardo		FPGA Compiler II	
	Clock rate	Slices	Clock rate	Slices
Fast design	79.3 MHz	313	82.6	323

Concerning Leonardo, we have a higher frequency and more area needed. FPGA compiler gets a higher frequency than Leonardo but uses more area.

Finally, the two designs are very close. The one synthesized is the best in our case.

We can compute the throughput (clock rate * 64 / 17):

- 298.5 Mbits/s for the design synthesized by Leonardo
- 311 Mbits/s for the design synthesized by FPGA Compiler II

So comparing the best fast and best small design, the area is divided by 6.3, the clock is 16% higher and the throughput is divided by 13.4!

FPGA

As with the fast design, no test could take place with the logic analyzer.

Conclusion

This project was very enriching, from the course of cryptographic engineering before the beginning of the semester, through the tests with the softwares and finally the work at the laboratory with sophisticated tools.

Despite the tests couldn't take place with the logic analyzer, this brought some new knowledge with such professional and interesting equipments.

We achieve however to two efficient designs which were working in simulation mode.

Some improvements for this project which have been discussed could be applied in a future work.

Acknowledgements

Many thanks go to Ilhan Hatirnaz, who helped me solving many problems and who was always available for me. I want also to thank Yusuf Leblebici who is a very friendly professor! Alexandre Schmid was very helpful for me, too.

I'm grateful to Alain Vachoux who gave me some advices for the VHDL files at the beginning of the project.

The students also deserve some thanks, especially Pierre Feller who never hesitated to spend some time to help me. Christian Studer was always nice and pleasant.

Finally, it was a pleasure to discover the great atmosphere of the LSM team and it was very motivating to work there.

References

Books and articles

- [1] Christof Paar, “Applied Cryptography And Data Security” (Lecture Notes), Ruhr-Universität Bochum (<http://www.crypto.ruhr-uni-bochum.de>), May 2000.
- [2] S.A. Vanstone A.J. Menezes, P.C. Oorschot, “Handbook of Applied Cryptography”, CRC Press, 1997.

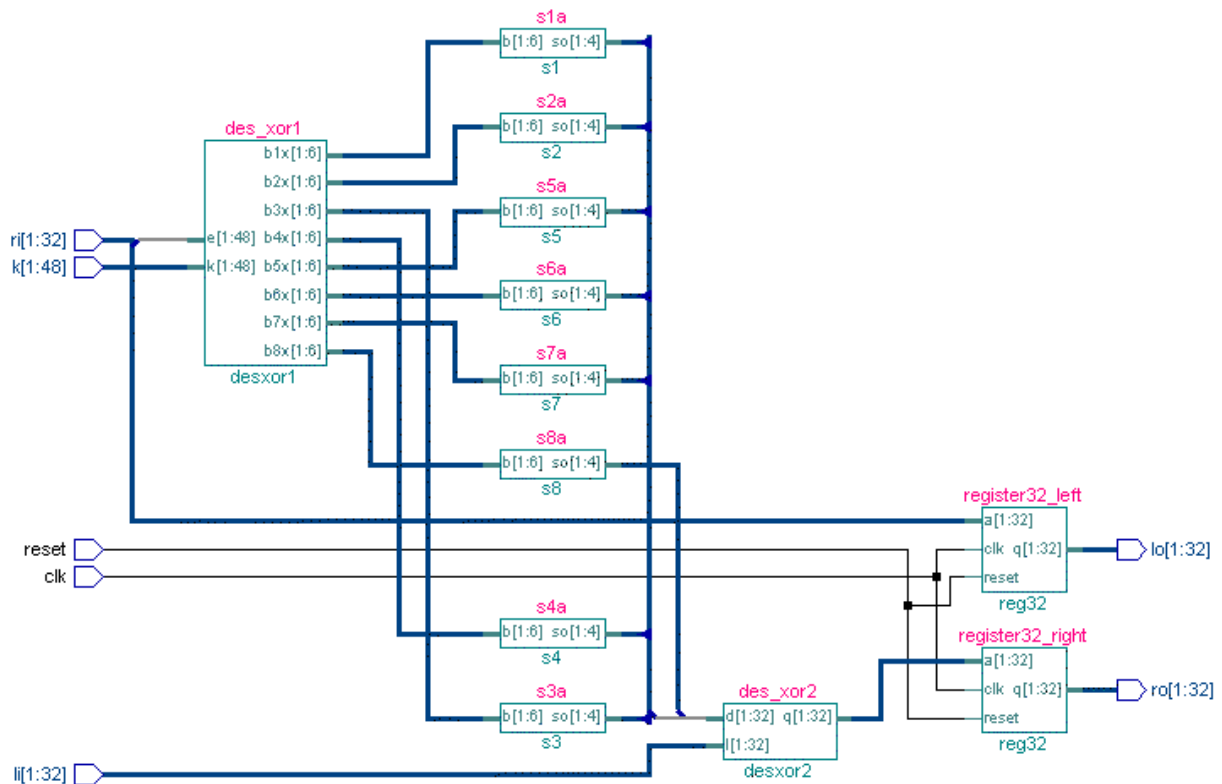
Web sites

- [3] <http://www.free-ip.com/DES/index.html> (Free-DES implemantation)
- [4] <http://www.aci.net/kalliste/des.htm> (The DES Algorithm Illustrated)
- [5] <http://mathweb.free.fr/crypto/moderne/des.php3> (La saga du DES)

Appendixes

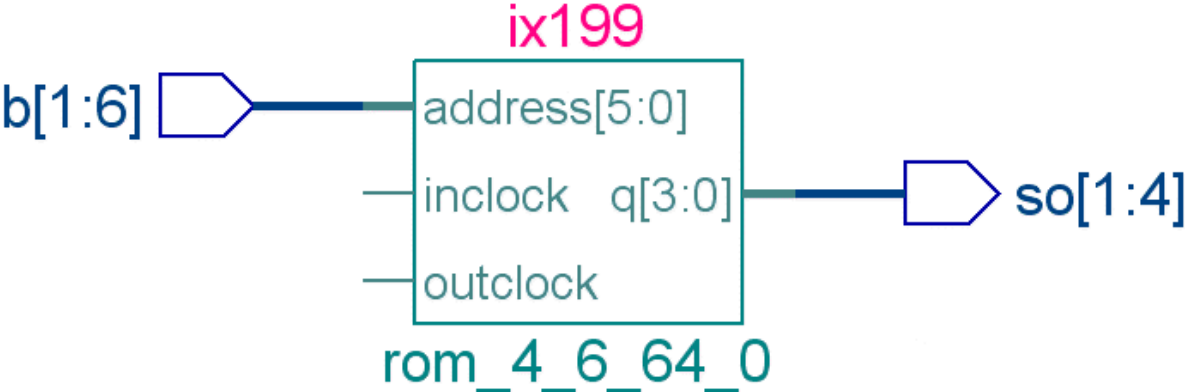
Figures for fast design

Figure 1a



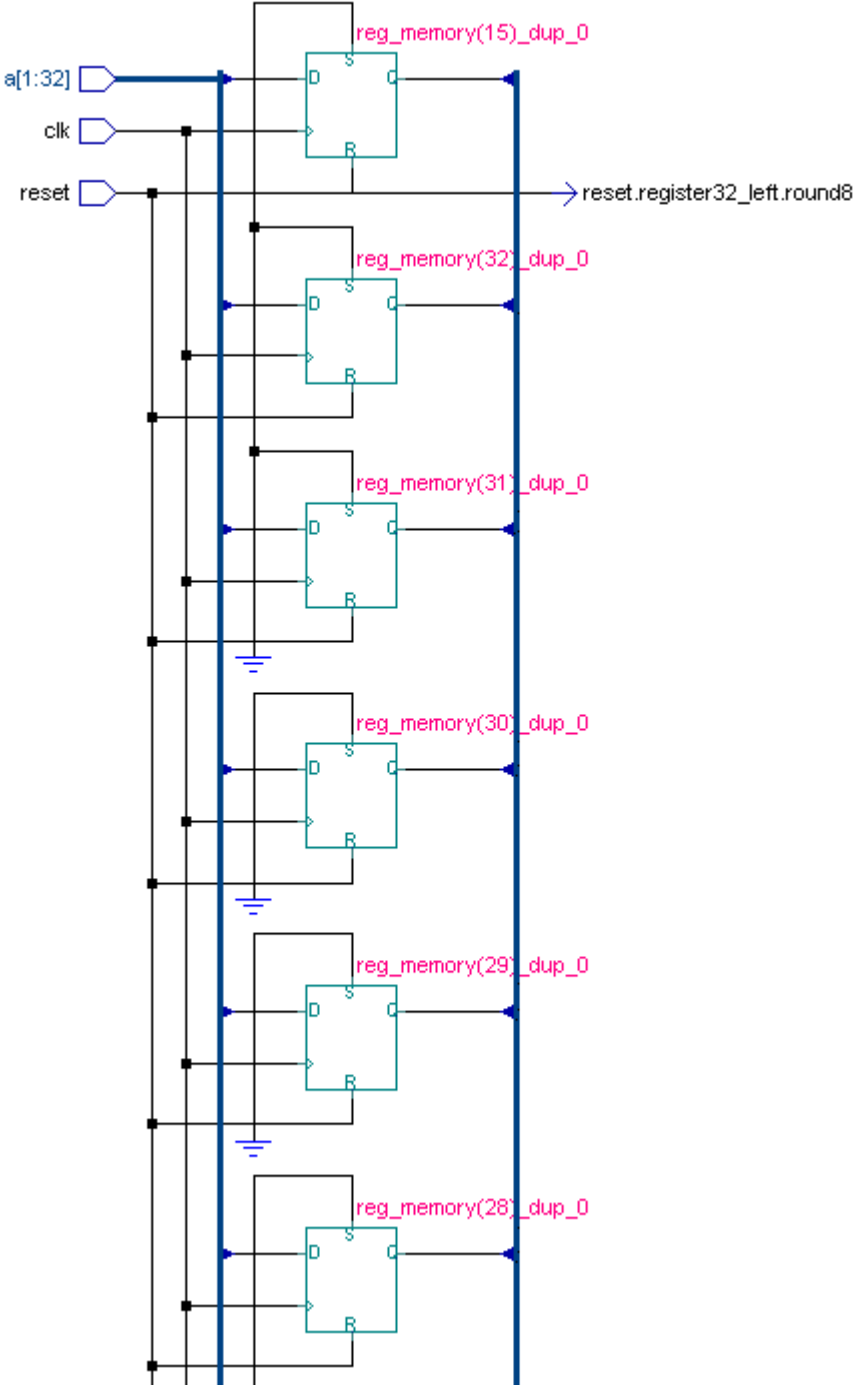
RTL schematic of one round (by Leonardo)

Figure 2a



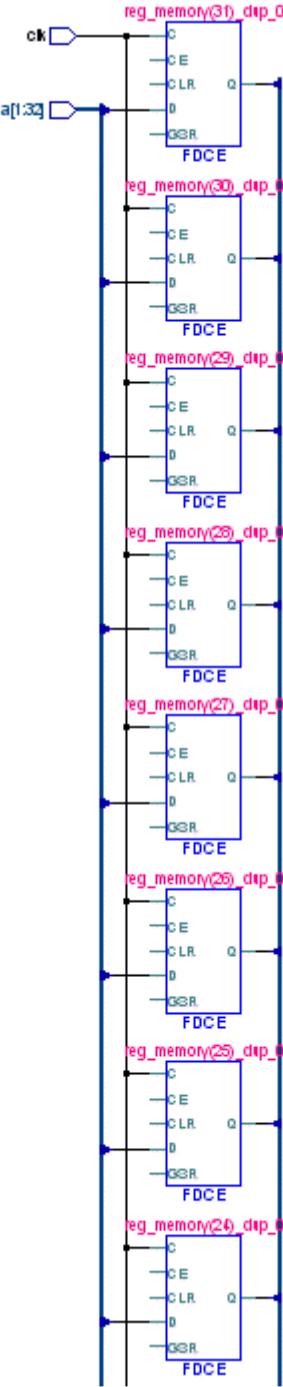
RTL schematic of one S-box (by Leonardo)

Figure 3a



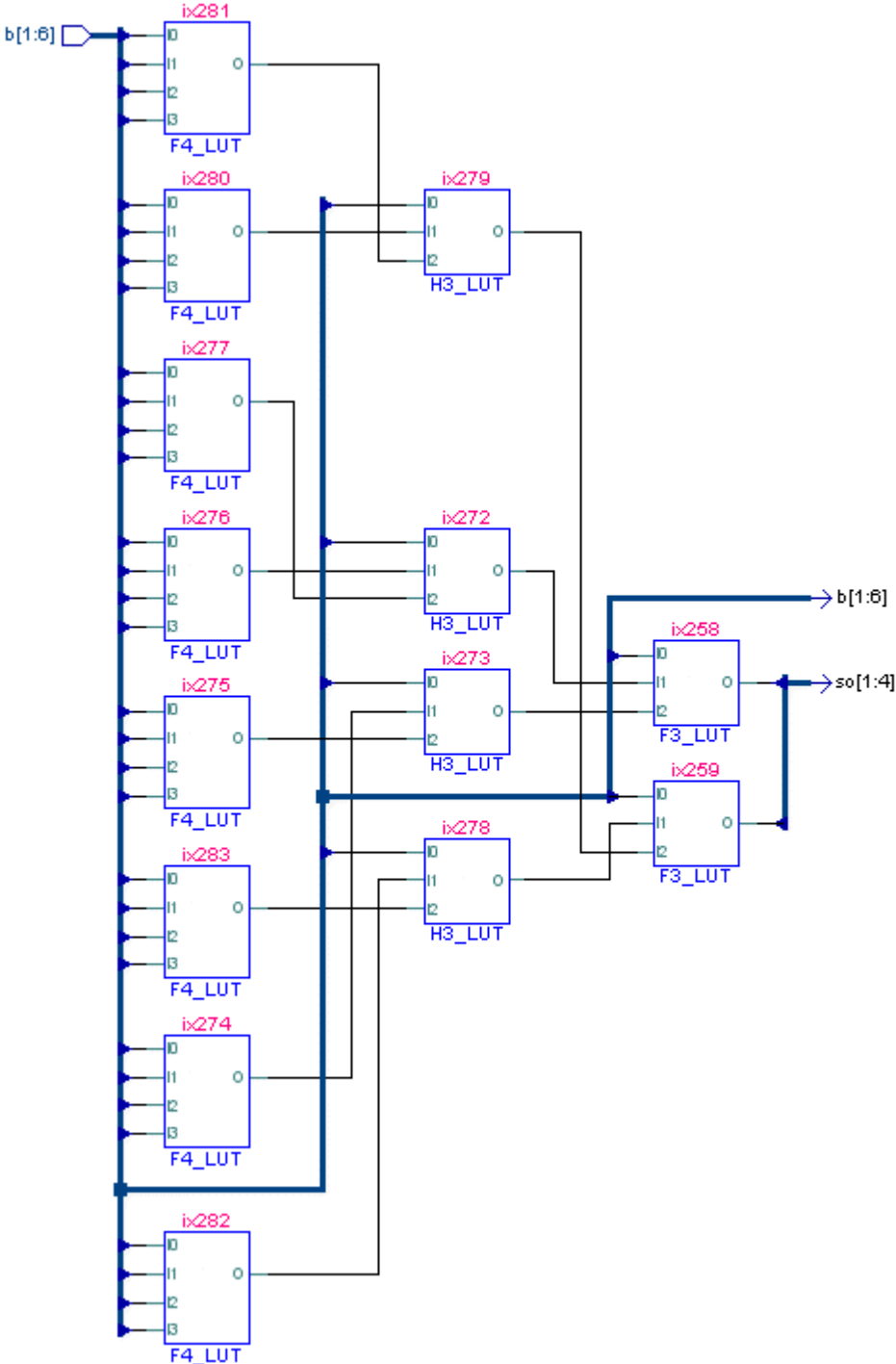
RTL schematic of reg32 (by Leonardo)

Figure 4a



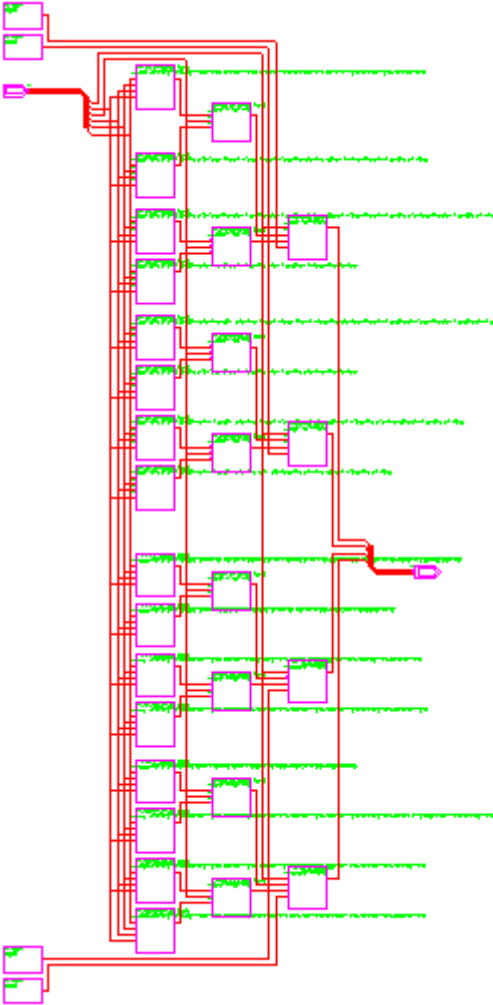
Block-level schematic view of reg32 (by Leonardo)

Figure 5a



Block-level schematic view of one S-box (by Leonardo)

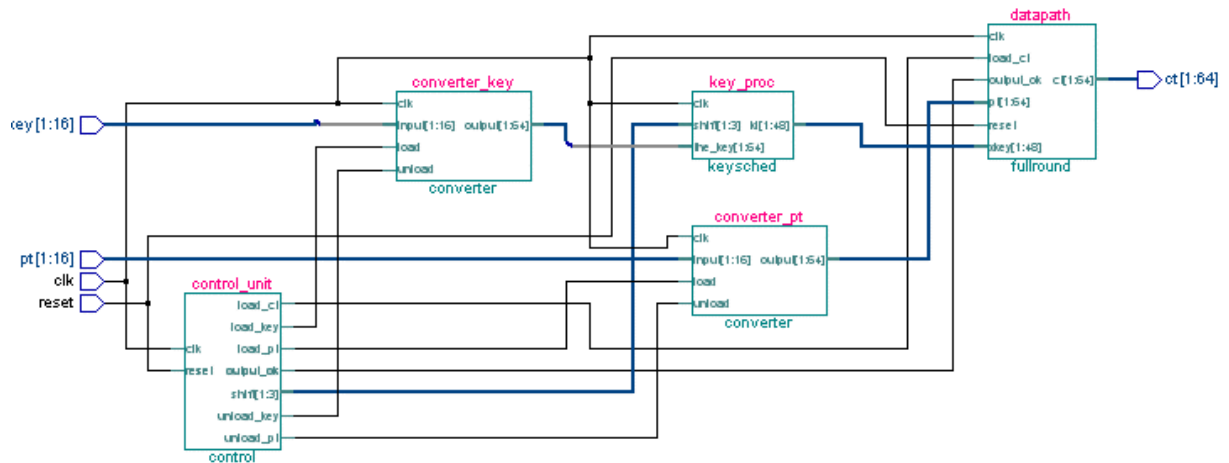
Figure 6a



Block-level schematic view of one S-box (by FPGA Compiler II)

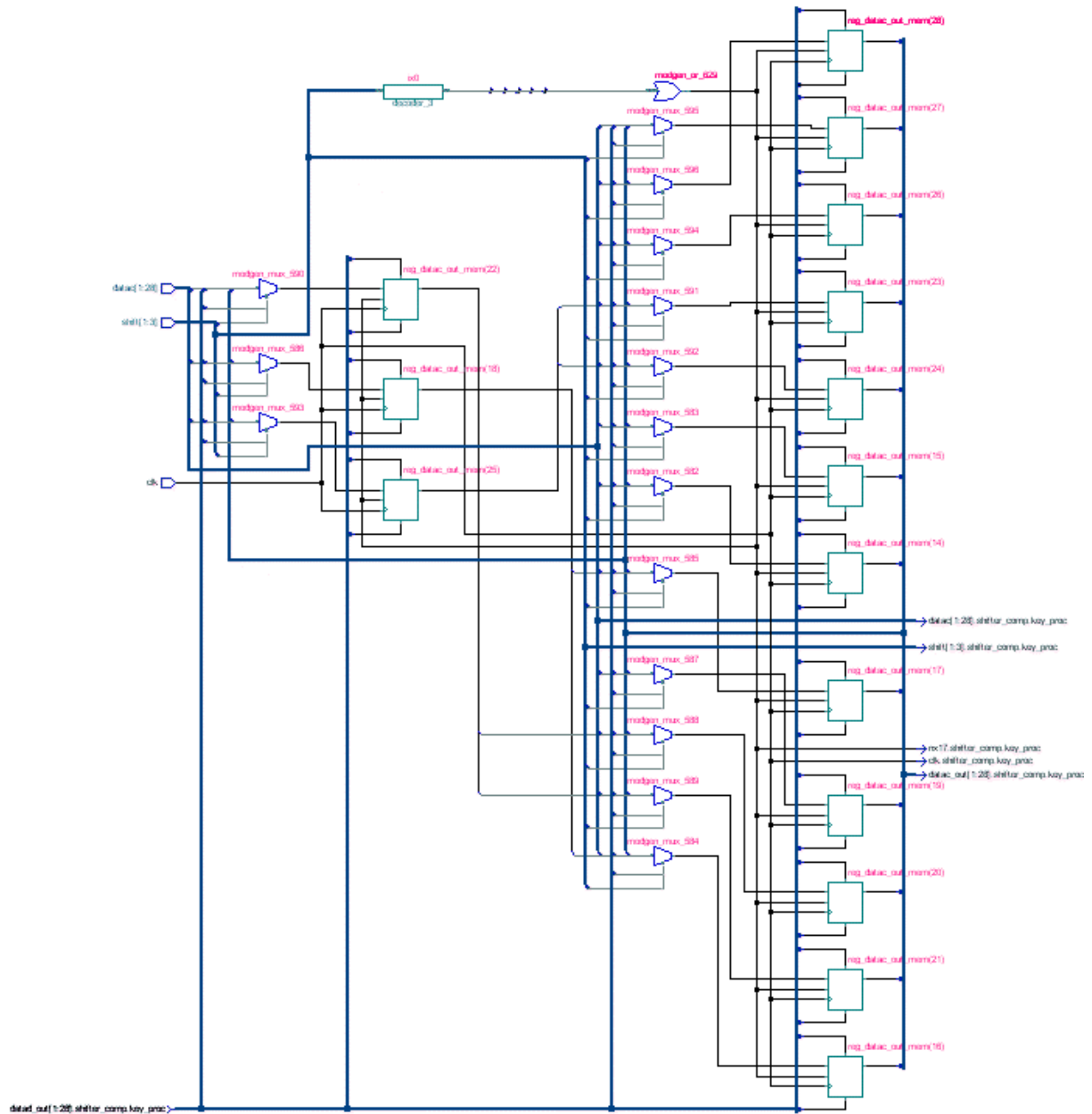
Figures for small design

Figure 1b



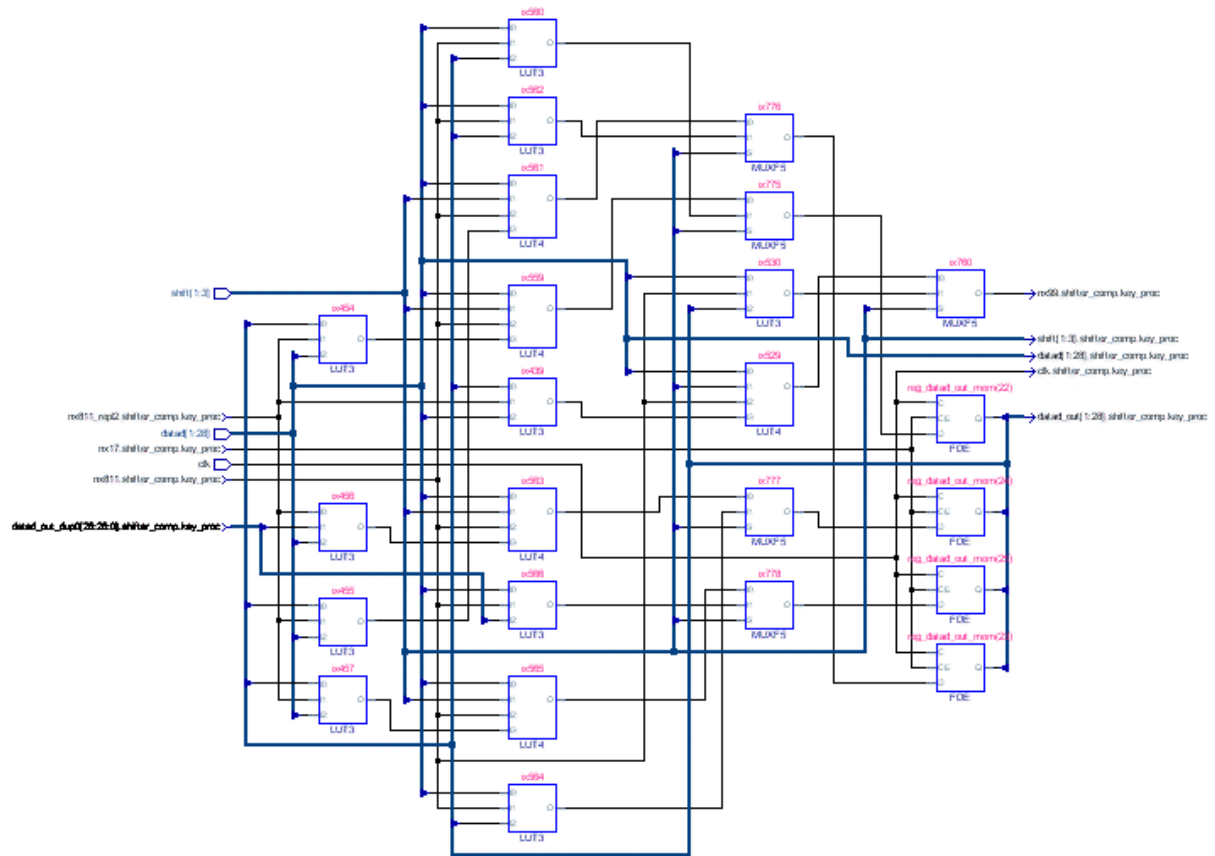
RTL schematic of the small design (by Leonardo)

Figure 2b



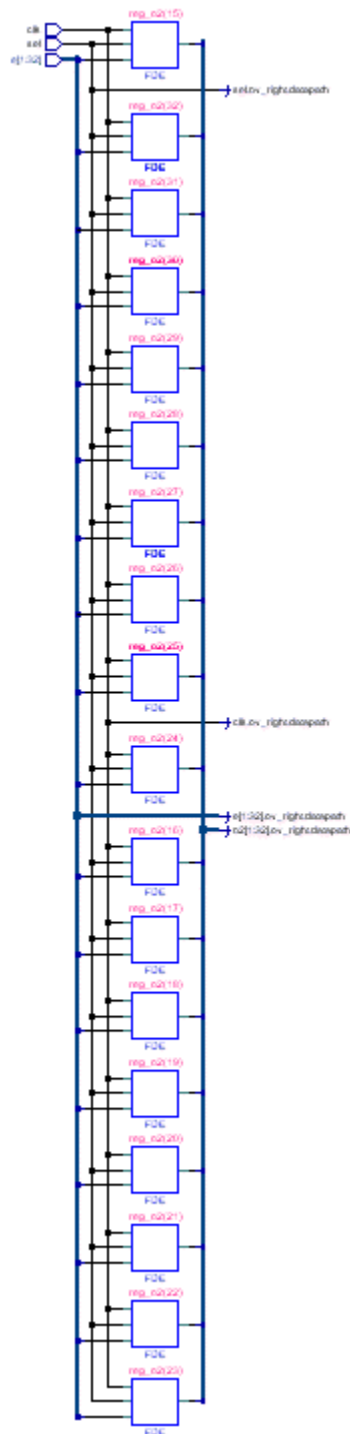
RTL schematic of shifter (by Leonardo)

Figure 3b



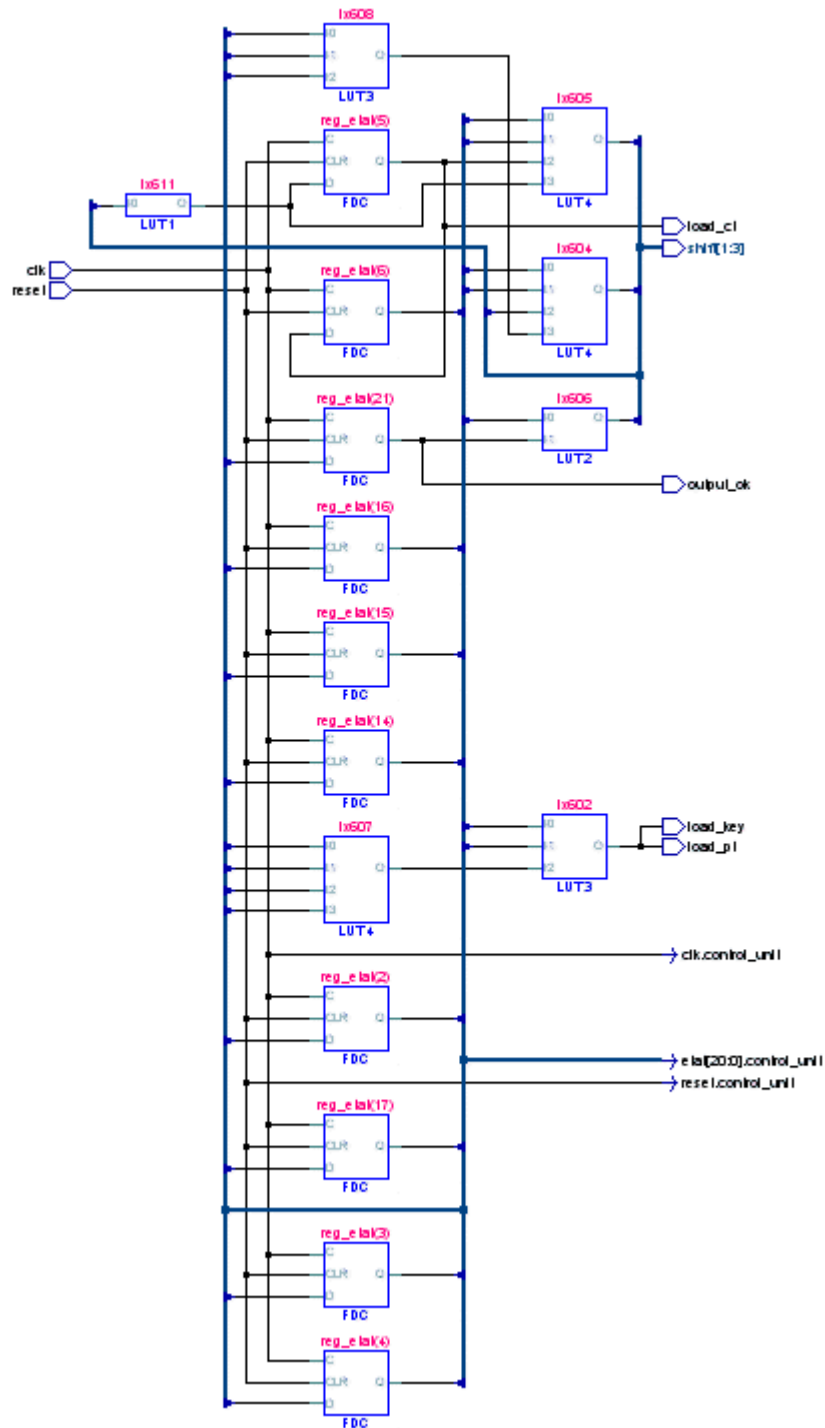
Block-level schematic view of shifter (by Leonardo)

Figure 4b



Block-level schematic view of one output valider (by Leonardo)

Figure 5b



Block-level schematic view of control unit (by Leonardo)

Excerpts of the VHDL code

Fast design

full pins version (without converters)

desenc.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity desenc is port
(
    pt      :    in    std_logic_vector(1 TO 64);
    key     :    in    std_logic_vector(1 TO 64);
    ct      :    out   std_logic_vector(1 TO 64);
    reset   :    in    std_logic;
    clk     :    in    std_logic
);
end desenc;

architecture behavior of desenc is
    signal k1x,k2x,k3x,k4x,k5x,k6x,k7x,k8x,k9x,k10x,k11x,k12x,k13x,k14x,k15x,k16x
        :    std_logic_vector(1 to 48);
    signal
    l0xa,l1x,l2x,l3x,l4x,l5x,l6x,l7x,l8x,l9x,l10x,l11x,l12x,l13x,l14x,l15x,l16x
        :    std_logic_vector(1 to 32);
    signal
    r0xa,r1x,r2x,r3x,r4x,r5x,r6x,r7x,r8x,r9x,r10x,r11x,r12x,r13x,r14x,r15x,r16x
        :    std_logic_vector(1 to 32);

    component keysched
    port (
        key     :    in    std_logic_vector(1 to 64);
        k1x,k2x,k3x,k4x,k5x,k6x,k7x,k8x,k9x,k10x,k11x,k12x,k13x,k14x,k15x,k16x
            :    out   std_logic_vector(1 to 48)
    );
    end component;

    component ip
    port (
        pt      :    in    std_logic_vector(1 TO 64);
        l0x     :    out   std_logic_vector(1 TO 32);
        r0x     :    out   std_logic_vector(1 TO 32)
    );
    end component;

    component roundfunc
    port (
        clk     :    in    std_logic;
        reset   :    in    std_logic;
        li,ri   :    in    std_logic_vector(1 to 32);
        k       :    in    std_logic_vector(1 to 48);
        lo,ro   :    out   std_logic_vector(1 to 32)
    );
```



```

end component;

component fp
port (
    l,r    :    in    std_logic_vector(1 to 32);
    ct    :    out    std_logic_vector(1 to 64)
);
end component;
begin

keyscheduling:    keysched    port map    (    key=>key,    k1x=>k1x,
k2x=>k2x,    k3x=>k3x,    k4x=>k4x,    k5x=>k5x,    k6x=>k6x,    k7x=>k7x,
k8x=>k8x,    k9x=>k9x,    k10x=>k10x,    k11x=>k11x,    k12x=>k12x,    k13x=>k13x,
k14x=>k14x,    k15x=>k15x,    k16x=>k16x    );

iperm:    ip    port map    (    pt=>pt,
l0x=>l0xa,    r0x=>r0xa    );

round1:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l0xa,    ri=>r0xa,    k=>k1x,    lo=>l1x,    ro=>r1x    );
round2:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l1x,    ri=>r1x,    k=>k2x,    lo=>l2x,    ro=>r2x    );
round3:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l2x,    ri=>r2x,    k=>k3x,    lo=>l3x,    ro=>r3x    );
round4:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l3x,    ri=>r3x,    k=>k4x,    lo=>l4x,    ro=>r4x    );
round5:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l4x,    ri=>r4x,    k=>k5x,    lo=>l5x,    ro=>r5x    );
round6:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l5x,    ri=>r5x,    k=>k6x,    lo=>l6x,    ro=>r6x    );
round7:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l6x,    ri=>r6x,    k=>k7x,    lo=>l7x,    ro=>r7x    );
round8:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l7x,    ri=>r7x,    k=>k8x,    lo=>l8x,    ro=>r8x    );
round9:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l8x,    ri=>r8x,    k=>k9x,    lo=>l9x,    ro=>r9x    );
round10:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l9x,    ri=>r9x,    k=>k10x,    lo=>l10x,    ro=>r10x    );
round11:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l10x,    ri=>r10x,    k=>k11x,    lo=>l11x,    ro=>r11x    );
round12:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l11x,    ri=>r11x,    k=>k12x,    lo=>l12x,    ro=>r12x    );
round13:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l12x,    ri=>r12x,    k=>k13x,    lo=>l13x,    ro=>r13x    );
round14:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l13x,    ri=>r13x,    k=>k14x,    lo=>l14x,    ro=>r14x    );
round15:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l14x,    ri=>r14x,    k=>k15x,    lo=>l15x,    ro=>r15x    );
round16:    roundfunc    port map    (    clk=>clk,    reset=>reset,
li=>l15x,    ri=>r15x,    k=>k16x,    lo=>l16x,    ro=>r16x    );

fperm: fp    port map    (    l=>r16x,    r=>l16x,    ct=>ct );

end behavior;

```

ip.vhd

```

library ieee;

use ieee.std_logic_1164.all;

entity ip is port
(

```

```

    pt      :      in std_logic_vector(1 TO 64);
    l0x     :      out std_logic_vector(1 TO 32);
    r0x     :      out std_logic_vector(1 TO 32)
);
end ip;

architecture behavior of ip is
begin

l0x(1)<=pt(58);          l0x(2)<=pt(50);          l0x(3)<=pt(42);
    l0x(4)<=pt(34);
l0x(5)<=pt(26);          l0x(6)<=pt(18);          l0x(7)<=pt(10);
    l0x(8)<=pt(2);
l0x(9)<=pt(60);          l0x(10)<=pt(52);          l0x(11)<=pt(44);          l0x(12)<=pt(36);
l0x(13)<=pt(28);          l0x(14)<=pt(20);          l0x(15)<=pt(12);          l0x(16)<=pt(4);
l0x(17)<=pt(62);          l0x(18)<=pt(54);          l0x(19)<=pt(46);          l0x(20)<=pt(38);
l0x(21)<=pt(30);          l0x(22)<=pt(22);          l0x(23)<=pt(14);          l0x(24)<=pt(6);
l0x(25)<=pt(64);          l0x(26)<=pt(56);          l0x(27)<=pt(48);          l0x(28)<=pt(40);
l0x(29)<=pt(32);          l0x(30)<=pt(24);          l0x(31)<=pt(16);          l0x(32)<=pt(8);

r0x(1)<=pt(57);          r0x(2)<=pt(49);          r0x(3)<=pt(41);
    r0x(4)<=pt(33);
r0x(5)<=pt(25);          r0x(6)<=pt(17);          r0x(7)<=pt(9);
    r0x(8)<=pt(1);
r0x(9)<=pt(59);          r0x(10)<=pt(51);          r0x(11)<=pt(43);          r0x(12)<=pt(35);
r0x(13)<=pt(27);          r0x(14)<=pt(19);          r0x(15)<=pt(11);          r0x(16)<=pt(3);
r0x(17)<=pt(61);          r0x(18)<=pt(53);          r0x(19)<=pt(45);          r0x(20)<=pt(37);
r0x(21)<=pt(29);          r0x(22)<=pt(21);          r0x(23)<=pt(13);          r0x(24)<=pt(5);
r0x(25)<=pt(63);          r0x(26)<=pt(55);          r0x(27)<=pt(47);          r0x(28)<=pt(39);
r0x(29)<=pt(31);          r0x(30)<=pt(23);          r0x(31)<=pt(15);          r0x(32)<=pt(7);

end behavior;

```

fp.vhd

```

library ieee;

use ieee.std_logic_1164.all;

entity fp is port
(
    l,r      :      in      std_logic_vector(1 to 32);
    ct      :      out      std_logic_vector(1 to 64)
);
end fp;

architecture behaviour of fp is
begin

ct(1)<=r(8); ct(2)<=l(8); ct(3)<=r(16);ct(4)<=l(16);ct(5)<=r(24);ct(6)<=l(24);
    ct(7)<=r(32);ct(8)<=l(32);
ct(9)<=r(7); ct(10)<=l(7);ct(11)<=r(15);          ct(12)<=l(15);          ct(13)<=r(23);
    ct(14)<=l(23);          ct(15)<=r(31);          ct(16)<=l(31);
ct(17)<=r(6);ct(18)<=l(6);ct(19)<=r(14);          ct(20)<=l(14);          ct(21)<=r(22);
    ct(22)<=l(22);          ct(23)<=r(30);          ct(24)<=l(30);
ct(25)<=r(5);ct(26)<=l(5);ct(27)<=r(13);          ct(28)<=l(13);          ct(29)<=r(21);
    ct(30)<=l(21);          ct(31)<=r(29);          ct(32)<=l(29);
ct(33)<=r(4);ct(34)<=l(4);ct(35)<=r(12);          ct(36)<=l(12);          ct(37)<=r(20);
    ct(38)<=l(20);          ct(39)<=r(28);          ct(40)<=l(28);
ct(41)<=r(3);ct(42)<=l(3);ct(43)<=r(11);          ct(44)<=l(11);          ct(45)<=r(19);
    ct(46)<=l(19);          ct(47)<=r(27);          ct(48)<=l(27);
ct(49)<=r(2);ct(50)<=l(2);ct(51)<=r(10);          ct(52)<=l(10);          ct(53)<=r(18);
    ct(54)<=l(18);          ct(55)<=r(26);          ct(56)<=l(26);
ct(57)<=r(1);ct(58)<=l(1);ct(59)<=r(9);ct(60)<=l(9);ct(61)<=r(17);
    ct(62)<=l(17);          ct(63)<=r(25);          ct(64)<=l(25);

```

end;

keysched.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity keysched is port
(
    key      :    in    std_logic_vector(1 to 64);
    k1x,k2x,k3x,k4x,k5x,k6x,k7x,k8x,k9x,k10x,k11x,k12x,k13x,k14x,k15x,k16x
      :    out   std_logic_vector(1 to 48)
);
end keysched;

architecture behaviour of keysched is
    signal
c0x,c1x,c2x,c3x,c4x,c5x,c6x,c7x,c8x,c9x,c10x,c11x,c12x,c13x,c14x,c15x,c16x      :
    std_logic_vector(1 to 28);
    signal
d0x,d1x,d2x,d3x,d4x,d5x,d6x,d7x,d8x,d9x,d10x,d11x,d12x,d13x,d14x,d15x,d16x    :
    std_logic_vector(1 to 28);

    component pc1
    port (
        key      :    in std_logic_vector(1 TO 64);
        c0x,d0x  :    out std_logic_vector(1 TO 28)
    );
    end component;

    component pc2
    port (
        c,d      : in std_logic_vector(1 TO 28);
        k        : out std_logic_vector(1 TO 48)
    );
    end component;

begin

    pc_1: pc1 port map ( key=>key, c0x=>c0x, d0x=>d0x );

    c1x<=To_StdLogicVector(to_bitvector(c0x) rol 1);
    d1x<=To_StdLogicVector(to_bitvector(d0x) rol 1);
    c2x<=To_StdLogicVector(to_bitvector(c1x) rol 1);
    d2x<=To_StdLogicVector(to_bitvector(d1x) rol 1);
    c3x<=To_StdLogicVector(to_bitvector(c2x) rol 2);
    d3x<=To_StdLogicVector(to_bitvector(d2x) rol 2);
    c4x<=To_StdLogicVector(to_bitvector(c3x) rol 2);
    d4x<=To_StdLogicVector(to_bitvector(d3x) rol 2);
    c5x<=To_StdLogicVector(to_bitvector(c4x) rol 2);
    d5x<=To_StdLogicVector(to_bitvector(d4x) rol 2);
    c6x<=To_StdLogicVector(to_bitvector(c5x) rol 2);
    d6x<=To_StdLogicVector(to_bitvector(d5x) rol 2);
    c7x<=To_StdLogicVector(to_bitvector(c6x) rol 2);
    d7x<=To_StdLogicVector(to_bitvector(d6x) rol 2);
    c8x<=To_StdLogicVector(to_bitvector(c7x) rol 2);
    d8x<=To_StdLogicVector(to_bitvector(d7x) rol 2);
    c9x<=To_StdLogicVector(to_bitvector(c8x) rol 1);
    d9x<=To_StdLogicVector(to_bitvector(d8x) rol 1);
    c10x<=To_StdLogicVector(to_bitvector(c9x) rol 2);
    d10x<=To_StdLogicVector(to_bitvector(d9x) rol 2);
```

```
c11x<=To_StdLogicVector(to_bitvector(c10x) rol 2);
d11x<=To_StdLogicVector(to_bitvector(d10x) rol 2);
c12x<=To_StdLogicVector(to_bitvector(c11x) rol 2);
d12x<=To_StdLogicVector(to_bitvector(d11x) rol 2);
c13x<=To_StdLogicVector(to_bitvector(c12x) rol 2);
d13x<=To_StdLogicVector(to_bitvector(d12x) rol 2);
c14x<=To_StdLogicVector(to_bitvector(c13x) rol 2);
d14x<=To_StdLogicVector(to_bitvector(d13x) rol 2);
c15x<=To_StdLogicVector(to_bitvector(c14x) rol 2);
d15x<=To_StdLogicVector(to_bitvector(d14x) rol 2);
c16x<=To_StdLogicVector(to_bitvector(c15x) rol 1);
d16x<=To_StdLogicVector(to_bitvector(d15x) rol 1);

pc2x1: pc2 port map (      c=>c1x,          d=>d1x,          k=>k1x );
pc2x2: pc2 port map (      c=>c2x,          d=>d2x,          k=>k2x );
pc2x3: pc2 port map (      c=>c3x,          d=>d3x,          k=>k3x );
pc2x4: pc2 port map (      c=>c4x,          d=>d4x,          k=>k4x );
pc2x5: pc2 port map (      c=>c5x,          d=>d5x,          k=>k5x );
pc2x6: pc2 port map (      c=>c6x,          d=>d6x,          k=>k6x );
pc2x7: pc2 port map (      c=>c7x,          d=>d7x,          k=>k7x );
pc2x8: pc2 port map (      c=>c8x,          d=>d8x,          k=>k8x );
pc2x9: pc2 port map (      c=>c9x,          d=>d9x,          k=>k9x );
pc2x10: pc2 port map (      c=>c10x,         d=>d10x,         k=>k10x );
pc2x11: pc2 port map (      c=>c11x,         d=>d11x,         k=>k11x );
pc2x12: pc2 port map (      c=>c12x,         d=>d12x,         k=>k12x );
pc2x13: pc2 port map (      c=>c13x,         d=>d13x,         k=>k13x );
pc2x14: pc2 port map (      c=>c14x,         d=>d14x,         k=>k14x );
pc2x15: pc2 port map (      c=>c15x,         d=>d15x,         k=>k15x );
pc2x16: pc2 port map (      c=>c16x,         d=>d16x,         k=>k16x );
```

end behaviour;

roundfunc.vhd

```
library ieee;
```

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

```
entity roundfunc is port
```

```
(
    clk          :    in    std_logic;
    reset        :    in    std_logic;
    li,ri       :    in    std_logic_vector(1 to 32);
    k            :    in    std_logic_vector(1 to 48);
    lo,ro       :    out   std_logic_vector(1 to 32)
);
```

```
end roundfunc;
```

```
architecture behaviour of roundfunc is
```

```
    signal xp_to_xor      :    std_logic_vector(1 to 48);
    signal blx,b2x,b3x,b4x,b5x,b6x,b7x,b8x
        :    std_logic_vector(1 to 6);
    signal solx,so2x,so3x,so4x,so5x,so6x,so7x,so8x
        :    std_logic_vector(1 to 4);
    signal ppo,r_toreg32,l_toreg32 :    std_logic_vector(1 to 32);
```

```
    component xp
```

```
    port (
        ri      : in std_logic_vector(1 TO 32);
        e       : out std_logic_vector(1 TO 48)
    );
```

```
end component;
```

```
component desxor1
port (
    e      :      in std_logic_vector(1 TO 48);
    b1x,b2x,b3x,b4x,b5x,b6x,b7x,b8x
    :      out std_logic_vector (1 TO 6);
    k      :      in std_logic_vector (1 TO 48)
);
end component;

component s1
port (
    b      :      in   std_logic_vector(1 to 6);
    so     :      out  std_logic_vector(1 to 4)
);
end component;

component s2
port (
    b      :      in   std_logic_vector(1 to 6);
    so     :      out  std_logic_vector(1 to 4)
);
end component;

component s3
port (
    b      :      in   std_logic_vector(1 to 6);
    so     :      out  std_logic_vector(1 to 4)
);
end component;

component s4
port (
    b      :      in   std_logic_vector(1 to 6);
    so     :      out  std_logic_vector(1 to 4)
);
end component;

component s5
port (
    b      :      in   std_logic_vector(1 to 6);
    so     :      out  std_logic_vector(1 to 4)
);
end component;

component s6
port (
    b      :      in   std_logic_vector(1 to 6);
    so     :      out  std_logic_vector(1 to 4)
);
end component;

component s7
port (
    b      :      in   std_logic_vector(1 to 6);
    so     :      out  std_logic_vector(1 to 4)
);
end component;

component s8
port (
    b      :      in   std_logic_vector(1 to 6);
    so     :      out  std_logic_vector(1 to 4)
);
end component;

component pp
```

```

    port (
        solx,so2x,so3x,so4x,so5x,so6x,so7x,so8x
        :   in   std_logic_vector(1 to 4);
        ppo  :   out  std_logic_vector(1 to 32)
    );
end component;

component desxor2
port (
    d,l    :   in   std_logic_vector(1 to 32);
    q      :   out  std_logic_vector(1 to 32)
);
end component;

component reg32
port (
    a  : in   std_logic_vector (1 to 32);
    q  : out  std_logic_vector (1 to 32);
    reset :   in   std_logic;
    clk : in   std_logic
);
end component;
begin

    xpension:          xp          port map      (      ri=>ri,
    e=>xp_to_xor );

    des_xor1:          desxor1          port map      (      e=>xp_to_xor,k=>k,
    b1x=>b1x,    b2x=>b2x,    b3x=>b3x,    b4x=>b4x,    b5x=>b5x,    b6x=>b6x,
    b7x=>b7x,    b8x=>b8x      );

    s1a:              s1          port map      (      b=>b1x,    so=>so1x
    );
    s2a:              s2          port map      (      b=>b2x,    so=>so2x
    );
    s3a:              s3          port map      (      b=>b3x,    so=>so3x
    );
    s4a:              s4          port map      (      b=>b4x,    so=>so4x
    );
    s5a:              s5          port map      (      b=>b5x,    so=>so5x
    );
    s6a:              s6          port map      (      b=>b6x,    so=>so6x
    );
    s7a:              s7          port map      (      b=>b7x,    so=>so7x
    );
    s8a:              s8          port map      (      b=>b8x,    so=>so8x
    );

    pperm:            pp          port map      (      solx=>solx,  so2x=>so2x,
    so3x=>so3x,  so4x=>so4x,  so5x=>so5x,  so6x=>so6x,  so7x=>so7x,  so8x=>so8x,
    ppo=>ppo      );

    des_xor2:          desxor2          port map      (      d=>ppo,
    l=>li, q=>r_toreg32 );
    l_toreg32<=ri;

    register32_left:  reg32          port map      (      a=>l_toreg32, q=>lo,
    reset=>reset, clk=>clk );
    register32_right: reg32          port map      (      a=>r_toreg32, q=>ro,
    reset=>reset, clk=>clk );

end;

```

s1.vhd

```
library ieee;
```

```
use ieee.std_logic_1164.all;

entity s1 is port
(
    b      :    in    std_logic_vector(1 to 6);
    so     :    out   std_logic_vector(1 to 4)
);
end s1;

architecture behaviour of s1 is
begin

process(b)
begin
case b is
when "000000"=>    so<=To_StdLogicVector(Bit_Vector'('x"e"));
when "000010"=>    so<=To_StdLogicVector(Bit_Vector'('x"4"));
when "000100"=>    so<=To_StdLogicVector(Bit_Vector'('x"d"));
when "000110"=>    so<=To_StdLogicVector(Bit_Vector'('x"1"));
when "001000"=>    so<=To_StdLogicVector(Bit_Vector'('x"2"));
when "001010"=>    so<=To_StdLogicVector(Bit_Vector'('x"f"));
when "001100"=>    so<=To_StdLogicVector(Bit_Vector'('x"b"));
when "001110"=>    so<=To_StdLogicVector(Bit_Vector'('x"8"));
when "010000"=>    so<=To_StdLogicVector(Bit_Vector'('x"3"));
when "010010"=>    so<=To_StdLogicVector(Bit_Vector'('x"a"));
when "010100"=>    so<=To_StdLogicVector(Bit_Vector'('x"6"));
when "010110"=>    so<=To_StdLogicVector(Bit_Vector'('x"c"));
when "011000"=>    so<=To_StdLogicVector(Bit_Vector'('x"5"));
when "011010"=>    so<=To_StdLogicVector(Bit_Vector'('x"9"));
when "011100"=>    so<=To_StdLogicVector(Bit_Vector'('x"0"));
when "011110"=>    so<=To_StdLogicVector(Bit_Vector'('x"7"));
when "000001"=>    so<=To_StdLogicVector(Bit_Vector'('x"0"));
when "000011"=>    so<=To_StdLogicVector(Bit_Vector'('x"f"));
when "000101"=>    so<=To_StdLogicVector(Bit_Vector'('x"7"));
when "000111"=>    so<=To_StdLogicVector(Bit_Vector'('x"4"));
when "001001"=>    so<=To_StdLogicVector(Bit_Vector'('x"e"));
when "001011"=>    so<=To_StdLogicVector(Bit_Vector'('x"2"));
when "001101"=>    so<=To_StdLogicVector(Bit_Vector'('x"d"));
when "001111"=>    so<=To_StdLogicVector(Bit_Vector'('x"1"));
when "010001"=>    so<=To_StdLogicVector(Bit_Vector'('x"a"));
when "010011"=>    so<=To_StdLogicVector(Bit_Vector'('x"6"));
when "010101"=>    so<=To_StdLogicVector(Bit_Vector'('x"c"));
when "010111"=>    so<=To_StdLogicVector(Bit_Vector'('x"b"));
when "011001"=>    so<=To_StdLogicVector(Bit_Vector'('x"9"));
when "011011"=>    so<=To_StdLogicVector(Bit_Vector'('x"5"));
when "011101"=>    so<=To_StdLogicVector(Bit_Vector'('x"3"));
when "011111"=>    so<=To_StdLogicVector(Bit_Vector'('x"8"));
when "100000"=>    so<=To_StdLogicVector(Bit_Vector'('x"4"));
when "100010"=>    so<=To_StdLogicVector(Bit_Vector'('x"1"));
when "100100"=>    so<=To_StdLogicVector(Bit_Vector'('x"e"));
when "100110"=>    so<=To_StdLogicVector(Bit_Vector'('x"8"));
when "101000"=>    so<=To_StdLogicVector(Bit_Vector'('x"d"));
when "101010"=>    so<=To_StdLogicVector(Bit_Vector'('x"6"));
when "101100"=>    so<=To_StdLogicVector(Bit_Vector'('x"2"));
when "101110"=>    so<=To_StdLogicVector(Bit_Vector'('x"b"));
when "110000"=>    so<=To_StdLogicVector(Bit_Vector'('x"f"));
when "110010"=>    so<=To_StdLogicVector(Bit_Vector'('x"c"));
when "110100"=>    so<=To_StdLogicVector(Bit_Vector'('x"9"));
when "110110"=>    so<=To_StdLogicVector(Bit_Vector'('x"7"));
when "111000"=>    so<=To_StdLogicVector(Bit_Vector'('x"3"));
when "111010"=>    so<=To_StdLogicVector(Bit_Vector'('x"a"));
when "111100"=>    so<=To_StdLogicVector(Bit_Vector'('x"5"));
when "111110"=>    so<=To_StdLogicVector(Bit_Vector'('x"0"));
when "100001"=>    so<=To_StdLogicVector(Bit_Vector'('x"f"));
when "100011"=>    so<=To_StdLogicVector(Bit_Vector'('x"c"));
when "100101"=>    so<=To_StdLogicVector(Bit_Vector'('x"8"));

```

```

        when "100111"=>      so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "101001"=>      so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "101011"=>      so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "101101"=>      so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "101111"=>      so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "110001"=>      so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "110011"=>      so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "110101"=>      so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "110111"=>      so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "111001"=>      so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "111011"=>      so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "111101"=>      so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when others=>        so<=To_StdLogicVector(Bit_Vector'(x"d"));
    end case;
end process;
end;
```

xp.vhd

```

library ieee;

use ieee.std_logic_1164.all;

entity xp is port
(
    ri      : in std_logic_vector(1 TO 32);
    e       : out std_logic_vector(1 TO 48));
end xp;

architecture behavior of xp is
begin
    e(1)<=ri(32);e(2)<=ri(1); e(3)<=ri(2); e(4)<=ri(3); e(5)<=ri(4);
    e(6)<=ri(5); e(7)<=ri(4); e(8)<=ri(5);
    e(9)<=ri(6); e(10)<=ri(7);e(11)<=ri(8);e(12)<=ri(9);e(13)<=ri(8);
    e(14)<=ri(9);e(15)<=ri(10);      e(16)<=ri(11);
    e(17)<=ri(12);      e(18)<=ri(13);      e(19)<=ri(12);      e(20)<=ri(13);
    e(21)<=ri(14);      e(22)<=ri(15);      e(23)<=ri(16);      e(24)<=ri(17);
    e(25)<=ri(16);      e(26)<=ri(17);      e(27)<=ri(18);      e(28)<=ri(19);
    e(29)<=ri(20);      e(30)<=ri(21);      e(31)<=ri(20);      e(32)<=ri(21);
    e(33)<=ri(22);      e(34)<=ri(23);      e(35)<=ri(24);      e(36)<=ri(25);
    e(37)<=ri(24);      e(38)<=ri(25);      e(39)<=ri(26);      e(40)<=ri(27);
    e(41)<=ri(28);      e(42)<=ri(29);      e(43)<=ri(28);      e(44)<=ri(29);
    e(45)<=ri(30);      e(46)<=ri(31);      e(47)<=ri(32);      e(48)<=ri(1);
end behavior;
```

desxor1.vhd

```

library ieee;

use ieee.std_logic_1164.all;

entity desxor1 is port
(
    e       : in std_logic_vector(1 TO 48);
    blx,b2x,b3x,b4x,b5x,b6x,b7x,b8x
    :       out std_logic_vector (1 TO 6);
    k       : in std_logic_vector (1 TO 48)
);
end desxor1;

architecture behavior of desxor1 is
    signal XX      : std_logic_vector( 1 to 48);
begin
    XX<=k xor e;
    blx<=XX(1 to 6);
```



```
        b2x<=XX(7 to 12);
        b3x<=XX(13 to 18);
        b4x<=XX(19 to 24);
        b5x<=XX(25 to 30);
        b6x<=XX(31 to 36);
        b7x<=XX(37 to 42);
        b8x<=XX(43 to 48);
end behavior;
```

desxor2.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity desxor2 is port
(
    d,l    :    in    std_logic_vector(1 to 32);
    q      :    out   std_logic_vector(1 to 32)
);
end desxor2;

architecture behaviour of desxor2 is
begin
    q<=d xor l;
end;
```

pc1.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity pc1 is port
(
    key      :    in std_logic_vector(1 TO 64);
    c0x,d0x  :    out std_logic_vector(1 TO 28)
);
end pc1;

architecture behavior of pc1 is
    signal XX    :    std_logic_vector(1 to 56);
begin
    XX(1)<=key(57);      XX(2)<=key(49);      XX(3)<=key(41);
    XX(4)<=key(33);      XX(5)<=key(25);      XX(6)<=key(17);
    XX(7)<=key(9);
    XX(8)<=key(1);      XX(9)<=key(58);      XX(10)<=key(50);
    XX(11)<=key(42);     XX(12)<=key(34);     XX(13)<=key(26);     XX(14)<=key(18);
    XX(15)<=key(10);     XX(16)<=key(2);      XX(17)<=key(59);
    XX(18)<=key(51);     XX(19)<=key(43);     XX(20)<=key(35);     XX(21)<=key(27);
    XX(22)<=key(19);     XX(23)<=key(11);     XX(24)<=key(3);
    XX(25)<=key(60);     XX(26)<=key(52);     XX(27)<=key(44);     XX(28)<=key(36);
    XX(29)<=key(63);     XX(30)<=key(55);     XX(31)<=key(47);     XX(32)<=key(39);
    XX(33)<=key(31);     XX(34)<=key(23);     XX(35)<=key(15);
    XX(36)<=key(7);      XX(37)<=key(62);     XX(38)<=key(54);
    XX(39)<=key(46);     XX(40)<=key(38);     XX(41)<=key(30);     XX(42)<=key(22);
    XX(43)<=key(14);     XX(44)<=key(6);      XX(45)<=key(61);
    XX(46)<=key(53);     XX(47)<=key(45);     XX(48)<=key(37);     XX(49)<=key(29);
    XX(50)<=key(21);     XX(51)<=key(13);     XX(52)<=key(5);
    XX(53)<=key(28);     XX(54)<=key(20);     XX(55)<=key(12);     XX(56)<=key(4);

    c0x<=XX(1 to 28);   d0x<=XX(29 to 56);
end behavior;
```

pc2.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity pc2 is port
(
    c,d    : in std_logic_vector(1 TO 28);
    k      : out std_logic_vector(1 TO 48)
);
end pc2;

architecture behavior of pc2 is
    signal YY      :      std_logic_vector(1 to 56);
begin
    YY(1 to 28)<=c;          YY(29 to 56)<=d;

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

pp.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity pp is port
(
    solx,so2x,so3x,so4x,so5x,so6x,so7x,so8x
        :      in      std_logic_vector(1 to 4);
    ppo      :      out      std_logic_vector(1 to 32)
);
end pp;

architecture behaviour of pp is
    signal XX      :      std_logic_vector(1 to 32);
begin
    XX(1 to 4)<=solx;  XX(5 to 8)<=so2x;  XX(9 to 12)<=so3x;  XX(13 to
16)<=so4x;
    XX(17 to 20)<=so5x; XX(21 to 24)<=so6x; XX(25 to 28)<=so7x; XX(29 to
32)<=so8x;

    ppo(1)<=XX(16);      ppo(2)<=XX(7);      ppo(3)<=XX(20);
    ppo(4)<=XX(21);
    ppo(5)<=XX(29);      ppo(6)<=XX(12);      ppo(7)<=XX(28);
    ppo(8)<=XX(17);
```

```
    ppo(9)<=XX(1);          ppo(10)<=XX(15);    ppo(11)<=XX(23);
    ppo(12)<=XX(26);       ppo(14)<=XX(18);    ppo(15)<=XX(31);
    ppo(13)<=XX(5);       ppo(16)<=XX(10);   ppo(17)<=XX(2);
    ppo(17)<=XX(2);       ppo(18)<=XX(8);      ppo(19)<=XX(24);
    ppo(20)<=XX(14);      ppo(21)<=XX(32);   ppo(22)<=XX(27);
    ppo(21)<=XX(32);     ppo(22)<=XX(27);   ppo(23)<=XX(3);
    ppo(24)<=XX(9);       ppo(25)<=XX(19);   ppo(26)<=XX(13);
    ppo(25)<=XX(19);     ppo(26)<=XX(13);   ppo(27)<=XX(30);
    ppo(27)<=XX(30);     ppo(28)<=XX(6);
    ppo(29)<=XX(22);     ppo(30)<=XX(11);  ppo(31)<=XX(4);
    ppo(32)<=XX(25);
end;
```

reg32.vhd

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

entity reg32 is
    port(
        a      : in    std_logic_vector (1 to 32);
        q      : out   std_logic_vector (1 to 32);
        reset  : in    std_logic;
        clk    : in    std_logic
    );
end reg32;

architecture synth of reg32 is

    signal memory : std_logic_vector (1 to 32) ;

begin

    process(clk,reset)
    begin
        if(clk = '1' and clk'event) then

            -- on affecte la mémoire interne au coup d'horloge
            memory <= a;

        end if;

        if(reset = '1') then
            memory <= (others => '0');
        end if;

    end process;

    q <= memory;

end synth;
```

testbench1.vhd

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

entity test_desenc is end test_desenc;

architecture testbench of test_desenc is

    component desenc port
```

```
(
    pt      :      in      std_logic_vector(1 TO 64);
    key     :      in      std_logic_vector(1 TO 64);
    ct      :      out     std_logic_vector(1 TO 64);
    reset   :      in      std_logic;
    clk     :      in      std_logic
);
end component;

type test_vector is record
    key      :      std_logic_vector(1 to 64);
    pt      :      std_logic_vector(1 to 64);
    ct      :      std_logic_vector(1 to 64);
end record;

type test_vector_array is array(natural range <>) of test_vector;

constant test_vectors: test_vector_array :=(
    ( key=>x"0000000000000000", pt=>x"0000000000000000",
      ct=>x"8ca64de9c1b123a7" ),
    ( key=>x"ffffffffffffffff", pt=>x"ffffffffffffffff",
      ct=>x"7359b2163e4edc58" ),
    ( key=>x"3000000000000000", pt=>x"1000000000000001",
      ct=>x"958e6e627a05557b" ),
    ( key=>x"1111111111111111", pt=>x"1111111111111111",
      ct=>x"f40379ab9e0ec533" ),
    ( key=>x"0123456789abcdef", pt=>x"1111111111111111",
      ct=>x"17668dfc7292532d" ),
    ( key=>x"1111111111111111", pt=>x"0123456789abcdef",
      ct=>x"8a5aelf81ab8f2dd" ),
    ( key=>x"0000000000000000", pt=>x"0000000000000000",
      ct=>x"8ca64de9c1b123a7" ),
    ( key=>x"fedcba9876543210", pt=>x"0123456789abcdef",
      ct=>x"ed39d950fa74bcc4" ),
    ( key=>x"7ca110454a1a6e57", pt=>x"01a1d6d039776742",
      ct=>x"690f5b0d9a26939b" ),
    ( key=>x"0131d9619dc1376e", pt=>x"5cd54ca83def57da",
      ct=>x"7a389d10354bd271" ),
    ( key=>x"07a1133e4a0b2686", pt=>x"0248d43806f67172",
      ct=>x"868ebb51cab4599a" ),
    ( key=>x"3849674c2602319e", pt=>x"51454b582ddf440a",
      ct=>x"7178876e01f19b2a" ),
    ( key=>x"04b915ba43feb5b6", pt=>x"42fd443059577fa2",
      ct=>x"af37fb421f8c4095" ),
    ( key=>x"0113b970fd34f2ce", pt=>x"059b5e0851cf143a",
      ct=>x"86a560f10ec6d85b" ),
    ( key=>x"0170f175468fb5e6", pt=>x"0756d8e0774761d2",
      ct=>x"0cd3da020021dc09" ),
    ( key=>x"43297fad38e373fe", pt=>x"762514b829bf486a",
      ct=>x"ea676b2cb7db2b7a" ),
    ( key=>x"07a7137045da2a16", pt=>x"3bdd119049372802",
      ct=>x"dfd64a815caf1a0f" ),
    ( key=>x"04689104c2fd3b2f", pt=>x"26955f6835af609a",
      ct=>x"5c513c9c4886c088" ),
    ( key=>x"37d06bb516cb7546", pt=>x"164d5e404f275232",
      ct=>x"0a2aeae3ff4ab77" ),
    ( key=>x"1f08260d1ac2465e", pt=>x"6b056e18759f5cca",
      ct=>x"ef1bf03e5dfa575a" ),
    ( key=>x"584023641aba6176", pt=>x"004bd6ef09176062",
      ct=>x"88bf0db6d70dee56" ),
    ( key=>x"025816164629b007", pt=>x"480d39006ee762f2",
      ct=>x"alf9915541020b56" ),
    ( key=>x"49793ebc79b3258f", pt=>x"437540c8698f3cfa",
      ct=>x"6fbf1cafcaffd0556" ),
    ( key=>x"4fb05e1515ab73a7", pt=>x"072d43a077075292",
      ct=>x"2f22e49bab7calac" ),
```

```

        ( key=>x"49e95d6d4ca229bf", pt=>x"02fe55778117f12a",
          ct=>x"5a6b612cc26cce4a" ),
        ( key=>x"018310dc409b26d6", pt=>x"1d9d5c5018f728c2",
          ct=>x"5f4c038ed12b2e41" ),
        ( key=>x"1c587f1c13924fef", pt=>x"305532286d6f295a",
          ct=>x"63fac0d034d9f793" ),
        ( key=>x"0101010101010101", pt=>x"0123456789abcdef",
          ct=>x"617b3a0ce8f07100" ),
        ( key=>x"1f1f1f1f0e0e0e0e", pt=>x"0123456789abcdef",
          ct=>x"db958605f8c8c606" ),
        ( key=>x"e0fee0fef1fef1fe", pt=>x"0123456789abcdef",
          ct=>x"edbfd1c66c29ccc7" ),
        ( key=>x"0000000000000000", pt=>x"ffffffffffffffff",
          ct=>x"355550b2150e2451" ),
        ( key=>x"ffffffffffffffff", pt=>x"0000000000000000",
          ct=>x"caaaaf4deaf1dbae" ),
        ( key=>x"0123456789abcdef", pt=>x"0000000000000000",
          ct=>x"d5d44ff720683d0d" ),
        ( key=>x"fedcba9876543210", pt=>x"ffffffffffffffff",
          ct=>x"2a2bb008df97c2f2" )
    );

    signal key : std_logic_vector(1 to 64);
    signal pt : std_logic_vector(1 to 64);
    signal ct : std_logic_vector(1 to 64);
    signal clk : std_logic;
    signal reset : std_logic;

begin

dut: desenc port map ( key=>key, pt=>pt, ct=>ct, reset=>reset,
clk=>clk );

    process
        variable vector : test_vector;
        variable errors : boolean:=false;
    begin
        for i in test_vectors'range loop
            vector:=test_vectors(i);
            key<=vector.key; pt<=vector.pt;

            for j in 0 to 15 loop clk<='0'; wait for 250 ns;
            clk<='1'; wait for 250 ns; end loop;

            if(ct/=vector.ct) then
                assert false
                report "Implementation Failure"
                severity note;
                errors:=true;
            end if;
        end loop;

        assert not errors
            report "Test vectors failed"
            severity note;
        assert errors
            report "Test vectors passed"
            severity note;
        wait;
    end process;

end testbench;

```

Spartan 2 version (with converters)

desenc.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity desenc is port
(
    pt      :    in    std_logic_vector(1 TO 16);
    key     :    in    std_logic_vector(1 TO 16);
    ct      :    out   std_logic_vector(1 TO 64);
    reset   :    in    std_logic;
    load_data :    in   std_logic;
    clk     :    in    std_logic
);
end desenc;

architecture behavior of desenc is
    signal k1x,k2x,k3x,k4x,k5x,k6x,k7x,k8x,k9x,k10x,k11x,k12x,k13x,k14x,k15x,k16x
        :    std_logic_vector(1 to 48);
    signal
    l0xa,l1x,l2x,l3x,l4x,l5x,l6x,l7x,l8x,l9x,l10x,l11x,l12x,l13x,l14x,l15x,l16x
        :    std_logic_vector(1 to 32);
    signal
    r0xa,r1x,r2x,r3x,r4x,r5x,r6x,r7x,r8x,r9x,r10x,r11x,r12x,r13x,r14x,r15x,r16x
        :    std_logic_vector(1 to 32);
    signal key_sig      :    std_logic_vector(1 to 64);
    signal pt_sig       :    std_logic_vector(1 to 64);

    component keysched
    port (
        key      :    in    std_logic_vector(1 to 64);
        k1x,k2x,k3x,k4x,k5x,k6x,k7x,k8x,k9x,k10x,k11x,k12x,k13x,k14x,k15x,k16x
            :    out   std_logic_vector(1 to 48)
    );
    end component;

    component ip
    port (
        pt      :    in std_logic_vector(1 TO 64);
        l0x     :    out std_logic_vector(1 TO 32);
        r0x     :    out std_logic_vector(1 TO 32)
    );
    end component;

    component roundfunc
    port (
        clk      :    in    std_logic;
        reset    :    in    std_logic;
        li,ri    :    in    std_logic_vector(1 to 32);
        k        :    in    std_logic_vector(1 to 48);
        lo,ro    :    out   std_logic_vector(1 to 32)
    );
    end component;

    component fp
    port (
        l,r     :    in    std_logic_vector(1 to 32);
        ct      :    out   std_logic_vector(1 to 64)
    );
    end component;

    component converter
    port (
        input   :    in std_logic_vector(1 to 16);
        clk     :    in std_logic;

```

```

        load_data      :      in      std_logic;
        output :      out std_logic_vector (1 to 64)
    );
end component;
begin

    key_converter:      converter port map ( input=>key, clk=>clk,
load_data=>load_data, output=>key_sig );
    pt_converter:converter port map ( input=>pt, clk=>clk, load_data=>load_data,
output=>pt_sig );

    keyscheduling:      keysched      port map      (      key=>key_sig,k1x=>k1x,
k2x=>k2x,      k3x=>k3x,      k4x=>k4x,      k5x=>k5x,      k6x=>k6x,      k7x=>k7x,
k8x=>k8x,      k9x=>k9x,      k10x=>k10x,      k11x=>k11x,      k12x=>k12x,      k13x=>k13x,
k14x=>k14x,      k15x=>k15x,      k16x=>k16x      );

    iperm:      ip      port map      (      pt=>pt_sig,
l0x=>l0xa,      r0x=>r0xa      );

    round1:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l0xa,      ri=>r0xa,      k=>k1x,      lo=>l1x,      ro=>r1x      );
    round2:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l1x,      ri=>r1x,      k=>k2x,      lo=>l2x,      ro=>r2x      );
    round3:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l2x,      ri=>r2x,      k=>k3x,      lo=>l3x,      ro=>r3x      );
    round4:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l3x,      ri=>r3x,      k=>k4x,      lo=>l4x,      ro=>r4x      );
    round5:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l4x,      ri=>r4x,      k=>k5x,      lo=>l5x,      ro=>r5x      );
    round6:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l5x,      ri=>r5x,      k=>k6x,      lo=>l6x,      ro=>r6x      );
    round7:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l6x,      ri=>r6x,      k=>k7x,      lo=>l7x,      ro=>r7x      );
    round8:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l7x,      ri=>r7x,      k=>k8x,      lo=>l8x,      ro=>r8x      );
    round9:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l8x,      ri=>r8x,      k=>k9x,      lo=>l9x,      ro=>r9x      );
    round10:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l9x,      ri=>r9x,      k=>k10x,      lo=>l10x,      ro=>r10x      );
    round11:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l10x,      ri=>r10x,      k=>k11x,      lo=>l11x,      ro=>r11x      );
    round12:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l11x,      ri=>r11x,      k=>k12x,      lo=>l12x,      ro=>r12x      );
    round13:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l12x,      ri=>r12x,      k=>k13x,      lo=>l13x,      ro=>r13x      );
    round14:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l13x,      ri=>r13x,      k=>k14x,      lo=>l14x,      ro=>r14x      );
    round15:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l14x,      ri=>r14x,      k=>k15x,      lo=>l15x,      ro=>r15x      );
    round16:      roundfunc      port map      (      clk=>clk,      reset=>reset,
li=>l15x,      ri=>r15x,      k=>k16x,      lo=>l16x,      ro=>r16x      );

    fperm: fp      port map      (      l=>r16x,      r=>l16x,      ct=>ct );

end behavior;

```

converter.vhd

```

library ieee;

use ieee.std_logic_1164.all;

entity converter is port
(
    input :      in std_logic_vector(1 to 16);

```

```
        clk      :      in std_logic;
        load_data :      in      std_logic;
        output   :      out std_logic_vector (1 to 64)
    );
end converter;

architecture behavior of converter is
    signal memory:      std_logic_vector( 1 to 48);
    type state_type is (load1, load2, load3, unload);
    signal state,f_state: state_type;
begin

    process(state)
    begin

    case state is

        when load1 =>
            memory <= input & memory(1 to 32);
            f_state <= load2;

        when load2 =>
            memory <= input & memory(1 to 32);
            f_state <= load3;

        when load3 =>
            memory <= input & memory(1 to 32);
            f_state <= unload;

        when unload =>
            output <= memory & input;
            f_state <= unload;
    end case;

    end process;

    process (clk,load_data)
    begin

    if (clk'event and clk = '1') then
        state <= f_state;
    end if;

    if(load_data='1') then
        state <= load1;
    end if;

    end process;

end behavior;
```

testbench1.vhd

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

entity test_desenc is end test_desenc;

architecture testbench of test_desenc is

component desenc port
(
    pt          :      in      std_logic_vector(1 TO 16);
    key         :      in      std_logic_vector(1 TO 16);
    ct          :      out     std_logic_vector(1 TO 64);

```



```
        reset      :      in      std_logic;
        load_data   :      in      std_logic;
        clk         :      in      std_logic
    );
end component;

type test_vector is record
    key      :      std_logic_vector(1 to 16);
    pt       :      std_logic_vector(1 to 16);
    ct       :      std_logic_vector(1 to 64);
end record;

type test_vector_array is array(natural range <>) of test_vector;

constant test_vectors: test_vector_array :=(
    (      key=>x"0000",pt=>x"0000", ct=>x"8ca64de9c1b123a7"   ),
    (      key=>x"ffff",pt=>x"ffff", ct=>x"7359b2163e4edc58"  ),
    (      key=>x"1111",pt=>x"1111", ct=>x"f40379ab9e0ec533"  )
);

signal      key : std_logic_vector(1 to 16);
signal      pt  : std_logic_vector(1 to 16);
signal      ct  : std_logic_vector(1 to 64);
signal      clk : std_logic;
signal      reset : std_logic;
signal      load_data : std_logic;

begin

dut: desenc port map      (      key=>key,      pt=>pt,      ct=>ct,      reset=>reset,
    load_data=>load_data,      clk=>clk      );

    process
        variable      vector :      test_vector;
        variable      errors :      boolean:=false;
    begin
        clk<='1';      wait for 250 ns;      clk<='0';      wait for 250 ns;

        for i in test_vectors'range loop
            vector:=test_vectors(i);
            key<=vector.key;      pt<=vector.pt;
            load_data <= '1';
            clk<='1';      wait for 250 ns;      clk<='0';      wait for 250 ns;
            load_data <= '0';

            for j in 0 to 17 loop clk<='1';      wait for 250 ns;      clk<='0';
            wait for 250 ns; end loop;

            end loop;

            wait;

        end process;

end testbench;
```

Small design

state.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity state is port
(
    pt      :    in    std_logic_vector(1 TO 16);
    key     :    in    std_logic_vector(1 TO 16);
    clk     :    in    std_logic;
    reset   :    in    std_logic;
    ct      :    out   std_logic_vector(1 TO 64)
);
end state;

architecture structural of state is

component control
port (
    reset      :    in    std_logic;
    clk        :    in    std_logic;
    load_new_pt :    out   std_logic;
    output_ok  :    out   std_logic;
    load_pt    :    :    out   std_logic;
    unload_pt  :    out   std_logic;
    load_key   :    out   std_logic;
    unload_key :    out   std_logic;
    shift      :    out   std_logic_vector(1 to 3)
);
end component;

component fullround
port (
    pt          :    in    std_logic_vector(1 TO 64);
    xkey        :    in    std_logic_vector(1 TO 48);
    reset       :    in    std_logic;
    clk         :    in    std_logic;
    load_new_pt :    in    std_logic;
    output_ok   :    in    std_logic;
    ct          :    out   std_logic_vector(1 TO 64)
);
end component;

component keysched
port (
    the_key     :    in    std_logic_vector(1 to 64);
    shift       :    in    std_logic_vector(1 to 3);
    clk         :    in    std_logic;
    ki         :    out   std_logic_vector(1 to 48)
);
end component;

component converter
port (
    input      :    in    std_logic_vector(1 to 16);
    load       :    in    std_logic;
    unload     :    in    std_logic;
    clk        :    in    std_logic;
    output     :    out   std_logic_vector (1 to 64)
);
end component;
```

```
signal load_new_pt :          std_logic;
signal output_ok   :          std_logic;
signal load_pt_sig :          std_logic;
signal unload_pt_sig :        std_logic;
signal load_key_sig :         std_logic;
signal unload_key_sig :       std_logic;
signal shift_sig   :          std_logic_vector(1 to 3);
signal ki_sig      :          std_logic_vector(1 to 48);
signal key_sig     :          std_logic_vector(1 to 64);
signal pt_sig      :          std_logic_vector(1 to 64);

begin

control_unit : control      port map ( reset=>reset, clk=>clk,
load_new_pt=>load_new_pt, output_ok=>output_ok, load_pt=>load_pt_sig,
unload_pt=>unload_pt_sig, load_key=>load_key_sig, unload_key=>unload_key_sig,
shift=>shift_sig );

converter_pt : converter    port map ( input=>pt, load=>load_pt_sig,
unload=>unload_pt_sig, clk=>clk, output=>pt_sig );
converter_key: converter    port map ( input=>key, load=>load_key_sig,
unload=>unload_key_sig, clk=>clk, output=>key_sig );

datapath     : fullround    port map ( pt=>pt_sig, xkey=>ki_sig, reset=>reset,
clk=>clk,load_new_pt=>load_new_pt, output_ok=>output_ok, ct=>ct );

key_proc     : keysched     port map ( the_key=>key_sig, shift=>shift_sig, clk=>clk,
ki=>ki_sig );

end structural;
```

control.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity control is port
(
    reset      :    in    std_logic;
    clk        :    in    std_logic;
    load_new_pt :    out   std_logic;
    output_ok   :    out   std_logic;
    load_pt     :    out   std_logic;
    unload_pt   :    out   std_logic;
    load_key    :    out   std_logic;
    unload_key  :    out   std_logic;
    shift       :    out   std_logic_vector(1 to 3)
);
end control;

architecture behavior of control is

    type typeetat is (LOAD1, LOAD2, LOAD3, LOAD4, INIT, R1, R2, R3, R4, R5, R6,
R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, KEY_LOADING);
    signal etat, etatfutur : typeetat;

begin

    process(etat)
    begin
```

```
case etat is

    when LOAD1 =>
        load_new_pt <= '0';
        output_ok<= '0';
        load_pt <= '1';
        unload_pt <= '0';
        load_key <= '1';
        unload_key <= '0';
        shift <= "000";
        etatfutur <= LOAD2;

    when LOAD2 =>
        load_new_pt <= '0';
        output_ok<= '0';
        load_pt <= '1';
        unload_pt <= '0';
        load_key <= '1';
        unload_key <= '0';
        shift <= "000";
        etatfutur <= LOAD3;

    when LOAD3 =>
        load_new_pt <= '0';
        output_ok<= '0';
        load_pt <= '1';
        unload_pt <= '0';
        load_key <= '1';
        unload_key <= '0';
        shift <= "000";
        etatfutur <= LOAD4;

    when LOAD4 =>
        load_new_pt <= '0';
        output_ok<= '0';
        load_pt <= '0';
        unload_pt <= '1';
        load_key <= '0';
        unload_key <= '1';
        shift <= "000";
        etatfutur <= INIT;

    when INIT =>
        load_new_pt <= '0';
        output_ok<= '0';
        load_pt <= '0';
        unload_pt <= '0';
        load_key <= '0';
        unload_key <= '0';
        shift <= "011";
        etatfutur <= R1;

    when R1 =>
        load_new_pt <= '1';
        output_ok<= '0';
        load_pt <= '0';
        unload_pt <= '0';
        load_key <= '0';
        unload_key <= '0';
        shift <= "010";
        etatfutur <= R2;

    when R2 =>
        load_new_pt <= '0';
```

```
        output_ok<= '0';
        load_pt <= '0';
        unload_pt <= '0';
        load_key <= '0';
        unload_key <= '0';
        shift <= "100";
        etatfutur <= R3;

when R3 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '0';
    unload_pt <= '0';
    load_key <= '0';
    unload_key <= '0';
    shift <= "100";
    etatfutur <= R4;

when R4 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '0';
    unload_pt <= '0';
    load_key <= '0';
    unload_key <= '0';
    shift <= "100";
    etatfutur <= R5;

when R5 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '0';
    unload_pt <= '0';
    load_key <= '0';
    unload_key <= '0';
    shift <= "100";
    etatfutur <= R6;

when R6 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '0';
    unload_pt <= '0';
    load_key <= '0';
    unload_key <= '0';
    shift <= "100";
    etatfutur <= R7;

when R7 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '0';
    unload_pt <= '0';
    load_key <= '0';
    unload_key <= '0';
    shift <= "100";
    etatfutur <= R8;

when R8 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '0';
    unload_pt <= '0';
    load_key <= '0';
    unload_key <= '0';
    shift <= "010";
    etatfutur <= R9;
```

```
when R9 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '0';
    unload_pt <= '0';
    load_key <= '0';
    unload_key <= '0';
    shift <= "100";
    etatfutur <= R10;

when R10 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '0';
    unload_pt <= '0';
    load_key <= '0';
    unload_key <= '0';
    shift <= "100";
    etatfutur <= R11;

when R11 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '0';
    unload_pt <= '0';
    load_key <= '0';
    unload_key <= '0';
    shift <= "100";
    etatfutur <= R12;

when R12 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '0';
    unload_pt <= '0';
    load_key <= '0';
    unload_key <= '0';
    shift <= "100";
    etatfutur <= R13;

when R13 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '1';
    unload_pt <= '0';
    load_key <= '1';
    unload_key <= '0';
    shift <= "100";
    etatfutur <= R14;

when R14 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '1';
    unload_pt <= '0';
    load_key <= '1';
    unload_key <= '0';
    shift <= "100";
    etatfutur <= R15;

when R15 =>
    load_new_pt <= '0';
    output_ok<= '0';
    load_pt <= '1';
    unload_pt <= '0';
    load_key <= '1';
```

```
        unload_key <= '0';
        shift <= "010";
        etatfutur <= R16;

    when R16 =>
        load_new_pt <= '0';
        output_ok<= '0';
        load_pt <= '0';
        unload_pt <= '1';
        load_key <= '0';
        unload_key <= '1';
        shift <= "000";
        etatfutur <= KEY_LOADING;

    when KEY_LOADING =>
        load_new_pt <= '0';
        output_ok<= '1';
        load_pt <= '0';
        unload_pt <= '0';
        load_key <= '0';
        unload_key <= '0';
        shift <= "011";
        etatfutur <= R1;

    end case;

    end process;

    process (clk, reset)
    begin

        if (clk'event and clk = '1') then
            etat <= etatfutur;
        end if;

        if(reset='1') then
            etat <= LOAD1;
        end if;

    end process;

end behavior;
```

converter.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity converter is port
(
    input  :    in std_logic_vector(1 to 16);
    load   :    in std_logic;
    unload :    in std_logic;
    clk    :    in std_logic;
    output :    out std_logic_vector (1 to 64)
);
end converter;

architecture behavior of converter is
    signal memory:    std_logic_vector( 1 to 48);
begin

    process(load,clk)
```

```
begin
if (clk'event and clk = '1') then
if(load = '1') then
memory <= input & memory(1 to 32);
end if;
end if;

end process;

process(unload,clk)
begin

if (clk'event and clk = '1') then
if(unload = '1') then
output <= memory & input;
end if;
end if;

end process;

end behavior;
```

fullround.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity fullround is port
(
    pt          : in    std_logic_vector(1 TO 64);
    xkey        : in    std_logic_vector(1 TO 48);
    reset       : in    std_logic;
    clk         : in    std_logic;
    load_new_pt : in    std_logic;
    output_ok   : in    std_logic;
    ct         : out   std_logic_vector(1 TO 64)
);
end fullround;

architecture behavior of fullround is

    component mux32
    port (
        e0 : in    std_logic_vector (1 to 32) ;
        e1 : in    std_logic_vector (1 to 32) ;
        o  : out   std_logic_vector (1 to 32) ;
        sel : in    std_logic
    );
    end component;

    component roundfunc
    port (
        clk         : in    std_logic;
        reset       : in    std_logic;
        li,ri      : in    std_logic_vector(1 to 32);
        k           : in    std_logic_vector(1 to 48);
        lo,ro      : out   std_logic_vector(1 to 32)
    );
    end component;

    component ov32
    port (
        e : in    std_logic_vector (1 to 32) ;
        o1 : out   std_logic_vector (1 to 32) ;
        o2 : out   std_logic_vector (1 to 32) ;
        clk : in    std_logic;
    );
    end component;
```



```

        sel : in      std_logic
    );
end component;

component fp
port (
    l,r      :   in      std_logic_vector(1 to 32);
    ct       :   out     std_logic_vector(1 to 64)
);
end component;

component ip
port (
    pt       :   in     std_logic_vector(1 TO 64);
    l0x      :   out    std_logic_vector(1 TO 32);
    r0x      :   out    std_logic_vector(1 TO 32)
);
end component;

signal left_in      :      std_logic_vector(1 to 32);
signal right_in     :      std_logic_vector(1 to 32);
signal mux_l_to_round :    std_logic_vector(1 to 32);
signal mux_r_to_round :    std_logic_vector(1 to 32);
signal round_l_to_ov :      std_logic_vector(1 to 32);
signal round_r_to_ov :      std_logic_vector(1 to 32);
signal ov_l_to_mux  :      std_logic_vector(1 to 32);
signal ov_r_to_mux  :      std_logic_vector(1 to 32);
signal ov_l_to_fp   :      std_logic_vector(1 to 32);
signal ov_r_to_fp   :      std_logic_vector(1 to 32);
begin

    initial_p:  ip    port map ( pt=>pt, l0x=>left_in, r0x=>right_in );

    mux_left:   mux32 port map ( e0=>ov_l_to_mux, e1=>left_in,
o=>mux_l_to_round, sel=>load_new_pt );
    mux_right:  mux32 port map ( e0=>ov_r_to_mux, e1=>right_in,
o=>mux_r_to_round, sel=>load_new_pt );

    round: roundfunc port map ( clk=>clk, reset=>reset,
li=>mux_l_to_round, ri=>mux_r_to_round, k=>xkey, lo=>round_l_to_ov,
ro=>round_r_to_ov );

    ov_left:   ov32 port map ( e=>round_l_to_ov, o1=>ov_l_to_mux,
o2=>ov_l_to_fp, clk=>clk, sel=>output_ok );
    ov_right:  ov32 port map ( e=>round_r_to_ov, o1=>ov_r_to_mux,
o2=>ov_r_to_fp, clk=>clk, sel=>output_ok );

    final_p:   fp    port map ( l=>ov_r_to_fp, r=>ov_l_to_fp, ct=>ct );
end behavior;

```

fullround.vhd

```

library ieee;

use ieee.std_logic_1164.all;

entity keysched is port
(
    the_key      :   in      std_logic_vector(1 to 64);
    shift       :   in      std_logic_vector(1 to 3);
    clk         :   in      std_logic;
    ki          :   out     std_logic_vector(1 to 48)
);
end keysched;

```

```
architecture behaviour of keysched is
    signal c,d,c1,d1      :      std_logic_vector(1 to 28);

    component pc1
    port (
        key      :      in std_logic_vector(1 TO 64);
        c0x,d0x  :      out std_logic_vector(1 TO 28)
    );
    end component;

    component shifter
    port (
        datac      :      in      std_logic_vector(1 to 28);
        datad      :      in      std_logic_vector(1 to 28);
        shift      :      in      std_logic_vector(1 to 3);
        clk        :      in      std_logic;
        datac_out  :      out     std_logic_vector(1 to 28);
        datad_out  :      out     std_logic_vector(1 to 28)
    );
    end component;

    component pc2
    port (
        c,d      : in std_logic_vector(1 TO 28);
        k        : out std_logic_vector(1 TO 48)
    );
    end component;

begin

pc_1: pc1 port map ( key=>the_key, c0x=>c, d0x=>d );

shifter_comp: shifter port map ( datac=>c, datad=>d, shift=>shift, clk=>clk,
datac_out=>c1, datad_out=>d1 );

pc_2: pc2 port map ( c=>c1, d=>d1, k=>ki );

end behaviour;
```

keysched.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity keysched is port
(
    the_key      :      in      std_logic_vector(1 to 64);
    shift      :      in      std_logic_vector(1 to 3);
    clk        :      in      std_logic;
    ki         :      out     std_logic_vector(1 to 48)
);
end keysched;

architecture behaviour of keysched is
    signal c,d,c1,d1      :      std_logic_vector(1 to 28);

    component pc1
    port (
        key      :      in std_logic_vector(1 TO 64);
        c0x,d0x  :      out std_logic_vector(1 TO 28)
    );
    end component;
```

```
    component shifter
    port (
        datac      : in    std_logic_vector(1 to 28);
        datad      : in    std_logic_vector(1 to 28);
        shift      : in    std_logic_vector(1 to 3);
        clk        : in    std_logic;
        datac_out   : out   std_logic_vector(1 to 28);
        datad_out   : out   std_logic_vector(1 to 28)
    );
end component;

    component pc2
    port (
        c,d       : in std_logic_vector(1 TO 28);
        k         : out std_logic_vector(1 TO 48)
    );
end component;

begin

pc_1: pc1 port map ( key=>the_key, c0x=>c, d0x=>d );

shifter_comp: shifter port map ( datac=>c, datad=>d, shift=>shift, clk=>clk,
datac_out=>c1, datad_out=>d1 );

pc_2: pc2 port map ( c=>c1, d=>d1, k=>ki );

end behaviour;
```

mux32.vhd

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;

ENTITY mux32 IS
    PORT(
        e0 : IN    std_logic_vector (1 to 32) ;
        e1 : IN    std_logic_vector (1 to 32) ;
        o  : OUT   std_logic_vector (1 to 32) ;
        sel : IN    std_logic
    );
END mux32 ;

ARCHITECTURE synth OF mux32 IS
BEGIN

process(sel,e0,e1)
begin

if sel = '0' then
    o <= e0;
else
    o <= e1;
end if;

end process;

END synth;
```

ov32.vhd

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all;
```

```
ENTITY ov32 IS
  PORT(
    e : in      std_logic_vector (1 to 32) ;
    o1 : out    std_logic_vector (1 to 32) ;
    o2 : out    std_logic_vector (1 to 32) ;
    clk : in    std_logic;
    sel : in    std_logic
  );
END ov32 ;
```

```
ARCHITECTURE synth OF ov32 IS
BEGIN

  process(sel,clk)
  begin

    if (clk'event and clk = '1') then
      if(sel = '1') then
        o2<=e;
      end if;
    end if;

    end process;

    o1<=e;

  END synth;
```

shifter.vhd

```
library ieee;

use ieee.std_logic_1164.all;

entity shifter is port
(
  datac      : in    std_logic_vector(1 to 28);
  datad      : in    std_logic_vector(1 to 28);
  shift      : in    std_logic_vector(1 to 3);
  clk        : in    std_logic;
  datac_out  : out   std_logic_vector(1 to 28);
  datad_out  : out   std_logic_vector(1 to 28)
);
end shifter;

architecture behaviour of shifter is
  signal datac_out_mem, datad_out_mem : std_logic_vector(1 to 28);
begin

  process(shift,clk)
  begin

    if (clk'event and clk = '1') then
      case shift is

        when "001" =>
          -- pas de shift, nouvelle clé
          datac_out_mem<=datac;
          datad_out_mem<=datad;

        when "010" =>
          -- shifter 1 fois, pas de nouvelle clé
```

```
        datac_out_mem<=To_StdLogicVector(to_bitvector(datac_out_mem) rol 1);
        datad_out_mem<=To_StdLogicVector(to_bitvector(datad_out_mem) rol 1);

    when "011" =>
        -- shifter 1 fois, nouvelle clé
        datac_out_mem<=To_StdLogicVector(to_bitvector(datac) rol 1);
        datad_out_mem<=To_StdLogicVector(to_bitvector(datad) rol 1);

    when "100" =>
        -- shifter 2 fois, pas de nouvelle clé
        datac_out_mem<=To_StdLogicVector(to_bitvector(datac_out_mem) rol 2);
        datad_out_mem<=To_StdLogicVector(to_bitvector(datad_out_mem) rol 2);

    when "101" =>
        -- shifter 2 fois, nouvelle clé
        datac_out_mem<=To_StdLogicVector(to_bitvector(datac) rol 2);
        datad_out_mem<=To_StdLogicVector(to_bitvector(datad) rol 2);

    when others =>
        -- erreur ou pas de shift, pas de nouvelle clé
end case;
end if;

end process;

datac_out<=datac_out_mem;
datad_out<=datad_out_mem;

end behaviour;
```

testbench_FPGA.vhd

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

entity test_fpga is end test_fpga;

architecture testbench of test_fpga is

    component state_port
    (
        pt      :    in    std_logic_vector(1 TO 16);
        key     :    in    std_logic_vector(1 TO 16);
        clk     :    in    std_logic;
        reset   :    in    std_logic;
        ct      :    out   std_logic_vector(1 TO 64)
    );
    end component;

    type test_vector is record
        key     :    std_logic_vector(1 to 16);
        pt      :    std_logic_vector(1 to 16);
        ct      :    std_logic_vector(1 to 64);
    end record;

    type test_vector_array is array(natural range <>) of test_vector;

    constant test_vectors: test_vector_array :=(
        ( key=>x"1111",pt=>x"1111", ct=>x"f40379ab9e0ec533" ),
        ( key=>x"ffff",pt=>x"0000", ct=>x"caaaaf4deaf1dbae" ),
        ( key=>x"0000",pt=>x"0000", ct=>x"8ca64de9c1b123a7" )
    );
```

```
    signal    key : std_logic_vector(1 to 16);
    signal    pt  : std_logic_vector(1 to 16);
    signal    ct  : std_logic_vector(1 to 64);
    signal    clk : std_logic;
    signal    reset : std_logic;

begin

dut:  state port map    (    pt=>pt,  key=>key,  ct=>ct,  reset=>reset,
    clk=>clk    );

    process
        variable    vector :    test_vector;
    begin

        key<=x"0000";pt<=x"0000";

        for j in 0 to 17 loop    clk<='0';    wait for 250 ns;    clk<='1';
    wait for 250 ns;    end loop;

        key<=x"ffff";pt<=x"ffff";
        for j in 0 to 3 loop    clk<='0';    wait for 250 ns;    clk<='1';
    wait for 250 ns;    end loop;

        for i in test_vectors'range loop

            for j in 0 to 16 loop    clk<='0';    wait for 250 ns;
    clk<='1';    wait for 250 ns;    end loop;
            vector:=test_vectors(i);
            key<=vector.key;    pt<=vector.pt;
            for j in 0 to 3 loop    clk<='0';    wait for 250 ns;
    clk<='1';    wait for 250 ns;    end loop;
            end loop;

            for j in 0 to 17 loop    clk<='0';    wait for 250 ns;    clk<='1';
    wait for 250 ns;    end loop;

            wait;

        end process;

end testbench;
```