>rhu(jdk8rjG&
do*ias8uH-g/i
4sds0f8afs$

VHDL

```
process(str)
begin
  if(key='1')
    decode(str);
end if;
end process;
```

FPGA

Lehrstuhl für
Entwurfsauto-
matisierung

VHDL System Design Lab

Lab Manual

Hardware Implementation
Of an Encryption Algorithm

Prof. Dr.-Ing. Ulf Schlichtmann

Institute of Electronic Design Automation
Technische Universität München
Arcisstr. 21, 80333 München
Tel.: (089) 289 23666

Contents

List of Figures

List of Tables

1 Checklist for Completion of the Laboratory

1. Evaluation

   - Moodle E-Test: In the test all the marked questions (marked with ?) in this manual are asked.

   - Project code: please insert your name and account as comments. The project code of RCS1, RCS2 and RCS2+ should work for the simulation and on FPGA hardware. Four subdirectories inside the submission directory for submission have been created there for the direct implementation, RCS1, RCS2 and RCS2+ respectively. There are some template files given which have to be used without changing their entities or filenames. For the direct implementation section 4, for RCS1 section 5.2, for RCS2 section 5.3 and for RCS2+ section 5.4 **must be strictly followed**. Only the code directly inside these directories (submit/direct, submit/rcs1, submit/rcs2, submit/rcs2plus) will be evaluated. For evaluation, you must guarantee that the submitted project code can be directly imported into the ISE tool and tested. Please make sure that all vhdl files are located inside these subdirectorys without using links! Furthermore use the mentioned names in this manual for the projects, the VHDL modules and the ports. Please make sure that all VHDL files belonging to an implementation are directly available (without subfolders) in the mentioned directories. Files with wrong names or in wrong places can not be graded. Furthermore wrong module names and port names will lead to an error and subsequently to a deduction of the grade.

   - Written exam: 60 min., no notes allowed.

2. Grade

   The E-Test and project code will be one grade and will contribute 60% to the final grade. The written exam contribute 40% to the final grade.

3. The working environment and tools

   You can work at home on your own computer or you can use our lab rooms (2977). You can access the room 2977 with your student-id. For login at institute the user name is `your_lrz_account` (e.g. `gu99xxx`) while the password is the same as that of TUMOnline. For remote login you need to install the vpn client

   Please refer to "VHDL System Design Laboratory Remote-login Instructions" and "Video: VHDL_RemoteTutorial " on Moodle to access computers in room 2977 remotely!

All the tools needed for this lab are integrated into a shell script called `hdl`. With `hdl help` all the possible subcommands for this lab are listed. Here are the meanings of individual subcommands.

- With `hdl manual` the manual of this lab will be opened.

- All projects of this lab are conducted in the integrated development environment Xilinx® ISE *WebPack^{TM}*. You can use it ~~at the institute in the lab room,~~ [by accesing computers in the lab room remotely] or download this software free of charge from the Xilinx website http://www.xilinx.com. On the lab computers this tool is started by executing `hdl ise` in a shell. Please make sure to use the given templates in the submission directory. Do not change the entities or filenames. It is recommended that you do your project in your working directory or already in the submission directory. For the submission, you should copy the project from your working directory to the submission directory.

- For final testing the tool `ideatester` is used to facilitate the communication between the host computer and the FPGA board. This tool is started by executing `hdl ideatester` in shell.

- With `hdl bash` a shell will be started in your lab working directory.

- With `hdl browse` the content of your lab working directory will be shown.

You can also start the tools using the corresponding entries in the system menu.

As working via [remote login] ~~ssh~~ is quite slow because of the limited bandwith of the internet connection it is recommend to work on your local computer and transfer later on the created files into the working directory: `/usr/local/labs/HDL/current/your_lrz_account` which is in the nfs system of our institute.

Please refer to "VHDL System Design Laboratory Remote-login Instructions" and "Video: VHDL_RemoteTutorial_Win10" to copy your local folders/files to the submission directory of the computers in room 2977.

`praktX.regent.e-technik.tu-muenchen.de` ~~in order to upload your files into your lab folder. You need an vpn connection with this tool, too! On Linux you can use the scp command. Please refer to the manual of this command.~~

4. Materials

- ~~There are reference models for the individual modules of the IDEA algorithm. They can help you to verify the funktion of your VHDL code. They can be found on our webpage at~~ ~~`http://www.eda.ei.tum.de/en/courses/laboratories/vhdl-system-design/reference-models`~~ ~~For this you need to enable Java and Java-Script in your web browser.~~

- The user constraints file of hardware implementation and the templates for the UART under `/usr/local/labs/HDL/share/template`

- The source code for the tool `ideatester` is available under `/usr/local/labs/HDL/share/source`.

- Link to information of the board can be found on http://www.xilinx.com/products/boards-and-kits/HW-SPAR3E-SK-US-G.htm.

- Schedule, lab manual and lecture notes are or will be available on Moodle.

5. Contact

- You can come to the tutor hours shown in Moodle in room 2977. Please check always on **Moodle** if there are changes concerning the tutor hours. They will create online meetings, send the links to you, and publish the links on Moodle.

- Or contact the lab supervisor:

  Dr.-Ing. Li Zhang
  Room 2918
  Tel.: (089) - 289 23644
  E-Mail: grace-li.zhang@tum.de

2 Introduction

2.1 Basics

Using today's production technology, the complexity of circuit designs can easily grow beyond the limits of manual design. The use of tools which support the designer in the design process is both necessary and economical in this situation. Such tools are available for a wide range of tasks. Some examples would be simulation tools to ensure that specifications are met, the computer-aided realization of specifications in an implementation, or the automatic generation of the layout data for production.

Modern design tools help accelerate the design process and lower the costs. Since Field Programmable Gate Arrays (FPGAs) provide a cheap target technology, the design of complex application-specific hardware has become economically feasible.

As a standardized hardware description language, VHDL has found industry-wide use for the design of highly integrated circuits. Essentially VHDL is based on the ADA language and provides constructs necessary for the description of hardware. The following are its most prominent features:

- Description of concurrency

- Description of time-behavior

- Description of circuit structures with support for hierarchial and modular design

VHDL also offers the option of describing a circuit on different levels of abstraction (Fig. 1). This gives the designer the possibility to (relatively) quickly test a system on an algorithmic level in order to verify the correct function of a concept. Consequently, flaws and weaknesses of a design become apparent at the early stage. Yet the simulation on an algorithmic level is still not very exact as information regarding the final implementation in hardware is missing. Information on signal delays in gates and wires are still excluded at this stage, for example.

Eventually the algorithmic description becomes refined to a register transfer description. On this level we can for the first time include propagation delays of the system components into the simulation. Thereby the simulation becomes more exact, but also more complex and takes more computing time.
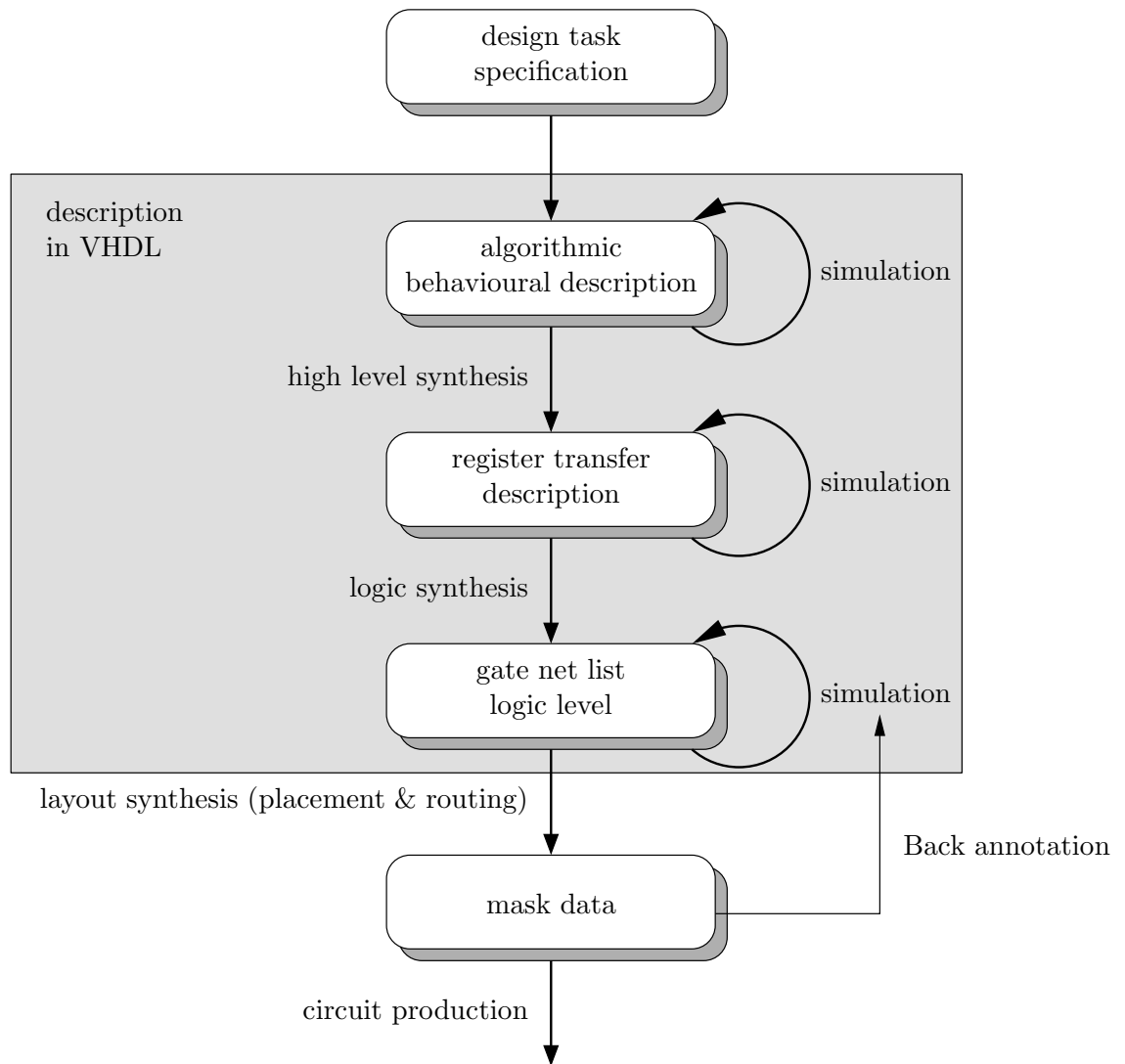
Figure 1: Design process for a typical circuit

Either by manual modeling in VHDL or by an automatic synthesis a netlist can be obtained from the description on the register transfer level. The netlist consists of simple gates which can be realized in the target technology.

During the layout, these gates are first placed geometrically on the chip and are then wired. This allows to calculate the exact propagation delay of the wiring in addition to the already known delay of the gates. After calculating these wiring delay times and adding them to the description (back annotation), another simulation can be used to verify the correct function of the circuit in reality. Once the correct function has been confirmed, the mask

data can be used to produce the chip.

2.2   Objective of the Laboratory

The objective of this laboratory is to introduce you to the essential aspects of computer-based circuit design using the hardware description language VHDL. Beginning with simple components on various levels of abstraction, we will proceed to more complex components. Eventually, we will use the various concepts to implement an encryption algorithm. In the next two steps, the implementation will be adjusted to a specific target technology. Based on these modifications the propagation delay in the single modules can be calculated, allowing an estimation of the shortest encryption time possible. The main aspects are the description of the circuit in VHDL as well as its simulation and implementation on hardware.

2.3   Overview of Cryptography

The explosive growth of transactions over the internet has given rise to the importance of data security. The challenge of exchanging information safely over an inherently unsafe medium is at the center of this development. This safe exchange includes not only making data accessible only to the communicating parties but also demands its integrity. Especially services like online banking or shopping depend on safe and reliable authentification of all parties to ensure that transactions are legally binding.

The importance of cryptography is obvious by the sheer multitude of available programs and algorithms. Applications like the Secure Socket Layer (SSL), Secure Shell and PGP are widely known.

The task of cryptography is the encryption of cleartext messages in a way that allows only a select group to read it. Usually the encryption/decryption algorithm as well as the ciphertext are public. If the encryption of a ciphertext uses the same key as the decryption, the encryption is called symmetric (Fig. 2). If encryption and decryption use different keys, it is called asymmetric (Fig. 3).

Symmetric approaches are normally faster than asymmetric ones, but have the disadvantage that the key needs to be transferred to the receiver through a secure channel. Once the key is known, everybody can read the message. The asymmetric approach uses two keys. One is public, the other private. If Person A wants to send a secret message to person B, person A uses B's

public key to encrypt the message. The encrypted text can now only be read using B's private key.
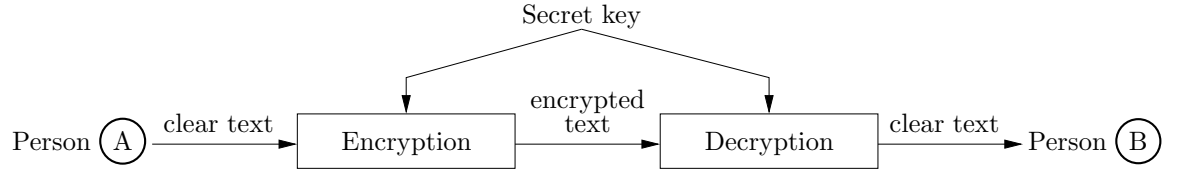
Figure 2: Symmetric encryption
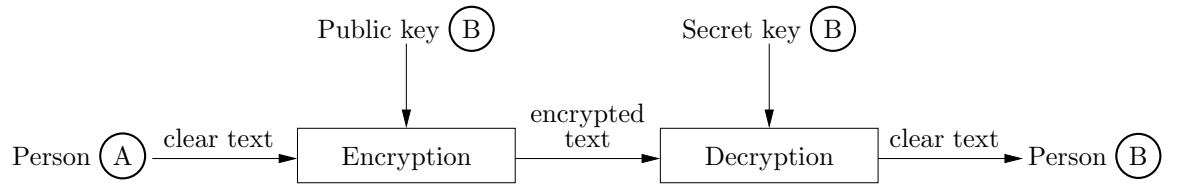
Figure 3: Asymmetric encryption

2.4 The Task

Based on a software prototype in Java, we will use VHDL to implement and simulate the IDEA-algorithm (International Data Encryption Algorithm, [10]). We will test the hardware-oriented implementations on the FPGA board. This algorithm dates from 1990 and is symmetric, i.e. it uses the same key for encryption and decryption. Four blocks of cleartext each 16 bit wide are encrypted at a time (Fig. 4). The IDEA-algorithm consists of 8 steps of encryption (also called rounds) with a final output transformation. The necessary partial keys for each cycle and the final transformation are generated from a 128 bit long input key.

For each round (Fig. 5) the following modules are required:

- bitwise XOR module for 16 bit blocks

- Addition of 16 bit numbers modulo 2^{16}

- Multiplication of 16 bit numbers modulo $2^{16} + 1$, where the input 0 is substituted by 2^{16}

The variables $X_i^{(r)}$, $Y_i^{(r)}$ and $Z_i^{(r)}$ are all 16 bit wide. $X_i^{(r)}$ represents the input of the cycle r, $Y_i^{(r)}$ the output and $Z_i^{(r)}$ the partial key from the key generator.
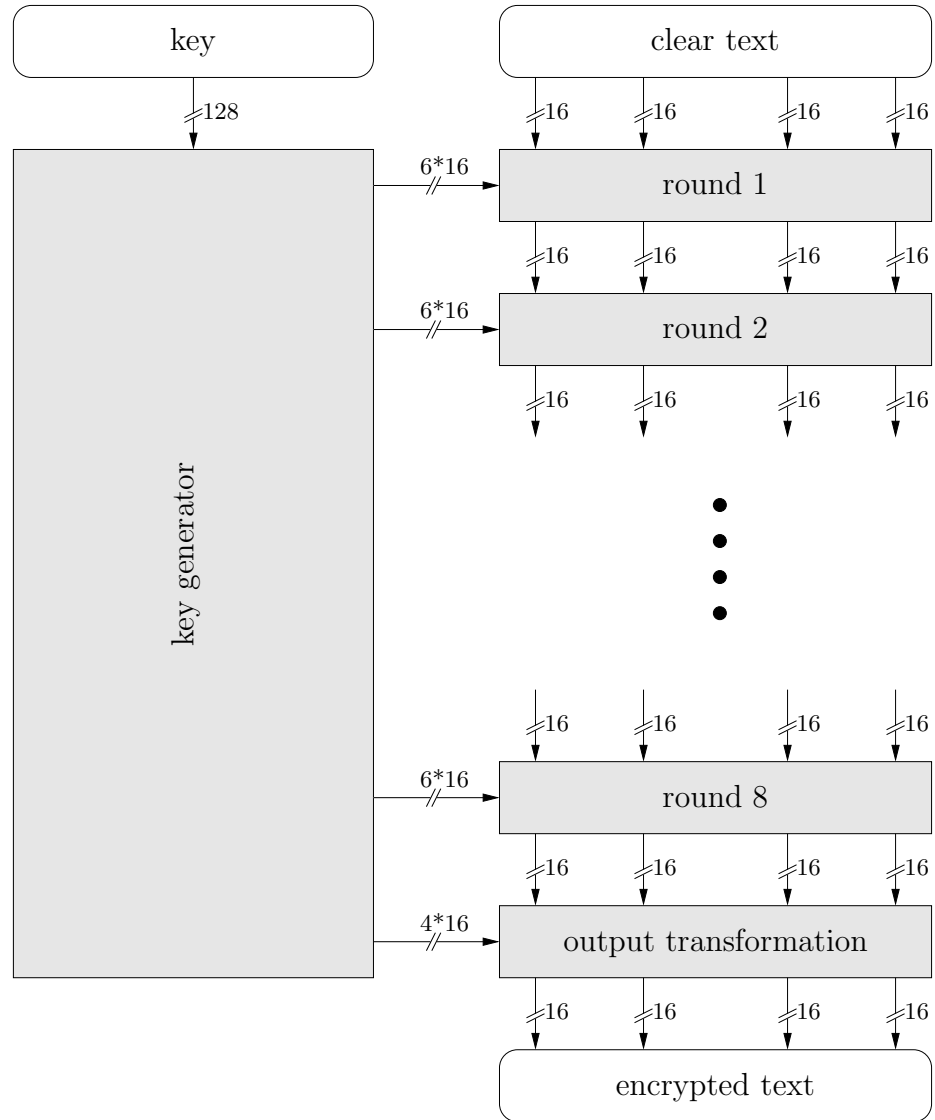
Figure 4: The IDEA algorithm

In the output transformation (Fig. 6) we need the following modules:

- Adding 16 bit numbers modulo 2^{16}

- Multiplying 16 bit numbers modulo $2^{16} + 1$,
  where the input 0 is substituted by 2^{16}

The key generator uses a 128 bit input word to generate 52 partial keys of 16 bit each. For the encryption the 128 bit input is split into 8 blocks of 16 bit each, which are used as the 8 first partial keys (Fig. 7).

labels for round-the question

$X_1^{(r)}$ $X_2^{(r)}$ $X_3^{(r)}$ $X_4^{(r)}$

$Z_1^{(r)}$ $Z_2^{(r)}$ $Z_3^{(r)}$ $Z_4^{(r)}$

T1 T2 T3 T4

T5

T6

$Z_5^{(r)}$ TX

TX T8

T9 T8 $Z_6^{(r)}$

T10 T9

$Z \to$ keys 6 in 1 round

⊕ XOR

adder

● modulo multiplier

$Y_1^{(r)}$ $Y_2^{(r)}$ $Y_3^{(r)}$ $Y_4^{(r)}$
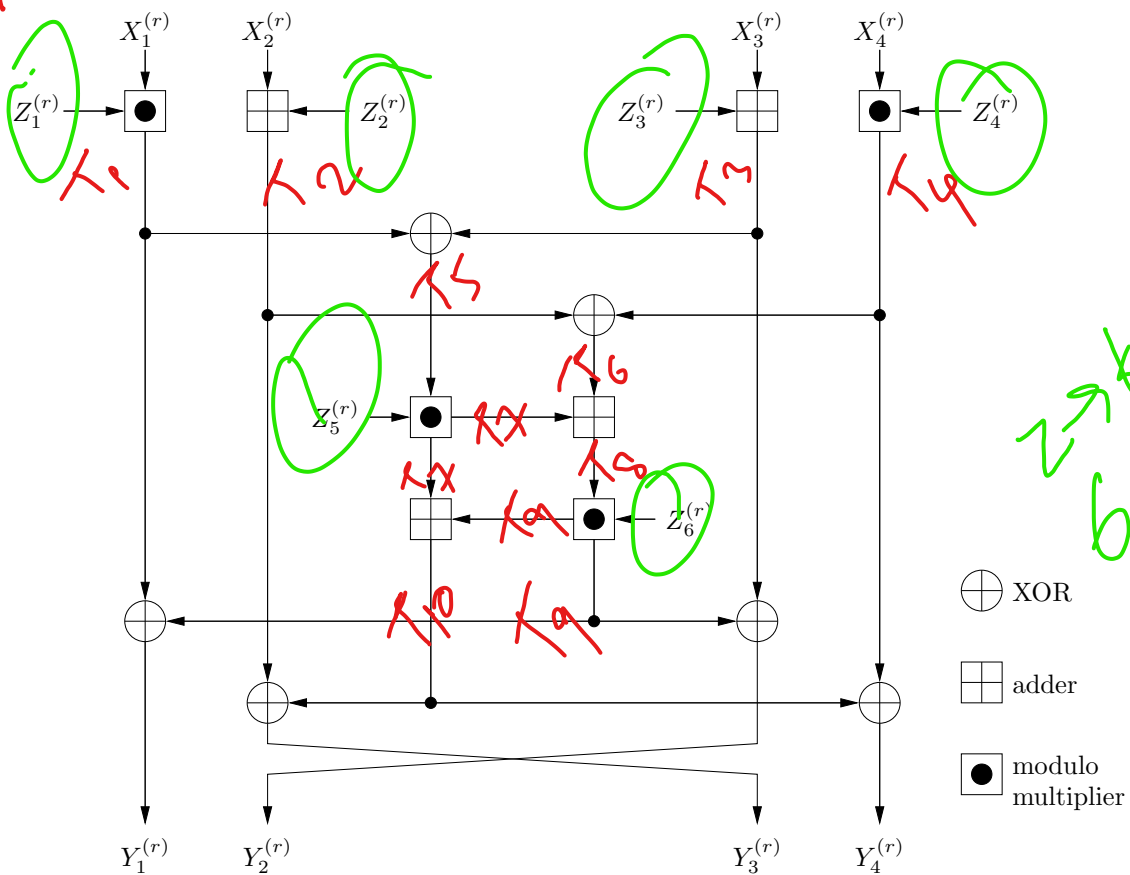
Figure 5: One round of the IDEA-algorithm

Question

$X_1^{(9)}$ $X_2^{(9)}$ $X_3^{(9)}$ $X_4^{(9)}$

$Z_1^{(9)}$ $Z_2^{(9)}$ $Z_3^{(9)}$ $Z_4^{(9)}$

adder

● modulo multiplier

$Y_1^{(9)}$ $Y_2^{(9)}$ $Y_3^{(9)}$ $Y_4^{(9)}$
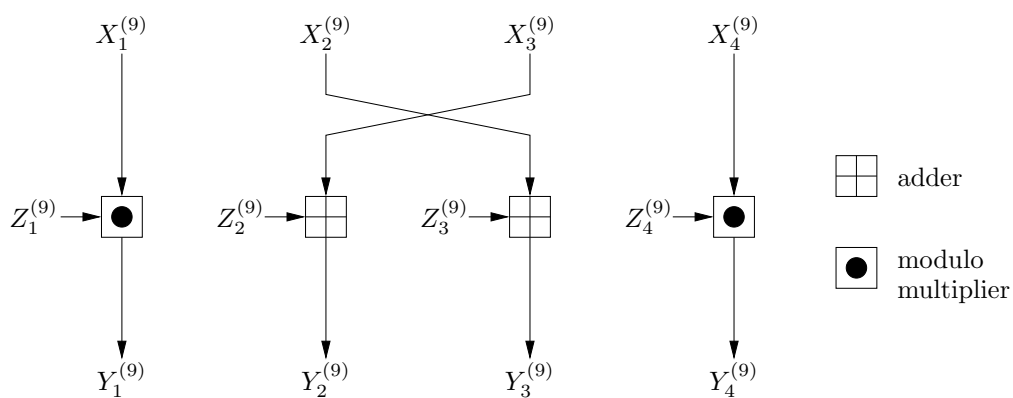
Figure 6: The output transformation of the IDEA-algorithm

11
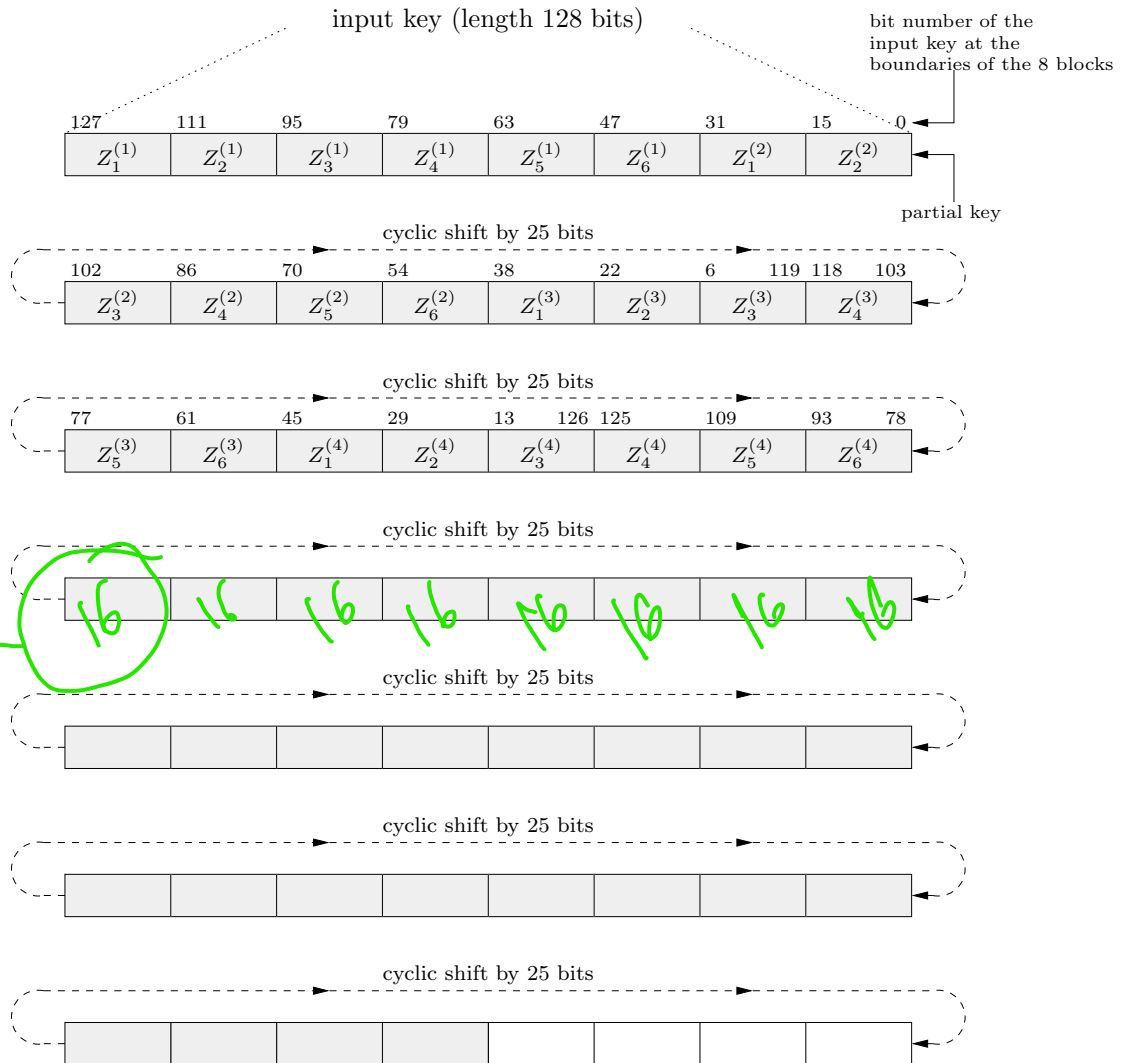
Figure 7: Generation of the partial keys

The sequence of the partial keys is as follows:

$$Z_1^{(1)}, Z_2^{(1)}, \ldots, Z_6^{(1)}, Z_1^{(2)}, \ldots, Z_6^{(2)}, \ldots, Z_1^{(8)}, \ldots, Z_6^{(8)}, Z_1^{(9)}, Z_2^{(9)}, Z_3^{(9)}, Z_4^{(9)}$$

After the extraction of the eight first partial keys the input is rotated clockwise by 25 bits and again split into 8 blocks. These new blocks become the new partial keys (Now $Z_3^{(2)}, \ldots, Z_6^{(2)}, Z_1^{(3)}, \ldots, Z_4^{(3)}$.). This is repeated until all 52 necessary partial keys are generated.

The unmodified IDEA algorithm can also be used for the decryption. One has to keep in mind that the key generator will produce other partial keys

though. The correct partial keys for decryption are the multiplicative or additive inverse of the new partial keys in reversed order. We won't go into further detail at this point.

Preparation:

- What are the variables $X_i^{(1)}$ and $Y_i^{(9)}$ with $i \in \{1 \ldots 4\}$?

- Which output feeds the input $X_i^{(r+1)}$, $r \in \{1, \ldots, 7\}$?

3   Example: Inverter

The following is a simple example for a 1-bit inverter including a testbench. It is taken to show how to use the tool Xilinx® ISE *WebPack^TM* for the design and simulation of VHDL code.

At the beginning, a new project is created. The individual steps are shown from Fig. 8 to Fig. 12. The design environment is loaded as shown in Fig. 8.
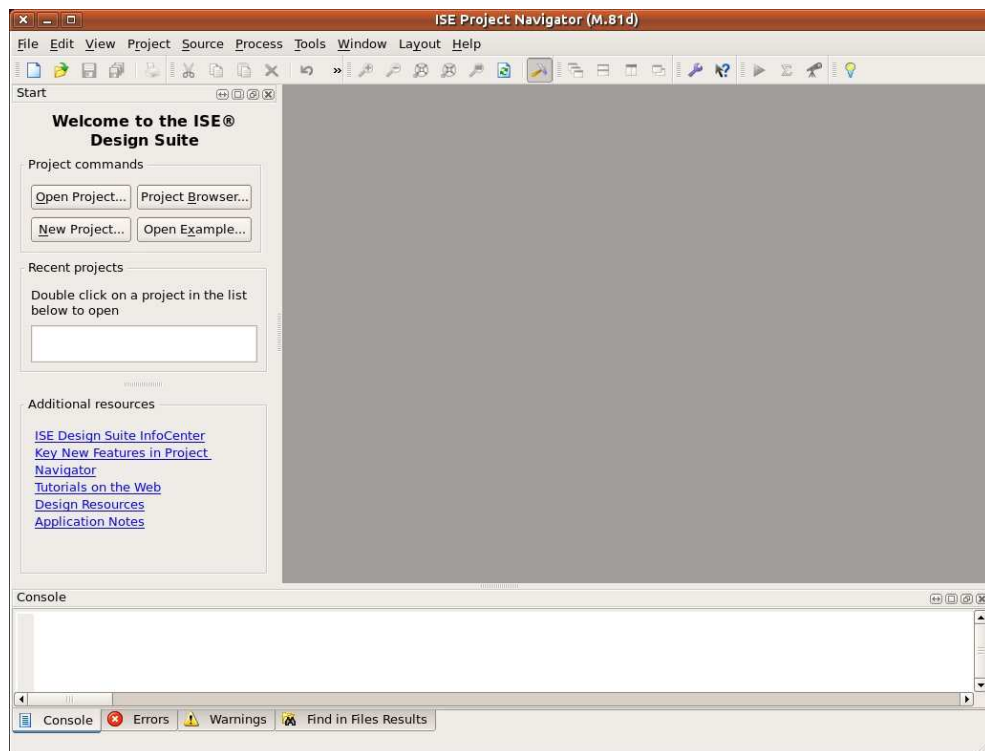
Figure 8: Design environment at start

Select `New Project...` under the menu `File` or just click on the button `New Project...`. A wizard in Fig. 9 is shown. Give `inverter_example` as the project name (different name than in the figure) and choose `HDL` as `Top-level source type`. Create a subdirectory `inverter_example` (different name than in the figure) under your home directory for the project location. In the next step the target FPGA and the simulator should be chosen. As shown in Fig. 10, choose "Spartan3E" for `Family`, "XC3S500E" for `Device`, "FG320" for `Package`, "4" for `Speed`, "XST (VHDL/Verilog)" for `Synthesis Tool`, "ISim (VHDL/Verilog)" for `Simulator` and "VHDL" for `Preferred Language`.

Just go ahead by clicking the `Next` button, a project summary is shown as in Fig. 11. Then click the `Finish` button and the whole project is created.

A new source file can be created. Click the `New Source` button under the menu `Project` and a window as in Fig. 13 is shown. Select `VHDL Module` as source type and use `inverter` as file name.

The definition of the inputs and outputs can then be made in the next step. Type `I` as input port name and `O` as output port name, as shown in Fig. 14. The port definition can be changed any time in the VHDL code. With another click on the `Next` button a summary of the new source file is shown in Fig. 15).

Fig. 16 shows the design environment after the project has been created. The wizard has generated the module `Inverter` automatically. Add one line to the VHDL code to let the output signal be the inverted signal of the input. The final code looks like this (without comments):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity inverter is
  port (I: in STD_LOGIC;
        O: out STD_LOGIC);
end inverter;

architecture Behavioral of inverter is
begin
    O <= not I;
end Behavioral;
```

For testing the inverter a testbench must be created. Select the item `New Source` under menu `Project`. In the window `New Source Wizard` select the type `VHDL Test Bench` and give `tb_inverter` for the file name. It is recommended that the name of a bestbench file called `tb_` plus the original module name. Fig. 17 shows the corresponding configuration.

After clicking the button `Next` the modules which are availabe for test are listed. Choose the inverter as shown in Fig. 18. Fig. 19 shows the testbench summay. Click the `Finish` button to finish. Click `Simulation` for `View`. The design environment after testbench generation is shown in Fig. 20. Edit the code like this (without comments):

```
library IEEE;
USE ieee.std_logic_1164.ALL;

ENTITY tb_inverter IS
END tb_inverter;

ARCHITECTURE behavior OF tb_inverter IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT inverter
    PORT(
         I : IN  std_logic;
         O : OUT  std_logic
        );
     END COMPONENT;
   --Inputs
   signal I : std_logic := '0';
   --Outputs
   signal O : std_logic;

   constant I_period : time := 10 ns;
BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: inverter PORT MAP (
          I => I,
          O => O
        );
   I_process : process
   begin
      I <= '0';
      wait for I_period/2;
      I <= '1';
      wait for I_period/2;
   end process;
END;
```

The testbench generates an oscillating signal with the frequency 100 MHz at the input.

Now select the source file tb_inverter. In the panel below all the processes relative to the selected file are listed as shown in Fig. 21. Under the ISIM Simulator, there are two items Behavioral Check Syntax and

`Simulate Behavioral Model`. It is recommended that after the change in the VHDL code double click the `Behavioral Check Syntax` to check the syntax. If no errors could be found, then double click `Simulate Behavioral Model` to start the simulation process. As default the first 1000 ns will be simulated. Fig. 22 shows the simulation window.

Sometimes it is necessary to enlarge the simulation time to be more than the default value. The buttons for controlling the simulation process are shown in Fig. 23 in the toolbar area of the simulator. You can give the simulation time and restart the simulation.

You can also manipulate the waveform by clicking the buttons for zoom in, zoom out or zoom to full view. Refine the time axis, until the signals are displayed as in Fig. 24. The function of the inverter can then be checked using the simulation results. This example is trivial. But it is to mention here that the modules which will be designed in the following sections could have a large number of input and output signals as well as internal signals. As default only the input signals and the output signals are automatically displayed by the simulator. The internal signals can be displayed by "Drag and Drop" from the left panel to the right simulation window. In this case the simulation must be restarted again in order to get the values of the newly added signals.
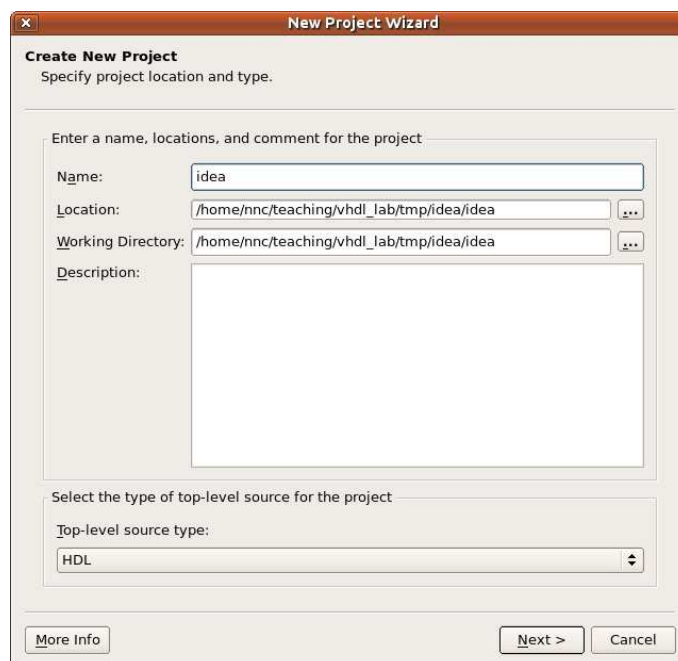
Figure 9: Create a new project
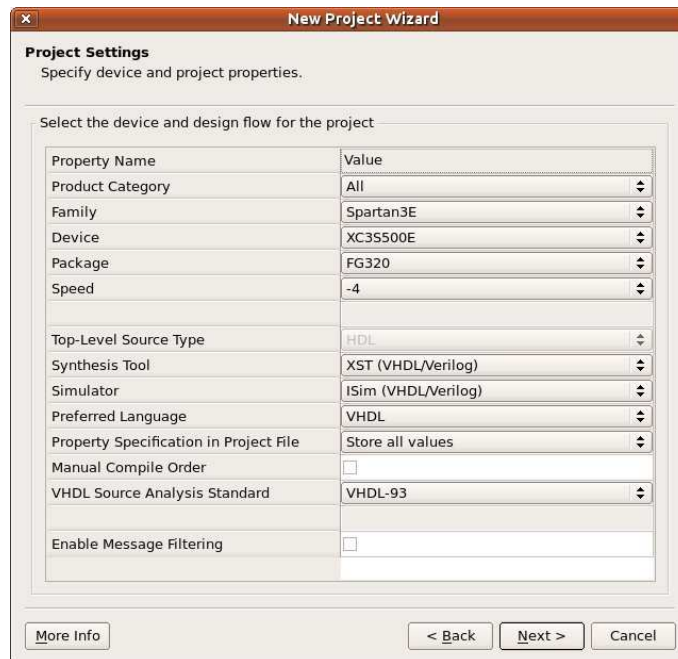
Figure 10: Configuration of FPGA and simulator tool
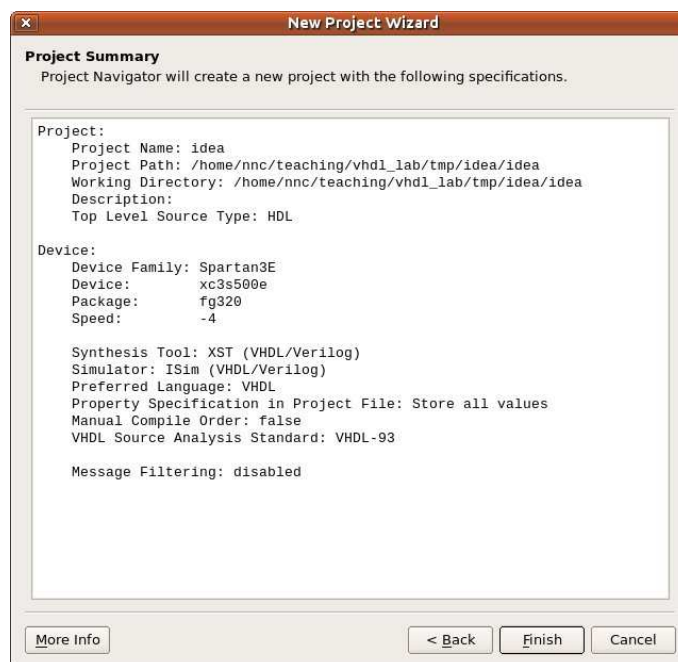
Figure 11: Project summary
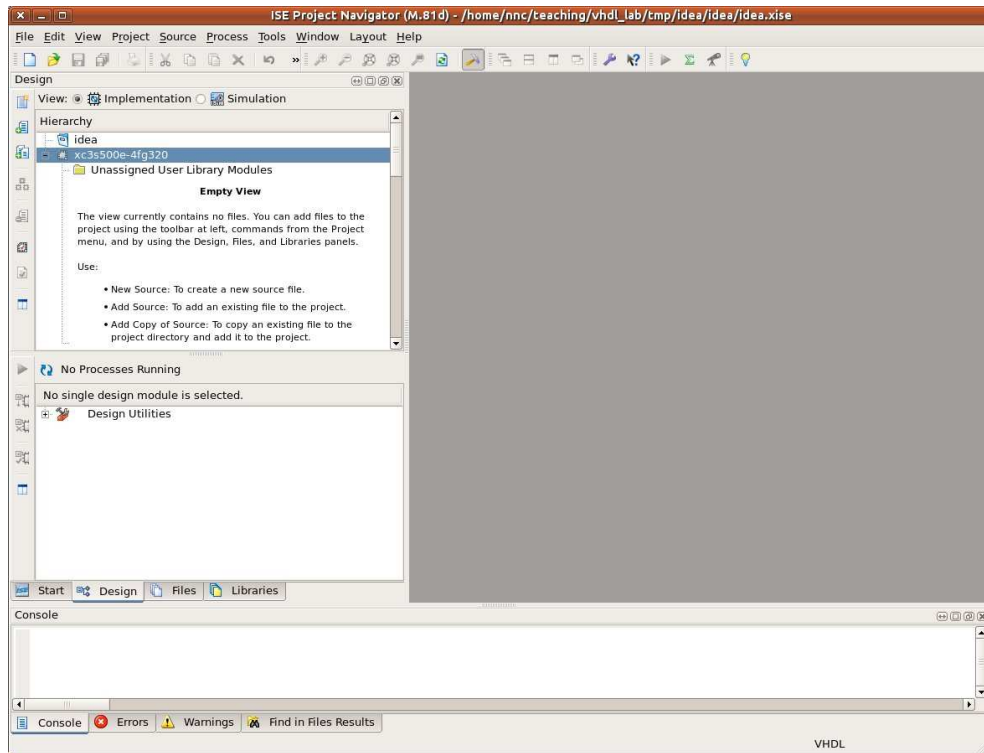
Figure 12: Design environment after a new project is created

Figure 13: Use the wizard
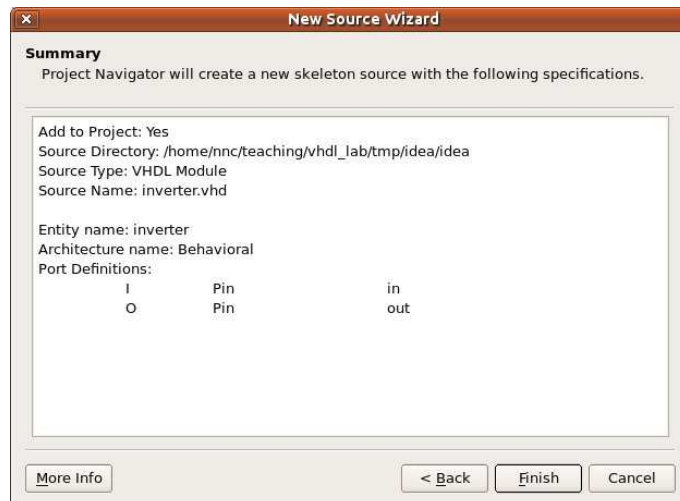
Figure 14: Port definition

Figure 15: Summary of new source file
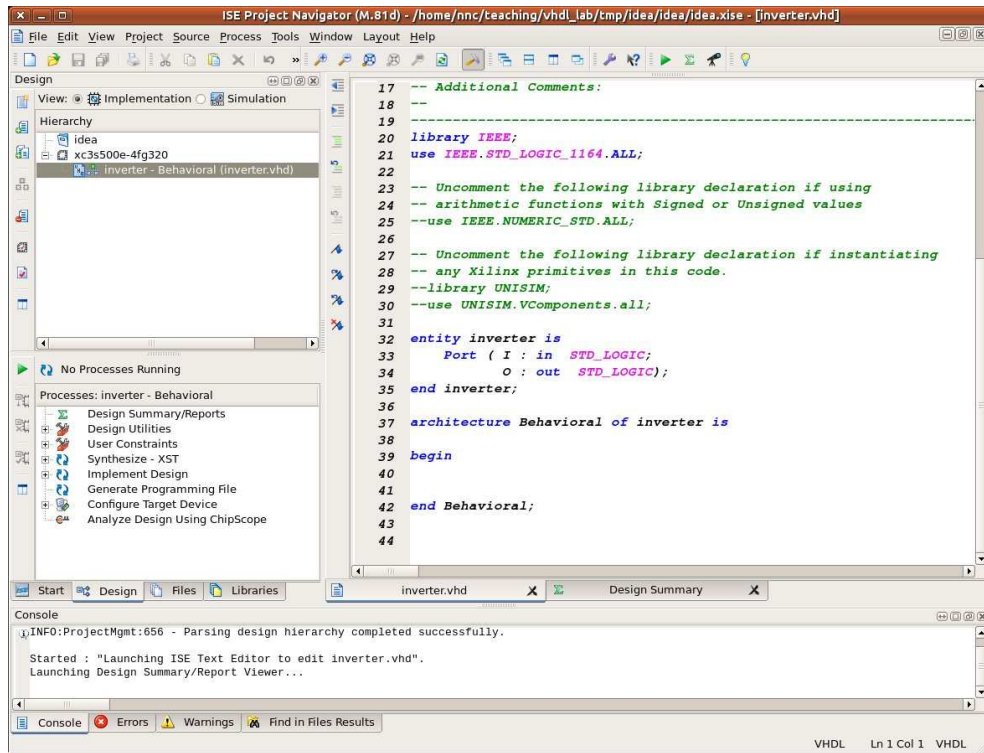
Figure 16: Design environment after a new source is created
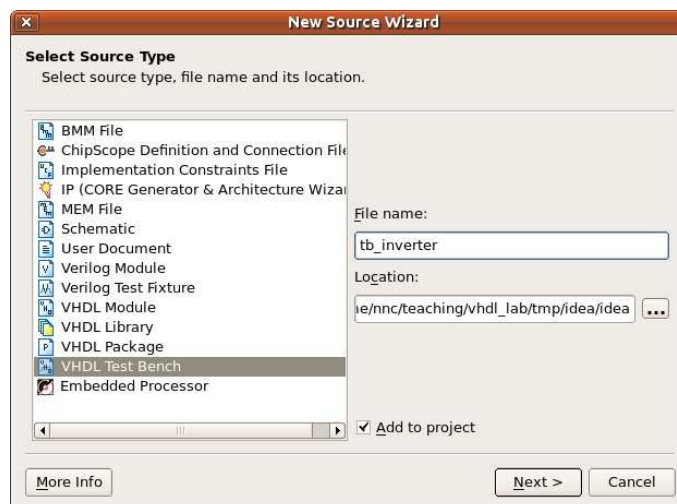
Figure 17: Create a testbench
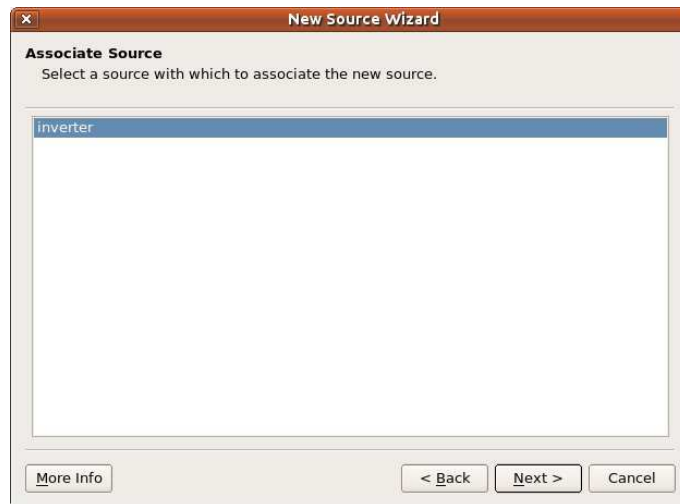
Figure 18: Choose the module under test

Figure 19: Testbench summary
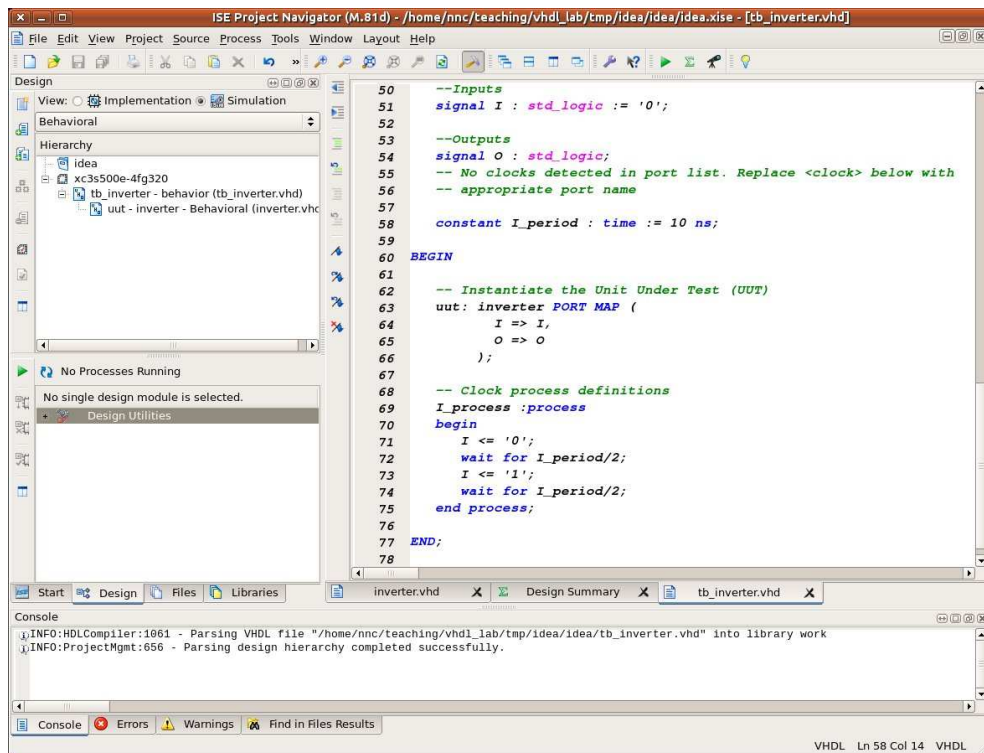
Figure 20: Design environment after testbench creation
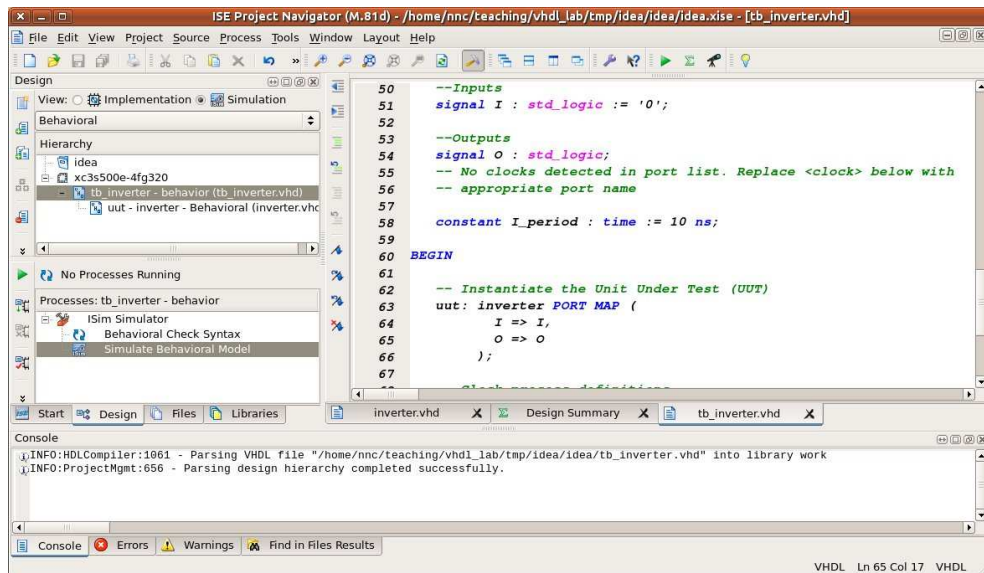
Figure 21: Start the behavioral simulation
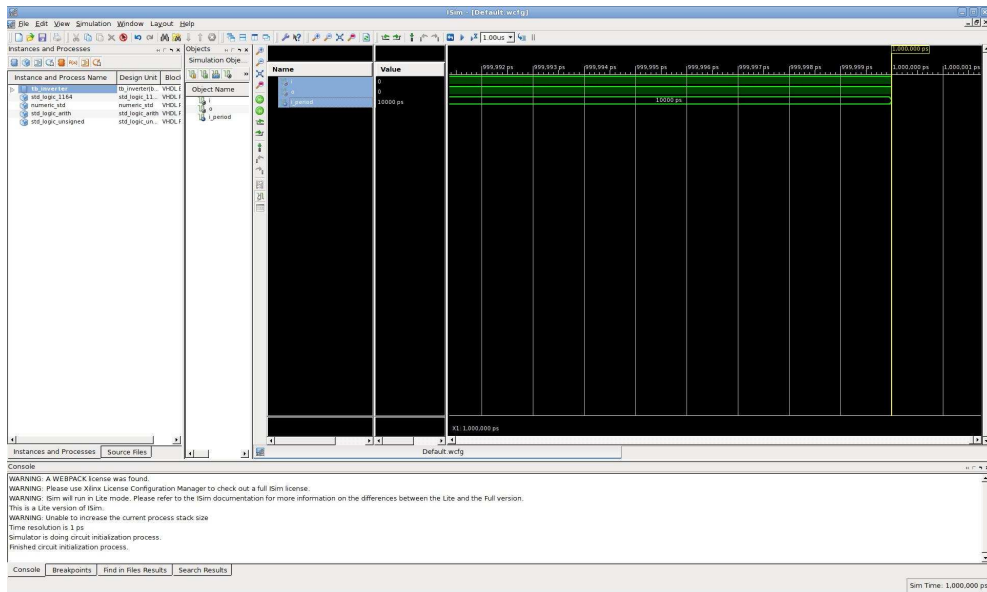
Figure 22: Simulator

Figure 23: Toolbar for simulation control
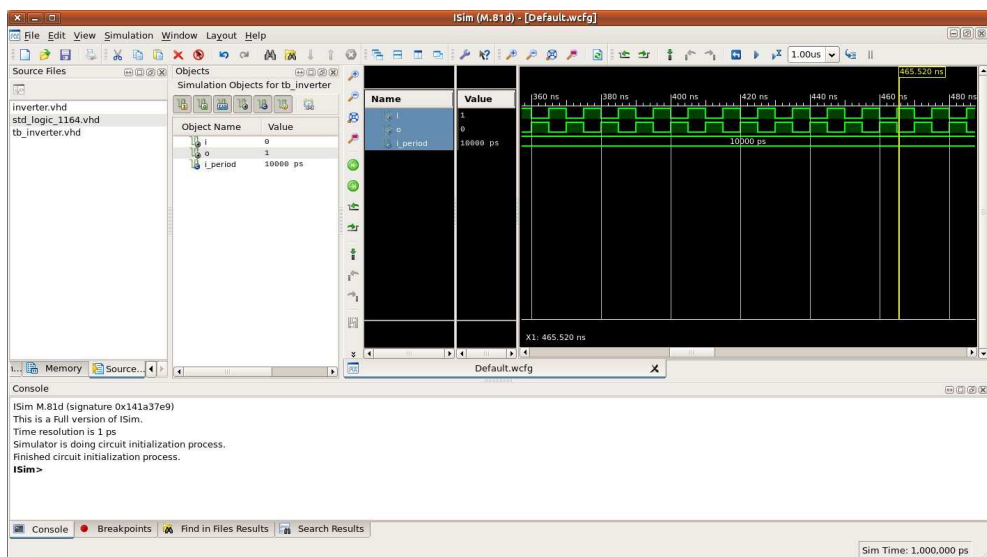
Figure 24: Refined time axis

4 Direct Implementation

During the course for this laboratory, several arithmetic models can be used for the implementation of the encryption process. Firstly these will be created on an algorithmic level in order to allow an early first simulation of the whole algorithm. Later, these simple models will be refined and will eventually be implemented on register transfer level, which also includes signal delay times. Normally, two files are created for every task. The first describes the module in question while the second contains a testbench. This testbench is used to evaluate and ensure the functional correctness of the module. For the Direct Implementation a project already exists in the folder submitdirect. Open the project by clicking on the `Open Project` button in figure 8. Alternatively select `File -> Open Project`. Then go into the correct directory and select the file `idea.xise` and click `Open`.

4.1 The XOR Module

The IDEA algorithm uses 48 XOR modules for a 16-bit wide word (input and output). Create a XOR module with two input vectors and one output vector of the `std_logic_vector` type for 16 bit width and a suitable testbench by following these steps:

- Create the file `xorop.vhd` (VHDL Module) in the editor of the IDEA $\boxed{\text{VHDL 1}}$ project.

- Create an `entity xorop`, which describes the input and output of the XOR module.

- Describe in an `architecture` the behavior of the XOR module in the context of a `process`.

- After saving the file, switch the view to `Simulation` in the left upper corner and start the process `Behavioral Check Syntax` on the left side. Correct any mistakes and check the syntax again until no further errors are displayed.

After creating the XOR module, we need to implement a testbench which can apply different input patterns to the module. By comparing the output values of the XOR module to the expected values we can confirm the functional correctness function of the module. The testbench is implemented as follows:

- Create a file tb_xorop.vhd (VHDL Test Bench) containing one `entity` VHDL 2 `tb_xorop`. Is it necessary to define the inputs and outputs of the `entity` as outbound?

- Create an `architecture behavior`, in which the XOR module `xorop` is implemented as a `component`. Now apply various test signals to the inputs of the module (see VHDL Introduction). Please note that we do not have any clock in the direct implementation.

After the syntax errors have been checked, start the Xilinx® ISE *Simulator* as in the inverter example.

4.2 Adder

Next we need 34 modules for 16-bit addition modulo 2^{16} for the IDEA algorithm . The task is to create an adder module providing these features. Follow these steps:

VHDL 3
- Create a file `addop.vhd` that includes an `entity addop` with input and output of the `std_logic_vector` type.

- Describe the behavior of the adder module with an `architecture` in the context of a `process`.

- You need to use either the `IEEE.std_logic_unsigned.all` package or use a variable of the type `unsigned` internally for the addition (see VHDL Introduction).

Once you checked the syntax of the adder module, create a testbench:

VHDL 4
- Create a file tb_addop.vhd including an `entity tb_addop`.

- Create an `architecture behavior`, in which you embed the adder module `addop` as a `component` and run tests with various signals on its inputs. Pay special attention to the behavior of the adder in overflow situations (i.e. the result cannot be stored using only 16 bit).

Use the simulator to test the new module.

4.3   Modulo-Multiplier

Used 34 times, the modulo multiplier is the most complex module in the IDEA algorithm. Two 16-bit input vectors are multiplied, where the input of 0 is substituted by 2^{16}. The 33 bit (33 bits needed because of $2^{16} * 2^{16}$) wide result is taken modulo $2^{16} + 1$ and cut down to use only the lowest 16 bit as output.

A separate implementation of the multiplication and the modulo operation would lead to two very complex and therefore expensive modules. In addition, these would have to be connected with a 33 bit wide datapath. A better solution is the low-high algorithm. Based on a single, modified multiplier the multiplication and the modulo operation can be performed in one module.

4.3.1   The Low-High Algorithm

a and b are two integers out of $\{1, \ldots, 2^n\}$:

$$ab \bmod (2^n + 1) = \begin{cases} (ab \bmod 2^n) - (ab \operatorname{div} 2^n), & (ab \bmod 2^n) \geq (ab \operatorname{div} 2^n) \\ (ab \bmod 2^n) - (ab \operatorname{div} 2^n) + 2^n + 1, & (ab \bmod 2^n) < (ab \operatorname{div} 2^n) \end{cases} \tag{1}$$

here $(ab \operatorname{div} 2^n)$ refers to the whole-numbered part of the quotient when ab is divided by 2^n.

Preparation:

$\boxed{?\ 2}$

- How are a and b calculated from the input vectors $X_i^{(r)}$ and $Z_i^{(r)}$ of the modulo-multiplier?

- a and b are to be two integers out of $\{1, \ldots, 2^n\}$. How many bits are necessary for the unsigned represetation of a and b? How many for the product ab (worst-case)?

- What would be the value of n for the modulo-multiplier in the IDEA algorithm?

- Which bits are masked by the modulo 2^n operation, which do remain? How many bits are necessary for the result of $(ab \bmod 2^n)$? (Hint: The LSB has number 0!)

- Which simple bit operation is the equivalent of $(ab \text{ div } 2^n)$? Which bits of the product ab remain? Where are they after the division? How many bits are needed for the result? Take also the case of a or b equal to 2^n into consideration!

- What is the result of $(ab \bmod (2^n + 1))$, if $(ab \bmod 2^n) = (ab \text{ div } 2^n)$? Is this case possible if a and b were integers in the range of $\{1, \ldots, 2^n\}$ and $2^n + 1$ was a prime number? Explain your answer!

Proof: We assume that a and b are integers in the range of $\{1, \ldots, 2^n\}$. For any a and b there is a clear allocation of p and q (whole-numbered):

$$ab = q(2^n + 1) + p, \qquad \begin{array}{l} 0 \le p < 2^n + 1 \\ 0 \le q < 2^n \end{array} \tag{2}$$

This leads to the following conclusion: $q + p < 2^{n+1}$ and $p = ab \bmod (2^n + 1)$. Furthermore we can show[1] by using (2) that

$$(ab \text{ div } 2^n) = \begin{cases} q, & q + p < 2^n \\ q + 1, & q + p \ge 2^n \end{cases} \tag{3}$$

and

$$(ab \bmod 2^n) = \begin{cases} q + p, & q + p < 2^n \\ q + p - 2^n, & q + p \ge 2^n \end{cases} . \tag{4}$$

Considering (3) and (4) we get

$$p = \begin{cases} (ab \bmod 2^n) - (ab \text{ div } 2^n), & q + p < 2^n \\ (ab \bmod 2^n) - (ab \text{ div } 2^n) + 2^n + 1, & q + p \ge 2^n \end{cases}, \tag{5}$$

whereas $q + p$ is exactly then smaller than 2^n, when $(ab \bmod 2^n) \ge (ab \text{ div } 2^n)$. This follows from (5), as $p \ge 0$ is given by (2).

4.3.2 Implementation Of The Modulo-Multiplier

Create a multiplier with integrated modulo $2^{16} + 1$ operation using the low-high algorithm. Follow these steps:

---

[1] $(q * 2^n) \text{ div } 2^n = q$
$(q * 2^n) \bmod 2^n = 0$

- Describe the behavior of the given module `mulop.vhd` within the `architecture` in the context of a `process`. Keep in mind that an input of 0 is to be substituted by a value of 2^{16}. How many bits are necessary to represent 2^{16}? What is the result if one or both inputs were 0?

- Consider that both variables have the same width in bits. The result of a multiplication has double width.

- Hint: You can use the `&` operator to join bit vectors together. Use the VHDL tool for limiting the bit range to select desired bits within a vector. Example: `a := b (7 downto 4)`.

- Initialize all signals and variables.

Create a testbench:

- Create a file tb_mulop.vhd including an `entity tb_mulop`.
  VHDL 5

- Create an `architecture behavior` in which you instantiate the multiplier module `mulop` as a `component`. Run different test signals on the inputs.

- Pay special attention to the behavior of the multiplier when one or both input signals are 0.

Use the simulator to test the new module.

4.4 Round Module

The IDEA algorithm uses 8 round modules for encryption or decryption. Create a round module as shown in Fig. 5 on page 11 using the previously designed arithmetic modules. Follow these steps:

- Create a file round.vhd including an `entity round`, which processes 6 partial keys, the input and the output as 16 bit wide `std_logic_vector` signals.
  VHDL 6

- Create an `architecture` and include the modules `xorop`, `addop` and `mulop` as `components`.

Create a testbench and test various input patterns. The simulated results should be in accordance to the expected results (you can use the online lab cource interface under `Reference model` to calculate the expected results).
VHDL 7

4.5 <mark>Output Transformation</mark>

Furthermore the IDEA algorithm needs a output transformation module. Create a module which realizes the output transformation as shown in Fig. 6, page 11.

<div style="float:left; border:1px solid black; padding:2px;">VHDL 8</div>

- Create a file <mark>`trafo.vhd`</mark> including an `entity trafo`, which processes 4 partial keys, input and output as 16 bit wide `std_logic_vector` signals.

- Include `addop` and `mulop` as `components` in the `architecture`.

- Create the necessary instantiations of the modules and connect these.

<div style="float:left; border:1px solid black; padding:2px;">VHDL 9</div> Create a testbench and check for various input patterns if the results comply with the expected results.

4.6 Encryption

Having designed all modules needed for the IDEA algorithm, the task is now to design a first implementation of the encryption process for a fixed key. Therefore the VHDL file `idea.vhd` is already given. Modify the `architecture` of this file by following these steps:

<div style="float:left; border:1px solid black; padding:2px;">VHDL 10</div>

- Include `round` and `trafo` as `component`.

- Create all necessary instantiations of the modules and connect these.

- Connect the partial keys out of the 128 bit key to the corresponding inputs of the round modules and the transformation module.

For testing the complete algorithm create a testbench including several different inputs and compare the results with the results given by the reference model on the website.

5   Hardware-Oriented Implementation

Having studied a first implementation of the IDEA algorithm in the previous section, this section is dedicated to developing a description that can be realized on a FPGA-chip. Today, programmable FPGA-chips[2] are gaining in importance to the classical ASICs. FPGAs are so-called semicustom chips, which allow for complex designs with a short development time. FPGAs are capable of repeated modifications. This makes them more cost-efficient for small production numbers and brings the additional benefit of short production times. The scale of realizable applications ranges from simple logic circuits like counters, adders and decoders to small processors. An especially wide-spread architecture are lookup-table-based FPGAs, for example the Spartan-3E family from Xilinx. One chip (XC3S500E) from this family on the test board is used for this design laboratory.
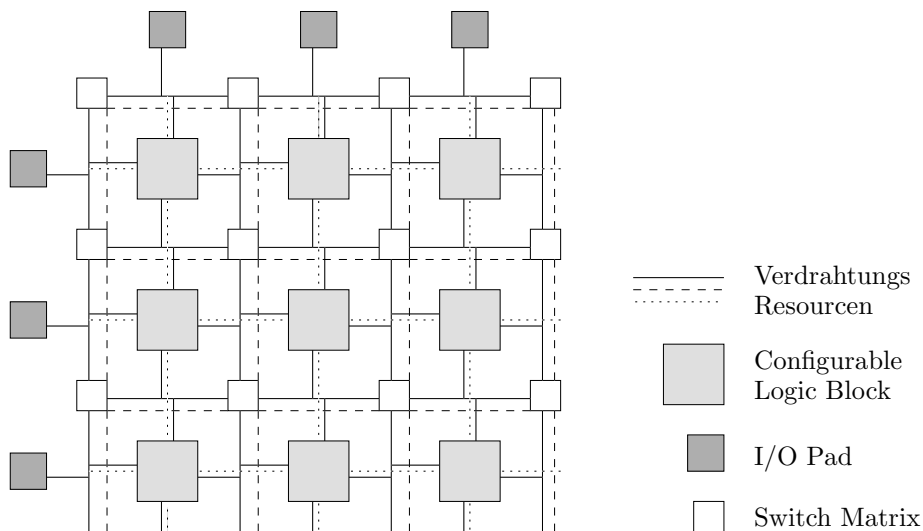
Figure 25: Structure of a Xilinx FPGA

As it appears from Figure 25, Xilinx FPGAs are structured into a chessboard-like array of logical blocks, called CLBs[3], and switching elements. I/O pads can be used for the input of external signals and the output of internal signals. Using the switching elements and linking resources, the logic blocks can be connected to each other as well as to the I/O pads.

Every logic block consists of four slices and correspondingly eight Static

---

[2]Field-Programmable Gate Array
[3]Configurable Logic Block

<u>RAM</u>[4] elements, each of which can realize any desired operation with up to 4 input variables through a look-up table. This look-up table assigns a resulting value to every possible combination of input values. In order to program a certain function, just 16 bits — the number of possible input combinations — are loaded into the SRAM element.

The wiring is realized using the following types of connections, which can also be connected to each other by the switching elements:

- *Direct Connects* connect neighboring CLBs.

- *Single Length Lines* connect neighboring switching elements.

- *Double Length Lines* connect every second switching element of a column or a line.

- *Quad Length Lines* connect every fourth switching element of a column or a line.

- *Long Lines* stretch over the whole length of the FPGA chip.

Programming a FPGA requires the configuration of both the logic blocks and the switching elements.

For an implementation of the IDEA algorithm on a Xilinx FPGA of the Type XC3S500E-FG320 the current description needs to be modified. This FPGA provides 1164 logic blocks (CLBs) and correspondingly 9312 LUTs that can be accessed over up to 232 input or output pins. In addition, 9312 flipflops, 74496 bits S-RAM and 20 hardware multipliers are available on the FPGA. The Table 1 shows approximately how many resources each of the basic modules of the IDEA algorithm needs. For the modulo-multiplier there are two alternatives. Either one of the 20 hardware multipliers is used, or the multiplication is realized with the help of the LUTs in CLBs.

The term design flow refers to the whole process towards a working design on the FPGA. The most important steps of the design flow in ISE are arranged in the same order (shown in Fig. 26) as they should be executed.

After the VHDL code has been written, the next step is to synthesize it. As a result of the synthesis, a netlist, which consists of logical elements, is created. This netlist does not depend on any special FPGA technology. The 'synthesis' here is equivalent to the logic synthesis in ASIC design.

---

[4]<u>R</u>andom <u>A</u>ccess <u>M</u>emory

Module	LUTs	FlipFlops	Hardcore Multiplier
XOR	16	0	0
Adder	16	0	0
Modulo-Multiplier 1	106	0	1
Modulo-Multiplier 2	390	0	0

Table 1: Resources for the basic IDEA modules
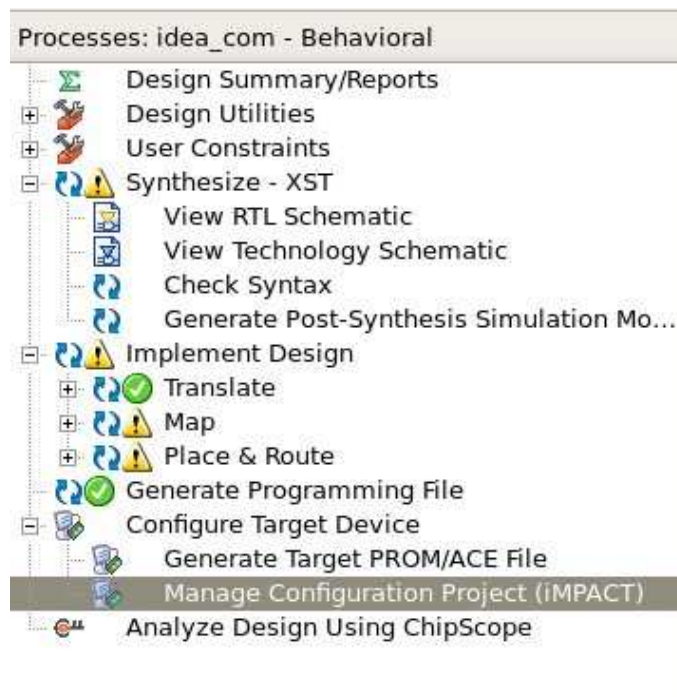
Figure 26: Design Flow

After the synthesis the next step is the implementation step. The 'implementation' from Xilinx has the same meaning as the layout synthesis. In this step there are three processes included which are explained in the following:

- **TRANSLATE:** In this process the netlist from the synthesis will be converted into a „Xilinx Native Generic Database (NGD)" file. The logical description as well as the design hierarchy in the original netlist are retained. This file can be understood as a new netlist that is specified for the Xilinx FPGA.

- **MAP:** In this process the logical elements such as gates are mapped

into the components such as LUTs on a certain target FPGA technology.

- **PLACE & ROUTE:** In this process the design will be placed and routed.

After the implementation step all the information for a design, corresponding to a certain target FPGA technology, is available. A so-called bitstream file is then generated in order to program the FPGA. This file can be downloaded into the FPGA using the tool Xilinx® *iMPACT^{TM}* .

For further information please check the „Development System Reference Guide" which can be downloaded from the online interface of this laboratory.

5.1 Synthesis and implementation of the direct algorithm

After the simulation of the VHDL model `idea.vhd` has been finished successfully, synthesize it as the next step. The design will be converted to a netlist of components which could be realized on FPGA, as long as there are enough resources for that.

VHDL 11

- Synthesize the VHDL model `idea.vhd`. Start the process `Synthesize - XST` and then the `Map`.

- Could the design in the current stage be synthesized or implemented? If not, which kind of resources circumvents the synthesis or the implementation? Notice: Start the process `View Design Summary` where you can see a summary of the resource usage.

An implementation of the current description with 48 XOR modules (768 LUTs), 34 adders (544 LUTs) and the same number of modulo-multipliers (3604 LUTs + 34 hardcore-multipliers) would require about 4916 LUTs. Although there are enough LUTs available, there are not enough hardcore multipliers to implement the design. When the remaining 14 multipliers are realized using LUTs, the total number of LUTs will reach 8892. Till here already 95% of the FPGA resources are used. Furthermore the key generator, which has an input vector of 128 bits and generates totally 52 subkeys of 16 bits, adds more usage of resources. So the available logical and routing resources will be not enough any more. That means, the direct algorithm is not suitable for implementation.

It is necessary to change the design so that less CLBs and multipliers are needed and the design could then be implemented.

Since the modulo-multiplier is by far the largest module and uses a hardcore multiplier which is the rare resource on FPGA, it is reasonable to find a description which needs less of these modules. In the following we will change the design in two ways. In section 5.2 the same algorithm is designed with only one round module and one output transformation. Then in the section 5.3 the round module will be adapted to use only one multiplier. Later on this new round module will be used for all the 8 rounds and the final transformation.

5.2   Resource Constrained Scheduling I
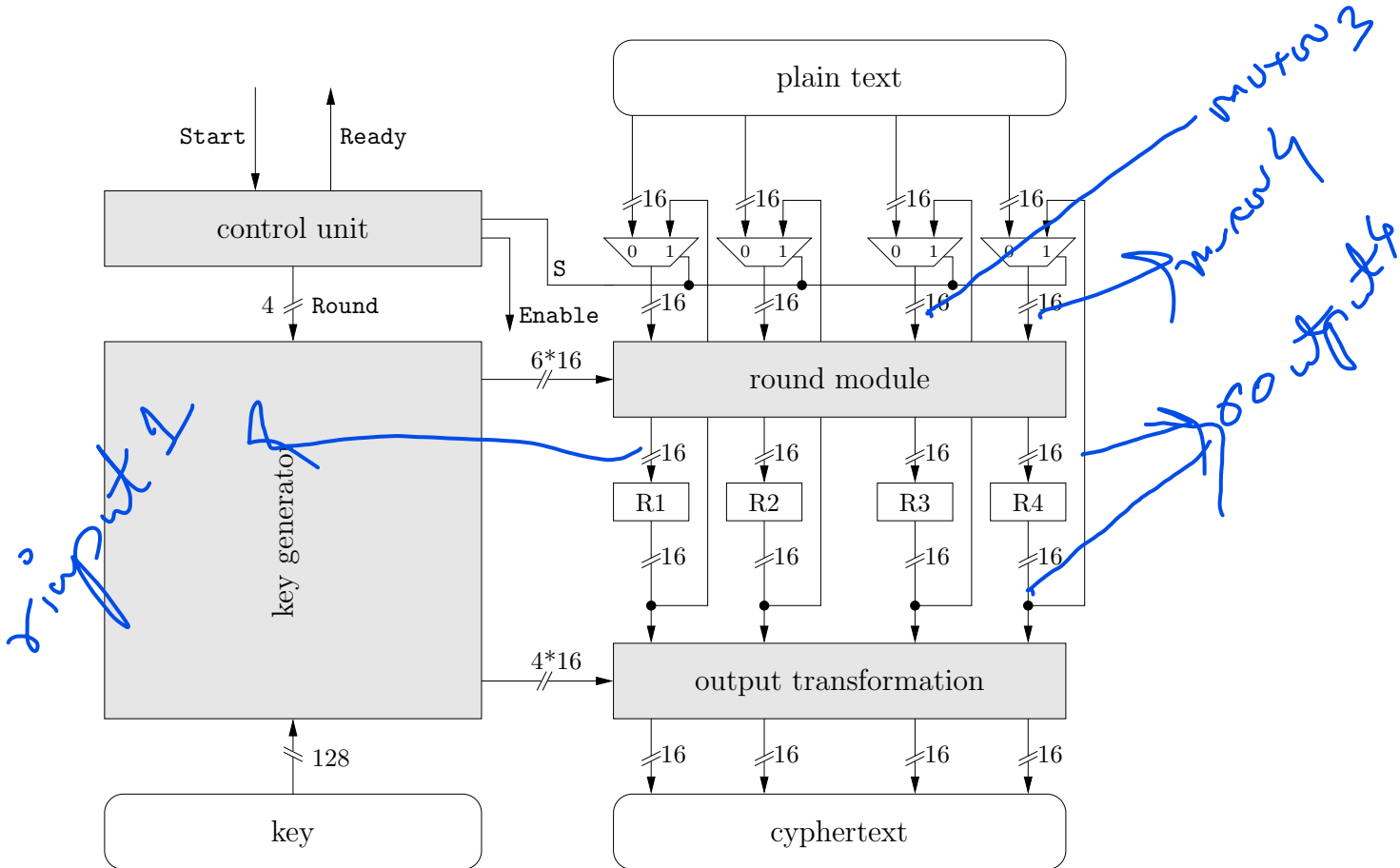
Figure 27: Design with only one round module

Only one round module and the output transformation should be used in

order to implement the design. This is possible, because all the round modules have the same architecture. The results of each round computation can be saved in registers and used as inputs for the round module at the next step. After 8 rounds the saved values are then given to the output transformation. A control module is used for controlling this data flow. Figure 27 shows the new design with only one round module and the output transformation. This kind of partitioning the design under limited resources is called Resource Constrained Scheduling. The multiple use of the same module is called Resource Sharing.

The control module and the new design should be now implemented. For this purpose open the project `idea_rcs1` in the folder submitrcs1. At first the register, 2-to-1 multiplexer and a new key generator are to be designed.

5.2.1   16 Bit Register

Develop a description for a 16 bit register with a *clock* and an *enable* input of the `std_logic` type and an input and an output each 16 bit wide of the type `std_logic_vector`. On a rising edge the register should set the output value to the input value if the *enable* signal is set.

Amplitude Controlled Register (Latch)	Clock Controlled Register
``process (Clock, D, EN-ABLE)
begin
if (Clock='1') then
if (ENABLE='1')
then
Q <= D;
end if;
end if;
end process;```	```process (Clock)
begin
if (Clock='1' and Clock'EVENT)
then
if (ENABLE='1') then
Q <= D;
end if;
end if;
end process;```

Hint: By including the *clock* signal in the `sensitivity list` of a `process` you can ensure that the `process` is only executed at a change of the signal value, i.e. an edge. Then you also need to check for rising edges by reading the actual value of the *clock* signal. For a rising edge it must be '1'. To avoid creating an amplitude controlled register instead of a clock controlled one in the synthesis, you should also check for a `Clock'EVENT` apart from the value of the *clock* signal (which should be '1'). For the simulation in itself this is not necessary, as the `process` is only executed when the signals in the

`sensitivity-list` change.

Check the correct function of the register by creating a testbench and simulating the module.

VHDL 13

5.2.2   2-to-1 Multiplexer

Create the file `mux2x1.vhd` with a description of a 2-to-1 multiplexer `mux2x1` with a 1 bit wide control input S of the type `std_logic` which switches one of the two 16 bit wide inputs $D0$ and $D1$ (`std_logic_vector`) to the 16 bit wide output O. Use the `case` command to realize the multiplexer. Keep in mind that when using `case` all possible signal values must be covered by the `case` statement. When S carries an undefined signal, the single bits of the output are to be set to `X` (it is helpful to use `when others` for this). When running the simulation, pay special attention to the behavior of the multiplexer at changing input signals with a constant control signal S.

VHDL 14

5.2.3   Key Generator

For the complete encryption algorithm we also need a key generator to perform the 8 round calculations and the output transformation. Since only one extended round module is used to calculate all eight rounds and the output transformation, it is sensible to implement the key generator to generate only the keys needed for the current operation. The generated keys depend on the 128 bit wide input key and the operation in question. The key generator is controlled by a round counter. Depending on the value of the 4 bit wide output of the round counter the corresponding bits in the 128 bit wide input key are selected to create the 6 partial keys of 16 bit each. For a value of "0000" the keys for the first round are generated, for "1000" the keys for the output transformation. The exact rules for the generation of keys have already been introduced on page 10.

Implement the key generator for a 128-bit input for the key, a 4-bit input for the selection of generation rules (as described in the paragraph above, "0000" to "0111" for round 1 to 8, "1000" for the output transformation) and six outputs for the 16 bit wide partial keys.

VHDL 15

The 128-bit input key is predefined with the hexadecimal value 0001 0002 0003 0004 0005 0006 0007 0008. Use the simulation and the values given by the reference model to check all generated partial keys for all eight rounds and the output transformation.

5.2.4 Control Module
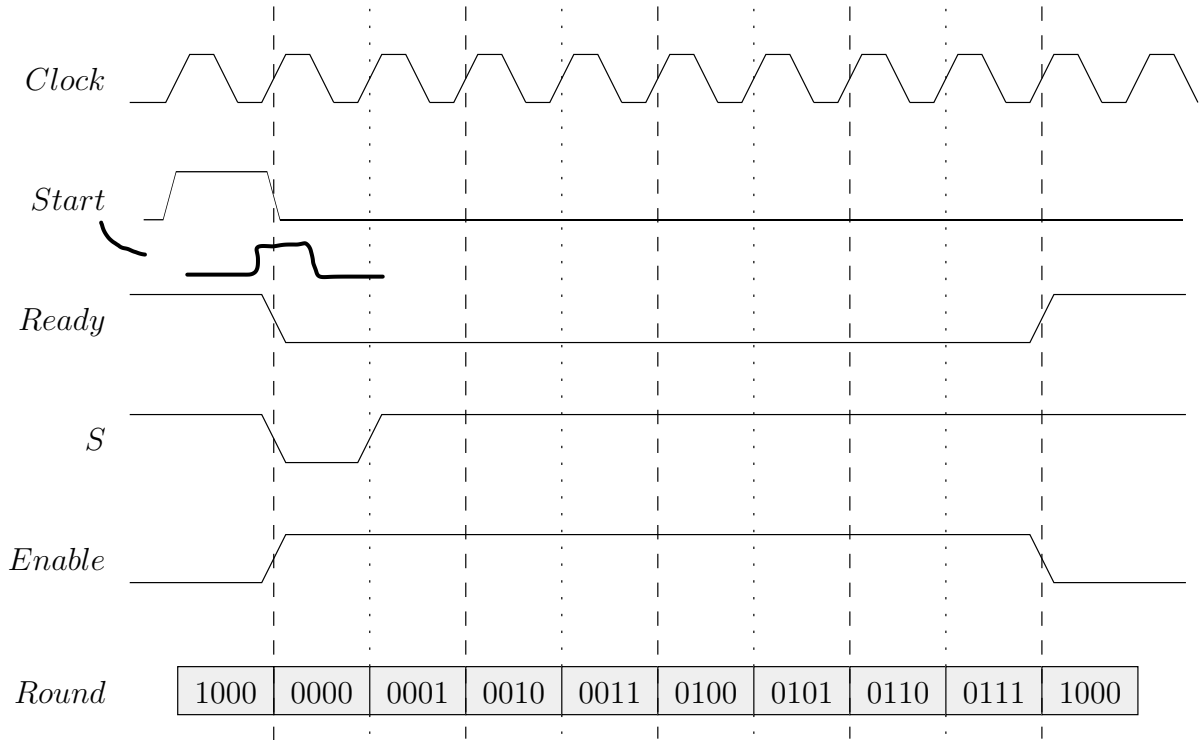
Figure 28: Behavior of the control module

For controlling the data path a control module must be designed. The behavior of the control module can be described as following: When the signal `Start` has a value `1` and the clock signal changes from `0` to `1`, the encryption should be started and the signal `Ready` is set to `0`. When the encryption has finished, the signal `Ready` should be set to 1 again. The signal `Enable` control the register and `S` controls the multiplexer. Implement the control module by following the points below:

- Adapt the `architecture` of the given VHDL file `control.vhd` to describe the behavior of the control module by using one `process`. The signal `Round` is used to generate the subkeys. Furthermore the signals `EN` and `S` are used to control the register and multiplexer respectively. Fig. 28 describes the behavior of the control module. **This figure must be strictly followed in the simulation and implementation. Any deviation will lead to a significant deduction of the grade.**

- Initialize all the signals and variables.

- Create a testbench `tb_control` to check the control module.

5.2.5 The New IDEA Algorithm

Modify the `architecture` of the given VHDL module `idea_single.vhd` in the following way:

- Add the round module, output transformation, control module, key generator, multiplexer and the register as `component`.

- Generate the correct subkeys from the signal `Round` for every round and the output transformation.

- Initialize all the signals and the variables.

After completing the architecture create a testbench `tb_idea_single` with appropriate stimulis. At the end of the computation the result must be the same as the direct algorithm in `idea.vhd`.

5.2.6 Communication With Outside World - UART

The VHDL module `idea_single` could be implemented now for FPGA. In order to test the algorithm running in the real hardware, an interface module is needed. Its task is to accept the data from a PC, start the encryption and then send the encrypted data back to PC. For this purpose the EIA-232-Standard[5] is used. On the FPGA it is then necessary to implement a so-called UART[6] which performs the communication. The necessary files are already added to your project and could also be found in `/usr/local/labs/HDL/share/template`. The template consists of the following VHDL files:

- `clk_div.vhd` serves for generating the clock signals for the UART.

- `rxcver.vhd` is the receiver side of the UART.

- `txmit.vhd` is the sender side of the UART.

- `uart.vhd` describes the UART as a total and contains the modules described in `rxcver.vhd` and `txmit.vhd`.

---

[5]See http://en.wikipedia.org/wiki/RS-232

[6]Universal Asynchronous Receiver and Transmitter.
See http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter

- `idea_com_inner.vhd` connects the UART and the IDEA algorithmus together. This file should be adapted to your own definition of the IDEA algorithm.

- In `idea_com.vhd` the clock generator and the module `idea_com_inner.vhd` are included. This file must be set as the „top module" (in view „Implementation"). Click the module `idea_com.vhd` with the right mouse button and select the item „Set as top module"in the pop-up menu.

In order to set how the inputs and outputs of the top level entity are connected with the pins of the FPGA, a `User Constraint File` (with file suffix `ucf`) is needed. In this lab such a file already exists in the project. (Additionally you find it in the template folder.) The connection between an input or output of the design and a pin of the FPGA can be done with the „Pin Assignment" command `LOC`.

```
NET "<port_name>" LOC=<pin_location>;
```

The name of the input or output of the design is labeled as `port_name`. `pin_location` is for the name of the pin of FPGA. The board used in this laboratory contains a serial interface. The FPGA pin `R7` has already been connected with `RxD` and `M14` with `TxD` of the serial interface[7]. For more information please check the online help under the key word `UCF` as well as the documents for the board used in this laboratory.

$\boxed{\text{VHDL 16}}$
- Check all the „Pin Assignment" commands in the UCF template file `idea_hw.ucf` and use it. Tip: On the board there is a clock generator integrated. This generates a signal with a frequency of 50 MHz which is connected with the FPGA pin `C9`.

- Double click `Generate Programming File` so that a bitstream file is generated for the programming of FPGA. Furthermore the program Xilinx® $iMPACT^{TM}$ is started and the bitstream can be downloaded.

- Program the FPGA. Connect the board with your PC using the USB cable. *T*ip: Because the board consumes a certain amount of power during the registration as an USB device, it is not possible to connect the board with a so-called „Bus-powered" hub. Plug the board directly on your PC and wait, until the firmware for programming the FPGA has been loaded. A green LED near to the USB jack on the board will light up after the firmware is loaded.

---

[7]The names of the corresponding FPGA pins are printed near to components on the board. The LED `LD0` is for example connected with the FPGA pin `F12`.
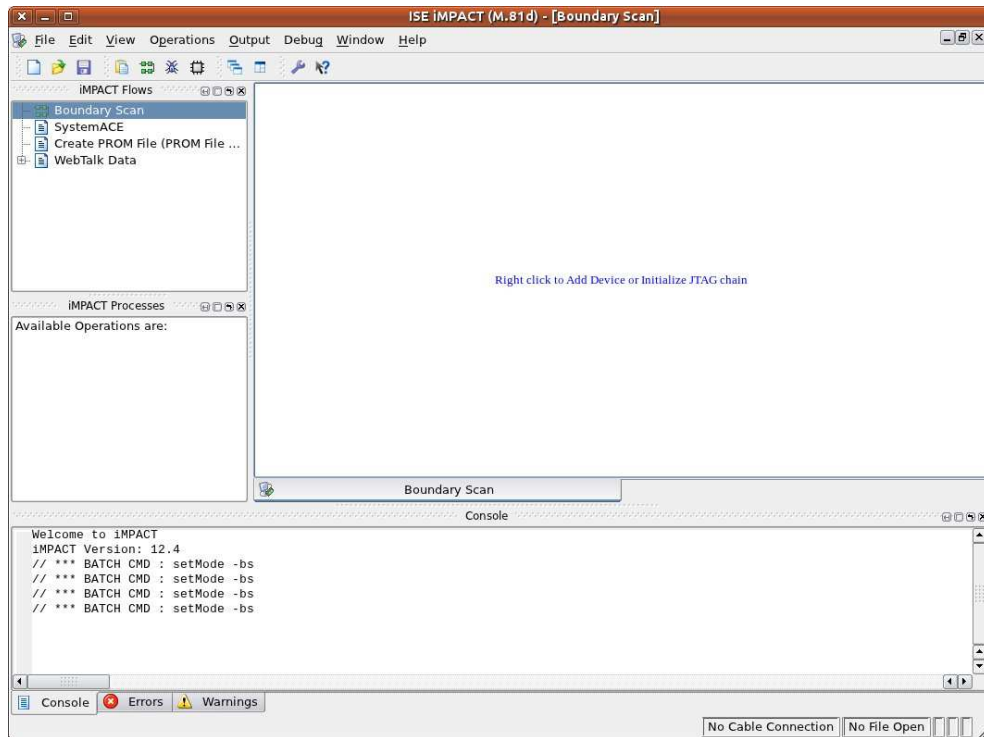
Figure 29: Options by opening Xilinx® *iMPACT*^{TM}
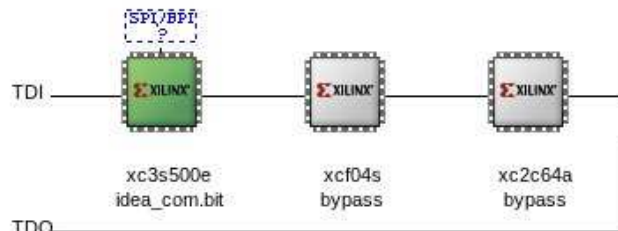
Figure 30: The identified chips in the JTAG chain

- Select the item „Manage Configuration Project (iMPACT)" from the processes panel (See Fig. 26). The program Xilinx® *iMPACT*^{TM} will be started. Select then „Boundary-Scan (JTAG)" and then „Initilize Chain" as shown in Fig. 29. The program will search for the connected FPGA automatically. If FPGA is found, there will be three available chips on the board displayed, which could be programmed by Xilinx® *iMPACT*^{TM} (see Fig. 30). The first chip is the FPGA used in this laboratory and should be directly programmed from PC. Because the

Figure 31: IdeaTester Start

FPGA contains no non-volatile memory, the design will get lost after a reset. The second chip in the JTAG chain is a platform flash memory which is non-volatile, but will not be used in this laboratory. (In this chip there exists a test program for checking the board. This test program will be loaded into the FPGA automatically every time when the board is powered up.) The third chip is not used.

- You will be required to give the file that should be loaded into the corresponding chips. For the first chip please give the `.bit` file which lies in your project directory. For other two chips the programming is not required, so just click „Bypass" for them.

- Click the first chip with the right mouse. A pop-up menu opens. Select „Program" there and then click „OK". The FPGA will then be programmed. At the end a message called „Program Succeeded" should be displayed.

- Test the algorithm with help of the tool `ideatester`. As shown in
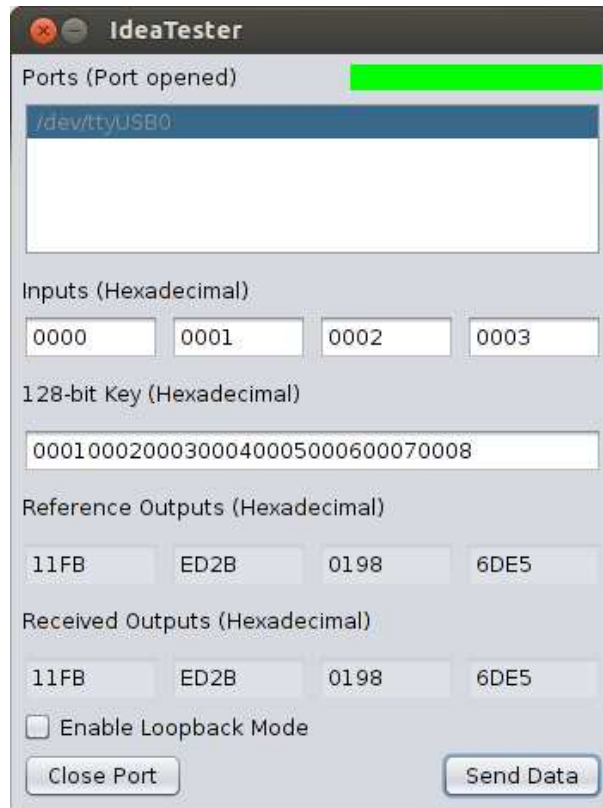
Figure 32: IdeaTester with Correct Result

Fig. 31, open the port first (the one including "USB") and then send the data including the plain text and the key from PC to the board. The result back from the board will be shown and checked with the calculated reference values. If the values are correct, the bar will become green, as shown in Fig. 32. If you enable the loopback mode, the data sent to the board will be returned back without any encryption. The loopback mode is used to test whether there is any problem with the cable connection, as shown in Fig. 33 or if the FPGA programming is done. The loopback mode cannot be used to check if your implementation is correct!

The system structure after binding the UART module is shown in Fig.34. The IdeaTester on PC communicates with the board using the serial-usb cable. The UART module is responsible for receiving the data from PC and feeding the IDEA module. After the encryption is finished, the UART module sends the data back to PC.
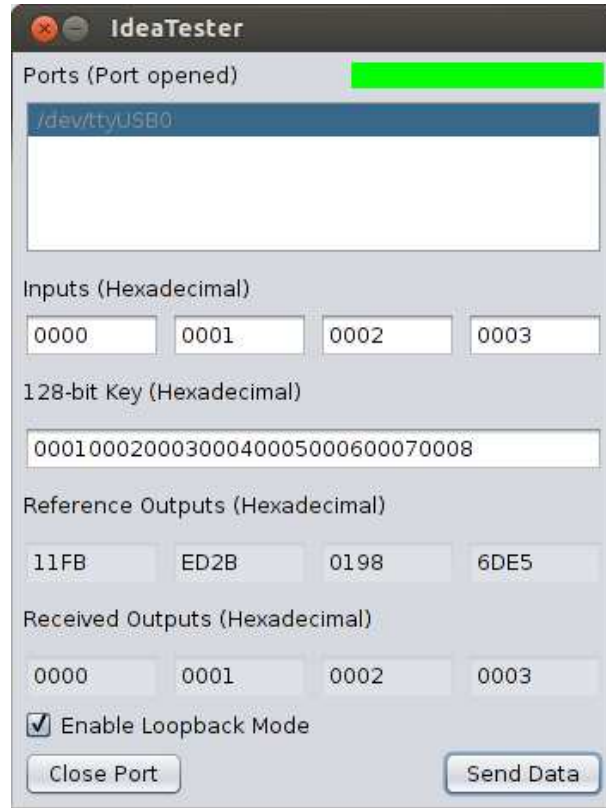
Figure 33: IdeaTester in Loopback Mode

5.3 Resource Constrained Scheduling II

Although the design in the last section could be implemented, it still needs a large amount of resources of the FPGA. Normally, it is required to integrate more complex modules such as processors on the same FPGA. Therefore, an implementation of the IDEA algorithm with less resource consumption is favored. In this section it is assumed that the IDEA algorithm is used as a hardware accelerator for a RISC processor which is also synthesized using the FPGA resources. So the IDEA algorithm should not consume more than 5% (circa 465 LUTs) of the resources of the FPGA.

This chapter is dedicated to creating a round module consisting of a data-path and a controller. The controller realizes a process control, which has access to the calculations on the datapath. All calculations take place on the data path and results are stored here as well. The data pathcan be realized using only one multiplier and one adder for a complete calculation round. Therefore, the round is split up into four steps so that four calculations are
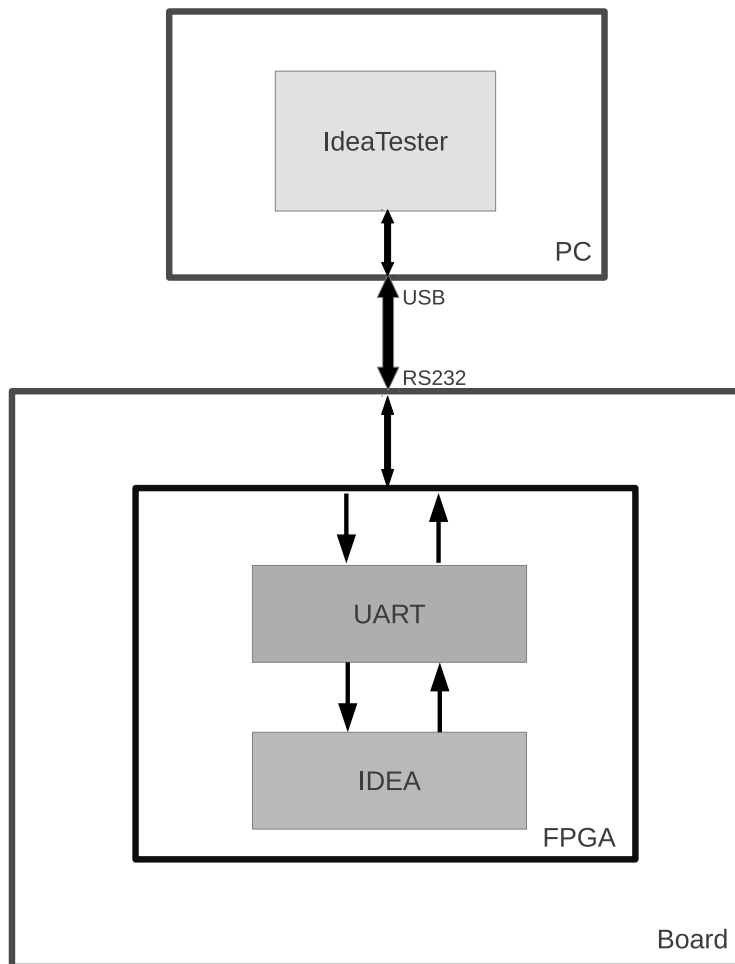
Figure 34: System Structure

performed in sequence. Each step consists of one modulo multiplication and one addition. In each round there are more XOR modules than adders or modulo multipliers and these modules need only few CLBs. Therefore it is legitimate to use more than one XOR module for each step. Splitting up the round into four steps is done by introducing 3 separating lines into Fig. 5 on page 11. Splitting up a circuit into several steps using limited resources is called Resource Constrained Scheduling. The repeated use of the same module for different calculations is called Resource Sharing.

In order to have exactly one modulo multiplier and one adder in every step, the modules are rearranged over the separating lines until there is one of each in every step. One has to keep in mind that the direction of the data flow always goes from top to bottom. The results of the modulo multiplier
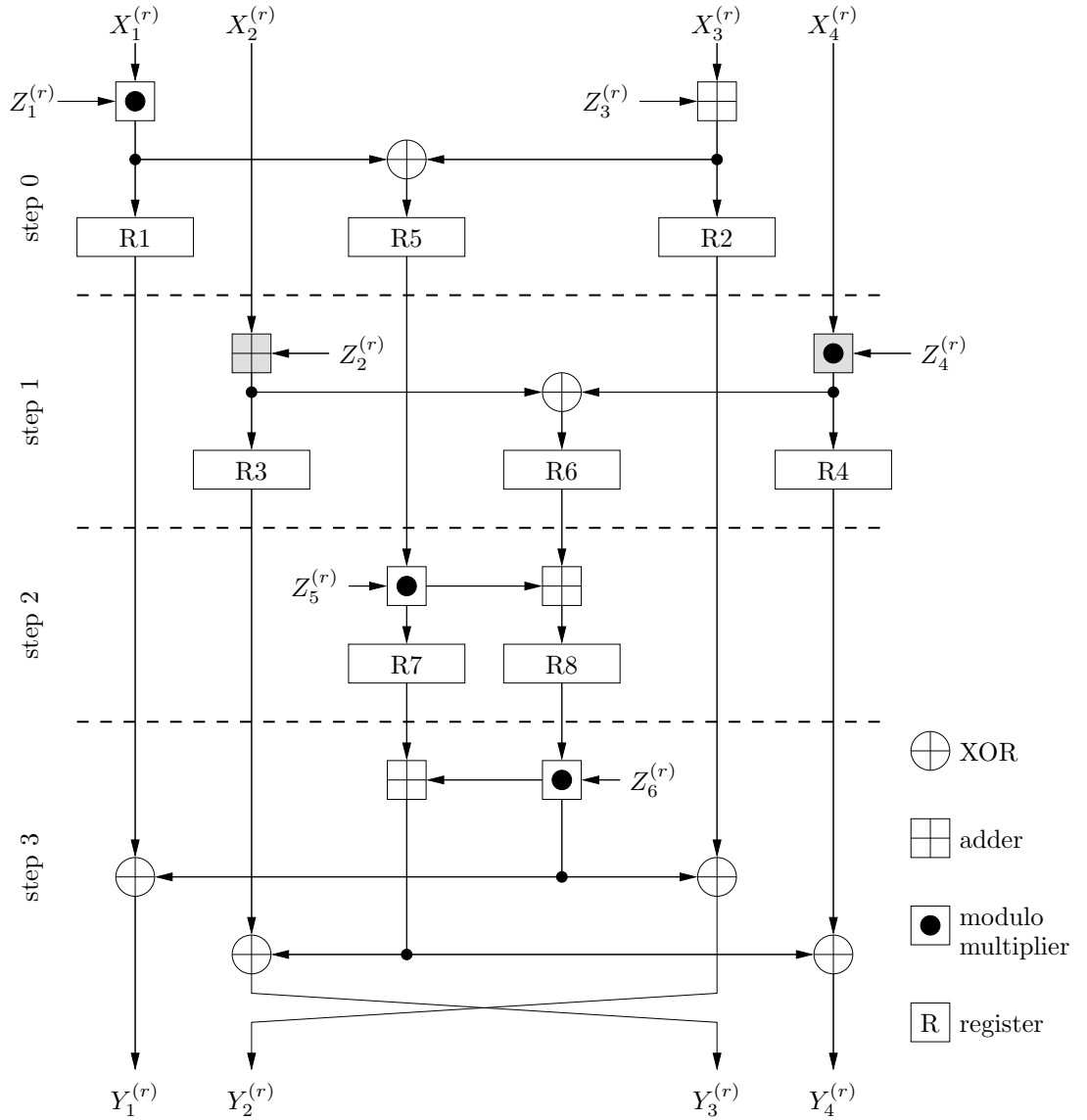
Figure 35: Slitting a round into 4 partial steps

and the adder are kept in registers[8] for later use. Figure 35 shows a possible division of the round into four steps. The elements marked in gray (an adder and a multiplier) have been moved into step 1 (compare with Fig. 5).

After splitting the round into 4 partial steps, we will now develop a multiplexer-based design for the datapath. The design should be able to calculate a round using one modulo multiplier, one adder and several XOR modules. The con-

---

[8]The registers are edge-triggered with Load/Enable. For a better overview, the CLK and EN signals are not shown in Fig. 35 and 36.
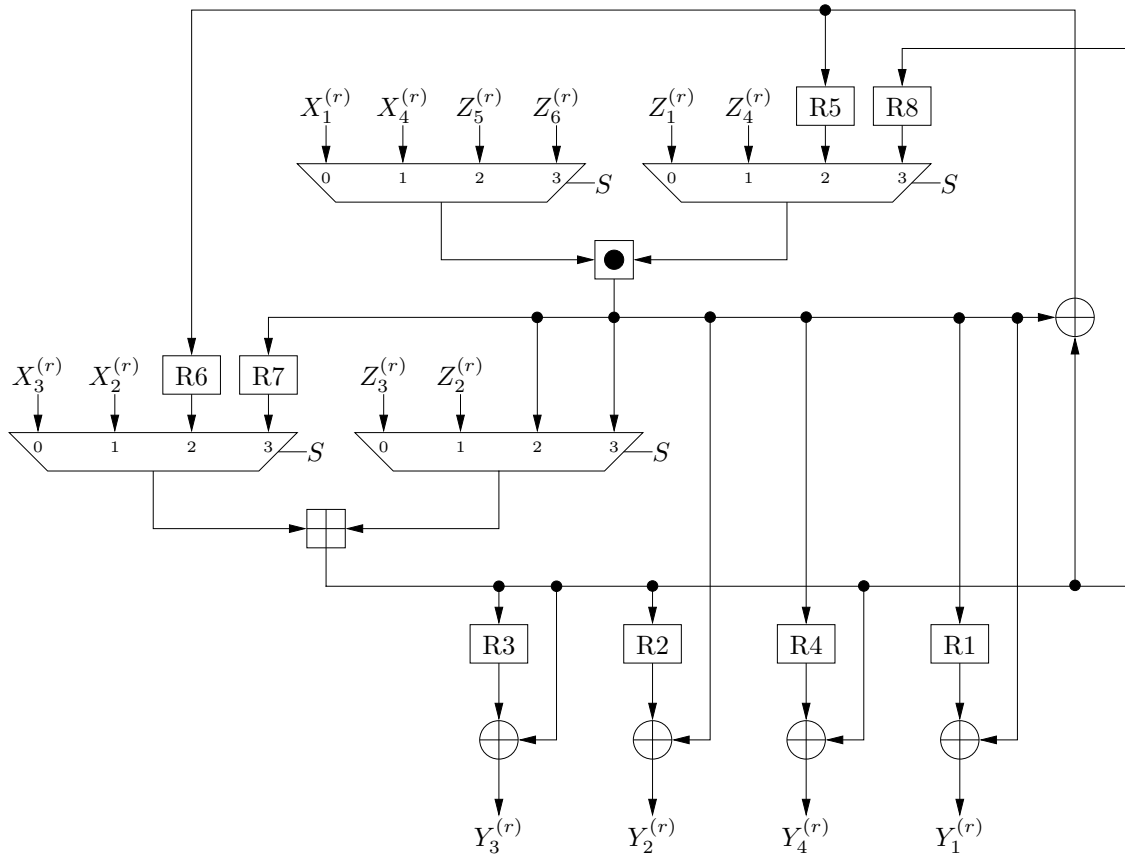
Figure 36: Datapath of the round module

trol signal S ensures that the correct input for the partial step is switched to the output of the multiplexer, thereby used in the calculations for the partial step.

Preparations:

- How would you split up the round if you had 2 modulo-multipliers, 2 adders and unlimited XOR modules? Draw a diagram of your design (see Fig. 35).

- How many partial steps would be necessary for this design?

- Where would you need to insert registers?

- Design a datapath with 2 adder and 2 modulo-multiplier. The design should have two steps. (see Fig. 36).

For the implementation of this section the project (`idea_rcs2`) in the folder submitrcs2 with the same settings than the projects before is given. In order to implement the clocked round module, a datapath and its controller is needed. The implementation of the datapath requires a new module: a 4 to 1 multiplexer.

5.3.1 4-to-1 Multiplexer

VHDL 17 Create a description of a 4-to-1 multiplexer with a 2 bit wide control input S of the type `std_logic_vector` which switches one of the four 16 bit wide inputs (`std_logic_vector`) to the 16 bit wide output. When running the simulation, pay special attention to the behavior of the multiplexer at changing input signals with a constant control signal S.

5.3.2 The Round Controller

A controller is necessary for the coordination of the multiplexers and the registers. After the initial signal *init* is set the controller switches the multiplexers to the next input for every two clock cycles. The *enable* signals are correspondingly set for the registers. This allows them to store the result of the calculation at the next rising edge. A *result* signal is used to indicate that the round is finished. This procedure is illustrated by the time diagram shown in Fig. 37.

It is helpful to split every partial step into two cycles for the purpose of implementation. At the beginning of the first cycle the control signals of the multiplexers are set. This connects the data input to the calculation modules. After the calculation the results are stored into the register at the beginning of the second cycle. For the realization of the 4 partial steps (with two cycles each) an internal state is introduced that describes both the actual partial step and the actual cycle of the partial step (fig. 37, internal state).

VHDL 18 Use the `architecture` of the given VHDL file `control.vhd` to realize each of the following points.

- Create a 3 bit counter for the controller that stores the internal state. The counter is incremented by 1 at every rising edge of the *clock* signal until it reaches the value '111'. It is reset to the value '000' when the actual state is '111', the *init* signal is set, and a rising edge occurs. Initialize the 3 bit counter with "111".

Clock

Init

Enable
R1 R2 R5

Enable
R3 R4 R6

Enable
R7 R8

Result

Multiplexer
control S

| 00 | 01 | 10 | 11 |

Internal
State

| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
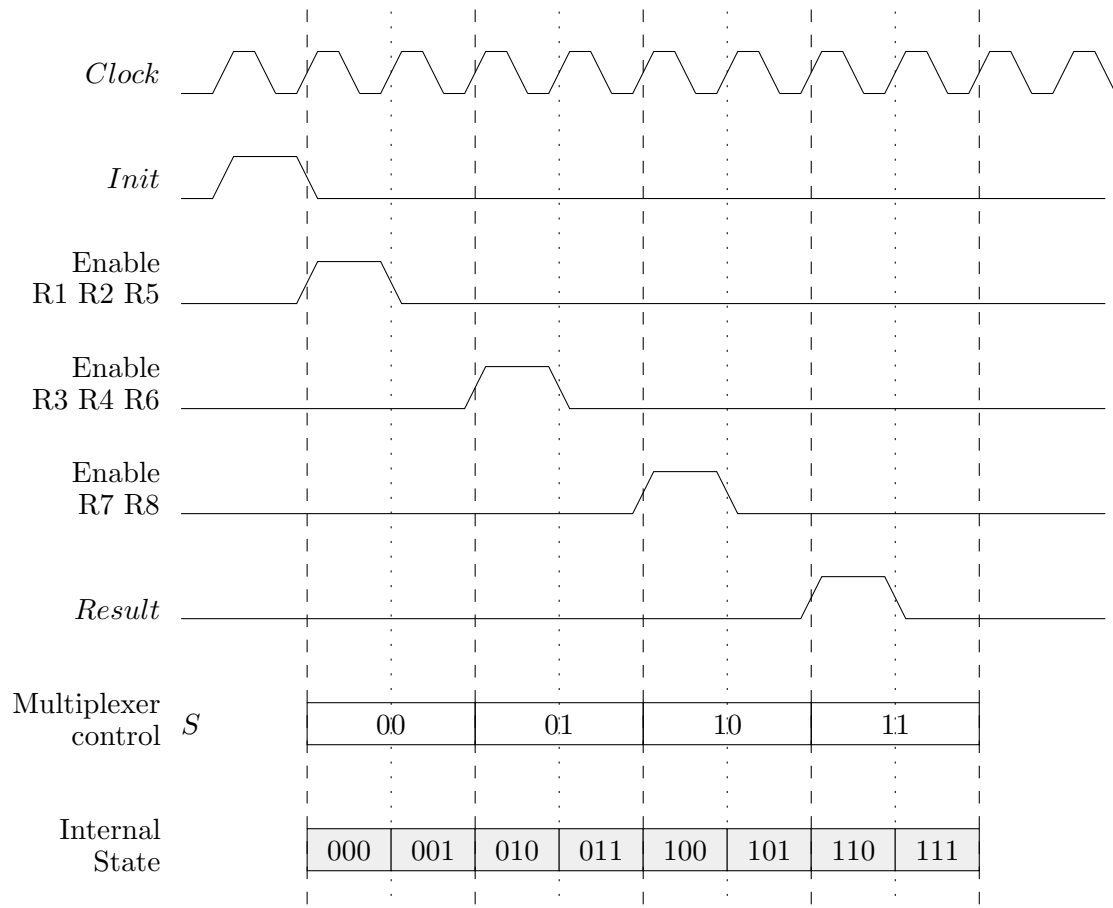
Figure 37: Round Controller

- Set the three *enable* signals for the registers and the *result* signal depending on the internal state.

- Use the internal state to create the S signal for the multiplexer control as simply as possible.

- Set the port ST to "00" for the first step. It will be implemented in section 5.3.5 for the extended round module.

Create a testbench and test the controller for one round. [VHDL 19]

5.3.3 Datapath of The Round Module

Create a file `datapath.vhd` including the description of the datapath for [VHDL 20]

the round module according to Fig. 36. Include the necessary modules as `components`. Consider that the datapath module needs inputs for the control of the registers and the multiplexers provided by the controller. Furthermore a *clock* signal is required. The datapath does not need its own testbench.

5.3.4 The Clocked Round Module

$\boxed{\text{VHDL 21}}$ Modify the `architecture` of the given VHDL file `clockedround.vhd`. Include the controller and the datapath modules as `components`, instantiate and connect them. Create a testbench and test the clocked round module.

5.3.5 The Extended Round Module

By resource sharing the number of CLBs per round could be reduced drastically. The current implementation uses about 465 LUTs (see table 2).

Module	LUTs per module	LUTs total
5 * XOR	16	80
1 * Adder	16	16
1 * Modulo-Multiplier	106	106
8 * 16 bit Register	16	128
4 * 4 to 1 16 bit Multiplexer	32	128
1 * controller	7	7
$\sum$		465

Table 2: Resources of the clocked round

A comparison between the output transformation module (Fig. 6, page 11) and the round module shows that the first two steps of each round are very similar to the output transformation. In the output transformation, first $X_1^{(9)}$ and $Z_1^{(9)}$ are multiplied and $X_3^{(9)}$ is added to $Z_2^{(9)}$. Then $X_4^{(9)}$ is multiplied with $Z_4^{(9)}$ and $X_2^{(9)}$ added to $Z_3^{(9)}$. A comparison to the first two steps of a round calculation (like it is realized in the design shown in Fig. 36) reveals that both calculations are identical when the partial keys $Z_2^{(9)}$ and $Z_3^{(9)}$ are interchanged. The result of the output transformation can be found in the registers R1 to R4 after the second step.

The datapath and the controller of the round are now to be modified so that the new input signal *trafo* can switch the clocked round module to perform the output transformation. To perform the transformation the fourth
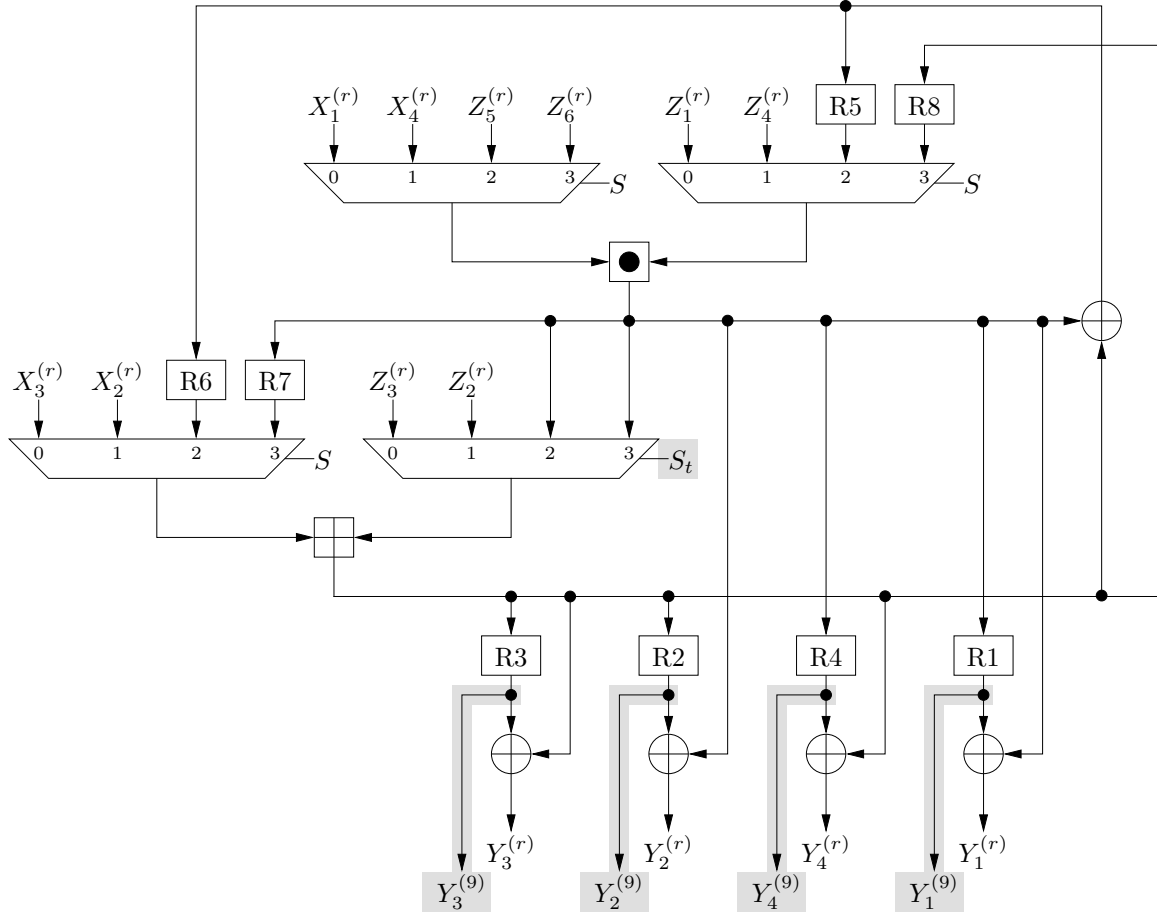
Figure 38: Datapath for the round calculation and output transformation

multiplexer switches to input `1` ($Z_2^{(9)}$) in the first phase and `0` ($Z_3^{(9)}$) in the second. The controller converts the signal S to the signal S_t for the fourth multiplexer by inverting the least significant bit of S when the input $trafo$ is set to `'1'`. Keep in mind that the internal states of `'100'` and `'101'` won't appear in the controller once $trafo$ is set to `'1'`. To read the result of the output transformation in the registers R1 to R4, four more output signals are necessary in the datapath module. The changes to the datapath module from Fig. 36 are shown in gray in Fig. 38.

Add the $trafo$ input to the controller for the extended round module. With the $trafo$ signal set the controller is to generate the multiplexer and register signals as shown in Fig. 39. If $trafo$ is not set, the controller is to generate the signals as before (fig. 37). In this case the S_t signal is identical to the S signal. Use a testbench to check the function of the controller in both, the

VHDL 22

51

Clock

Init

Enable
R1 R2 R5

Enable
R3 R4 R6

Enable
R7 R8

Result

Multiplexer
control S

| 00 | 01 | 11 |

Multiplexer
control S_T

| 01 | 00 | 10 |

Internal
state

| 000 | 001 | 010 | 011 | 110 | 111 |
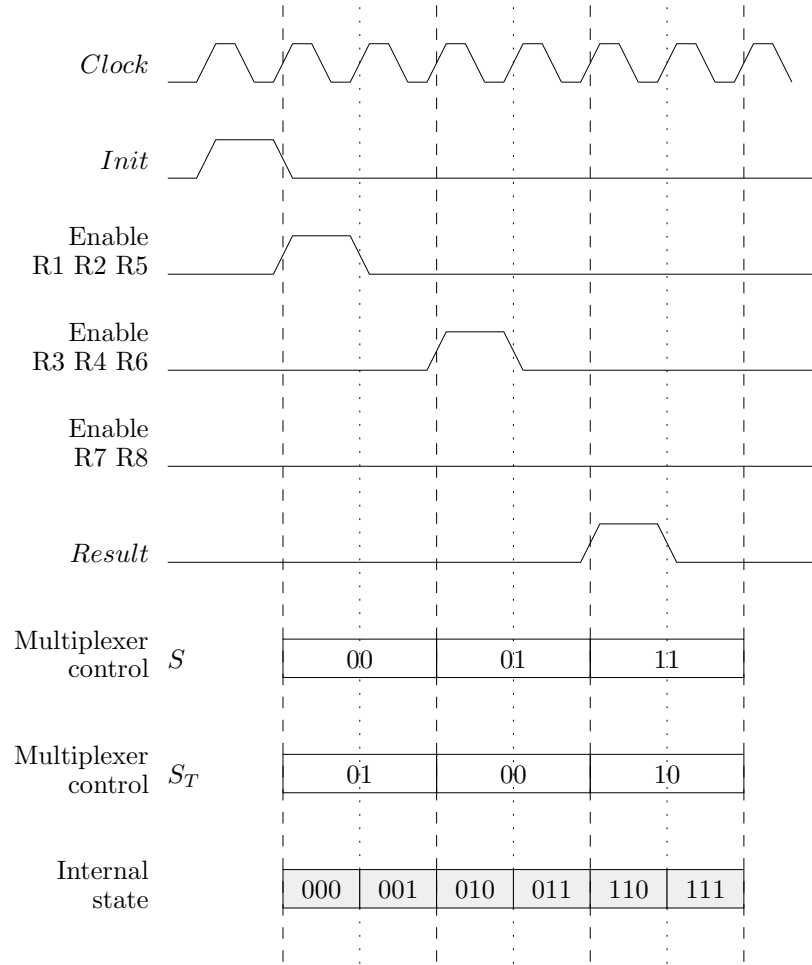
Figure 39: Controller for the extended round, Trafo $= 1$

round calculation mode and the output transformation mode.

VHDL 23   Modify the datapath to include the four additional outputs for reading the registers R1 to R4. Use the S_t signal to control the fourth multiplexer.

VHDL 24   Test the extended clocked round module in a testbench both in round calculation mode and in output transformation mode. To get some reference values for $Y_i^{(r)}$ with different combinations of $X_i^{(r)}$ and $Z_i^{(r)}$ values, you may use the reference models.

5.3.6   Implementation of the Round Counter

To complete the algorithm we need to design a controller which controls all eight calculation rounds, the output transformation and the key generator.

It should be realized as a round counter. The original input values are put into the round module in the first step. The results of previous rounds are then stored in registers and used as input for the following rounds. The whole IDEA algorithm is to be assembled as shown in Fig. 40. The round counter is also used to generate the control signals for the 4 input multiplexer and the extended round module.
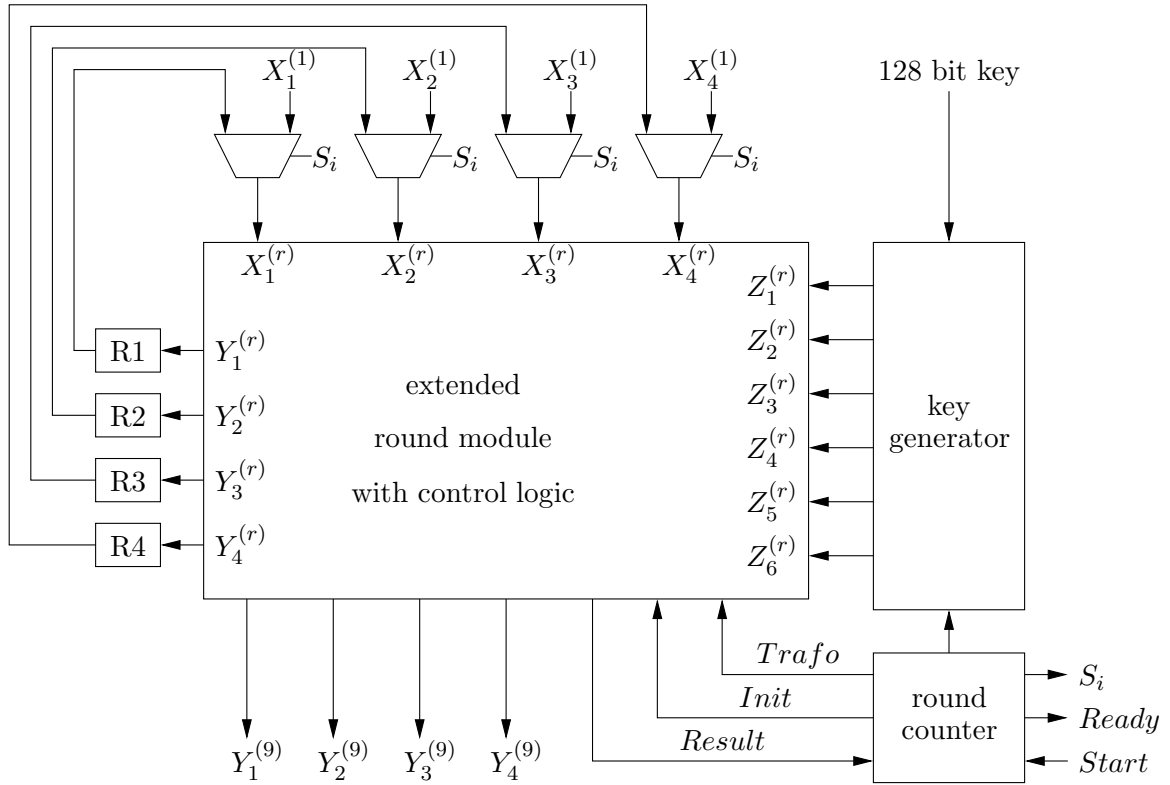
Figure 40: Realization of the IDEA algorithm

In addition to the round count, the round counter provides the *init* and *trafo* signals which control the extended round module. Furthermore the signal *ready* is used to show when the whole encryption process is finished and the result can be read from $Y_1^{(9)}$ to $Y_4^{(9)}$. The *result* signal of the round module signals the round counter that the current calculation is finished and the module is ready for the next. The encryption of the initial input $X_1^{(1)}$ to $X_4^{(1)}$ begins with the setting of the *start* signal.

The round counter is specified as a finite state machine with 3 states: `Sleep`, `Setup` and `Calc` (Fig. 41). In the standby mode `Sleep` it waits for the *Start*
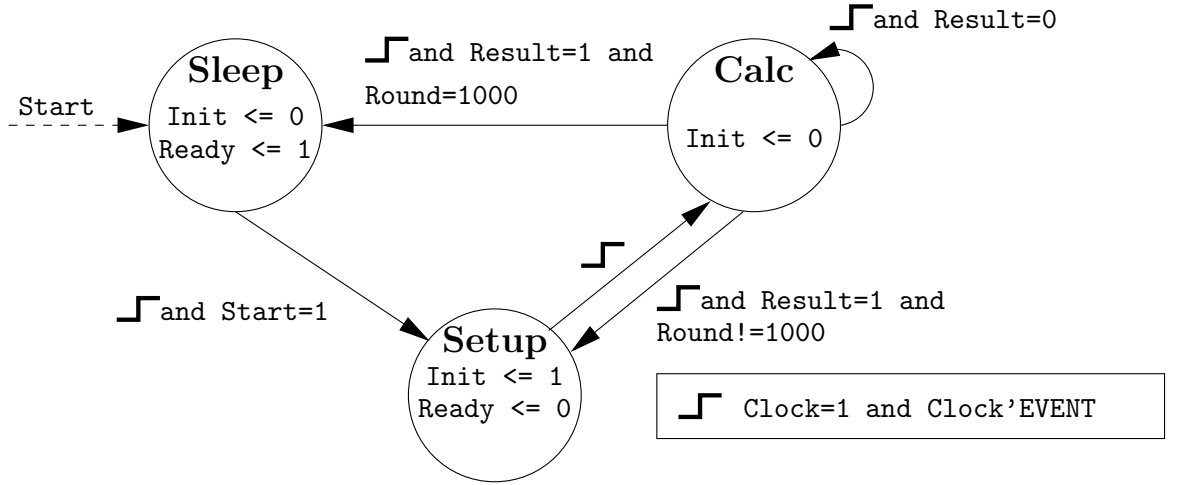
Figure 41: Specification of the round counter as a finite state machine

signal and changes to `Setup` at the next rising edge of the clock signal. Here the *Init* signal is set to '1', triggering the calculation in the round module. At the next rising edge it changes to the state `Calc`. If the *Result* signal is '0', it will stay at this state. If the *Result* signal is '1', the next rising edge changes the state to `Sleep` if the round counter has the value of '1000'. Otherwise the finite state machine changes to `Setup`. In the `Sleep` state the round counter is always '1000'.

The internal counter can also be understood as a finite state machine (Fig. 42). Here the 4 bit wide counter is incremented by 1 by a rising edge if the *Result* is '1'. If the round counter has the value "1000" it is set back to "0000". This counter corresponds to the signal *Round*.

Figure 42: The internal counter also as a finite state machine

The signals $Trafo$ for the extended round module and S_i for controlling the

multiplexer are to be derived from the current value of the round counter. The most significant bit of the round counter can be used directly as the $Tr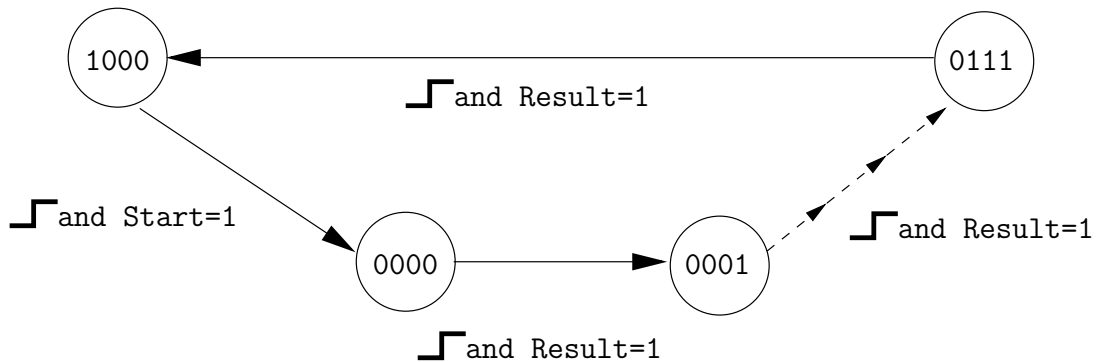afo$ signal. The signal S_i for controlling the input multiplexer must be set when the round counter has the value '0000'. The $Ready$ signal is set only when the encryption has been completed, i.e. when the round counter is in the Sleep state. This is the case when the result of the last calculation of the round module (output transformation) can be read at the outputs $Y_1^{(9)}$ to $Y_4^{(9)}$.

Implement the behavior of the round counter in the architecture of the given file roundcounter.vhd as described and test its functioning in a testbench. ⬚VHDL 25

5.3.7 Hardware Oriented Implementation of the Algorithm

Implement the complete IDEA algorithm as shown in Fig. 40. Use the four already available 2-to-1 multiplexers as the input multiplexers. Keep in mind that only 2 inputs are needed and the 1 bit S_i signal can select one of them. ⬚VHDL 26

The results are stored by the registers R1 to R4 when the round module signals that it performed the last calculation and that the results can be read at the outputs $Y_1^{(r)}$ to $Y_4^{(r)}$. Connecting the *enable* signals of the registers to the *result* signal of the round controller provides a simple solution. The *clock* input is connected to the global *clock* signal.

Implement all needed modules in the architecture of the given VHDL file idea_rcs2.vhd and create a testbench in order to test the correct functioning of the implementation. ⬚VHDL 27

5.3.8 UART - Resource Constrained Scheduling II

Less resources should be used with the new IDEA algorithm comparing with the one before. In order to test the new algorithm, an interface must be used to receive the data from a PC, to start the encryption and to send the encrypted data back to PC. Use the UART module as described in section 5.2.6.

Program the FPGA now and test the implementation as described in section 5.2.6. ⬚VHDL 28

5.4 Resource Constrained Scheduling II+

Now we consider the question whether we can run the clocked round module at a higher throughput. From Fig. 37 and Fig. 39, we see that for each step

two clock cycles are used. In the first clock cycle, the multiplexers select the corresponding input data and the calculated results are propagated to the inputs of the registers. Then at the beginning of the second clock cycle, the data are stored by registers and are already prepared for further calculations. However, since the select signal of the multiplexers is scheduled to change after this second clock cycle, the data will just stay at the registers without being processed. Actually the second clock cycle in each step is not needed.

- Use the project with the name `idea_rcs2plus` in the folder submitrcs2plus and add all the files of the RCS2 implementation by `Add Copy of Source`.

- Rename the file `idea_rcs2.vhd` to `idea_rcs2plus.vhd` and the module from `idea_rcs2` to `idea_rcs2plus`. Make the needed adaptions.

- Adapt the control logic in Fig. 37 and Fig. 39 and modify the `architecture` of the file `control.vhd` to reflect the thoughts above.

- Simulate your design and test your design on hardware again.

6 Delay Estimation

In order to evaluate the overall performance of the implemented algorithm, we need to be able to estimate the propagation delays in the single modules. This requires a closer look at the target technology.
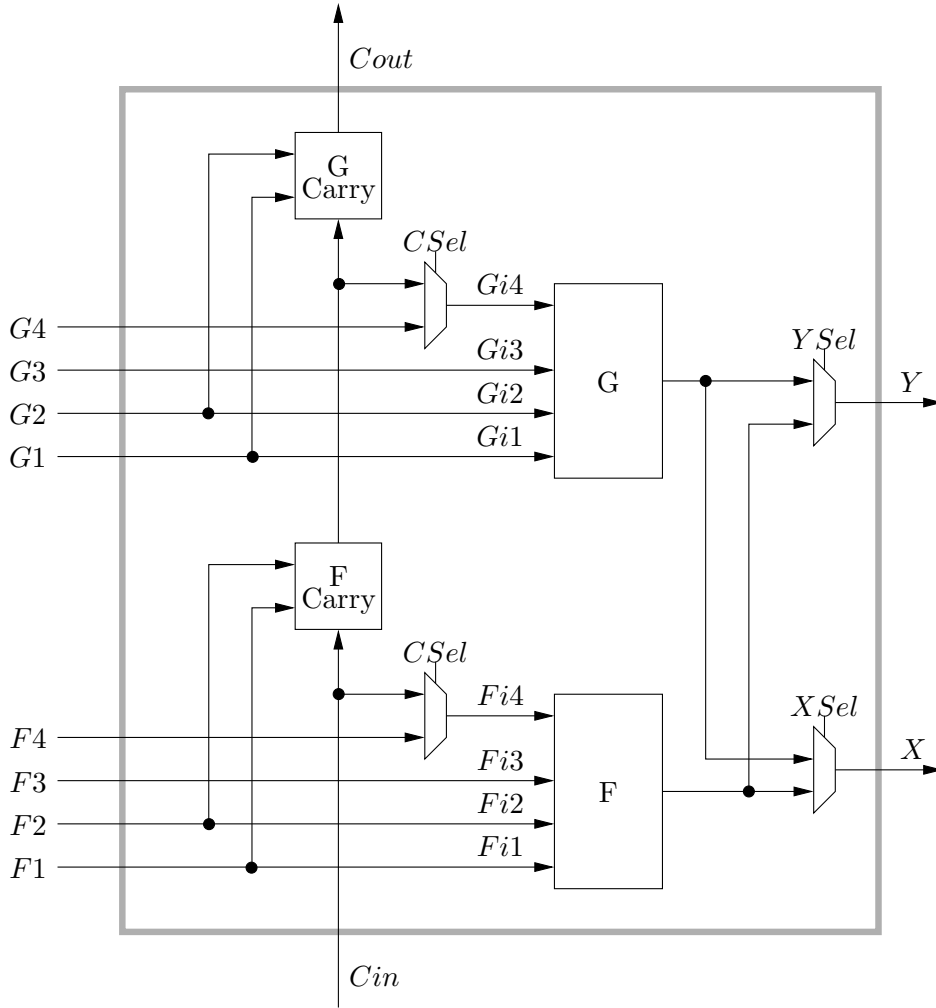

Figure 43: Simplified structure of a SLICE

6.1 Target Architecture

We already have a description suitable for the realization of the IDEA algorithm on a Xilinx FPGA of the type Spartan 3E XC3S500E. This FPGA

provides 9312 Look Up Tables (LUTs) which can be accessed over 232 in-put/output pins. Four SLICEs form a CLB, where each SLICE has two function generators, `F` and `G`, as well as a special carry logic. The function generators are realized as LUTs and can be programmed to perform any desired logic function with four inputs F1 to F4 (G1 to G4). Furthermore every CLB has eight D-flipflops. Figure 43 shows the simplified structure of a SLICE[9]. The CLBs are connected by freely programmable wiring resources. The wiring resources are available in varying numbers and can have rather different propagation delays. An estimation of the propagation delays in the wiring is difficult in this early stage of design, since the delays depend both on the number and the type of the wiring resources used. Typically the wiring accounts for 20-60% of the the total propagation delay. These delays can only be calculated after the layout of the whole circuit is complete (i.e. after placing all modules and routing wires). Since the "Post-Route Simulation" is a complicated task and led to erroneous results in current versions of Xilinx® ISE *WebPack^{TM}* , we will not consider wire delays in this lab. Furthermore the input and output drivers by that signals are fed into and read out of the FPGA need some time for switching the signals. These delays will be ignored as well.

6.1.1   Delay Estimation: 16 Bit XOR

Every SLICE (Fig. 43) provides two function generators (F and G). These are
`? 4` used to perform one XOR operation each. How many SLICEs are therefore required to realize a 16 bit XOR operation?

The delays given in table 3 can be used to estimate the propagation delays
`? 5` in the single components of the CLB. How long does a 16 bit XOR operation take? Is this time dependent on the input width?

CLB component	propagation delay
F/G function generator	3 ns
F/G-Carry	1 ns
CSel/XSel/YSel-Multiplexer	0 ns

Table 3: Propagation delays CLB

---

[9]The complete specifications can be found as a PDF file under http://www.xilinx.com/support/documentation/user_guides/ug331.pdf

6.1.2 Delay Estimation: 16 Bit Adder

The addition of binary numbers more than one bit long can be realized with
several adder architectures. This laboratory will focus on the ripple-carry
architecture[10]. Generally every bit needs one full adder with the input x_n,
y_n and c_n as well as the output s_n and c_{n+1}. The output s_n is the sum and
c_{n+1} is the carry. They are defined by the following logic functions:

$$
\begin{aligned}
s_n &= x_n \oplus y_n \oplus c_n \\
c_{n+1} &= x_n y_n + x_n c_n + y_n c_n
\end{aligned}
\tag{6}
$$

The full adder can be split into two blocks (fig. 44): The calculation of the
sum (in the function generators F and G) and the calculation of the carry
(carry logic). Figure 44 shows a possible realization. n full adders are required
for an n bit adder. The carry output of a single full adder is then connected
to the carry input of the adder of next higher significance.
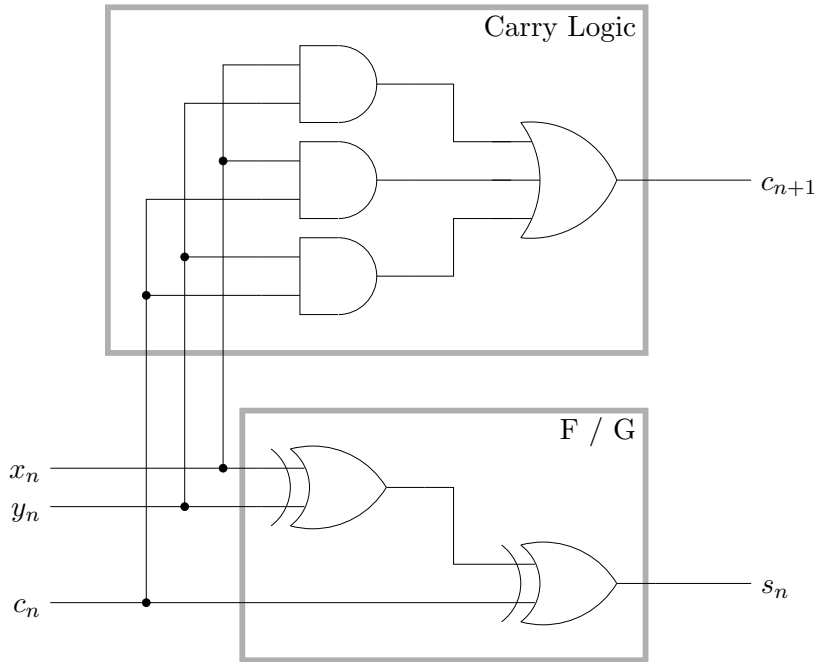

Figure 44: Full adder

A SLICE of the target architecture can realize two full adders. The logic
for calculating the carry is already hardwired into the SLICE and can be
activated by the *csel* signal. With the two function generators the calculation

---

[10]Another widely used architecture is the carry-look-ahead adder. See also [8].

of the sum can be realized. The propagation delay for the carry logic is 1 ns, for the function generator 3 ns. **The delay for the multiplexer is negligible**.

`? 6` Use your knowledge to find the maximum delay of a 16 bit adder. What path causes the longest delay? Consider the fact that at the time $t = 0$, all input signals are valid.

In the next questions, assume that the synthesis process of the FPGA uses hardware adders instead of LUT based ripple carry adders. Assume the delay to be **10 ns** for a 16 bit adder.

6.1.3 Delay Estimation: Modulo-Multiplier

The layout for the reference implementation, which serves as a basis for an estimation of the propagation delay of the module, has been created by fully automatic synthesis. With that the propagation delay of the module can be estimated [11]. That delay time was found to be **22 ns**.
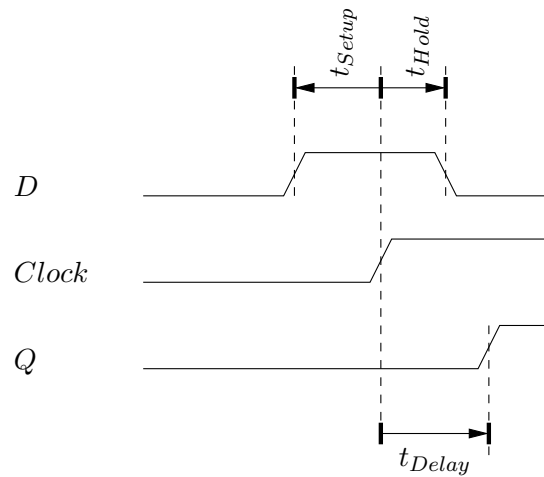
6.1.4 Delay Estimation: Register

Figure 45: Propagation delay register

The propagation delay of registers are characterized by three times: The switching time t_{delay}, the setup time t_{setup} and the hold time t_{hold} (see Fig. 45).

---

[11]The 16 bit multiplication has been realized using a hardware multiplier.

The propagation delay is the time between a rising *clock* edge occurring and the input signal D appearing stable at the output Q. The setup time gives the time for which the input signal D must have been stable before a rising edge. The hold time is the time for which the input must be stable after the edge has occurred. Only if the signal is kept stable for these times a correct output can be expected. Furthermore the *enable* signal has to be set in time before the rising edge.

For the implementation of the 16 bit register on a XC3S500E FPGA the setup time is estimated to be 4 ns. The switching time and the hold time are then negligible. The *enable* signal also needs to be stable 4 ns before the rising edge.

6.1.5   Delay Estimation: Multiplexer

As shown in Fig. 46, a 4 to 1 multiplexer can be also realized with three 2 to 1 multiplexers arranged in two stages. Describe a way to realize this concept using only one SLICE (fig. 43, without using the carry logic). Estimate the propagation delay of a 16 bit wide 4 to 1 multiplexer based on your previous considerations.
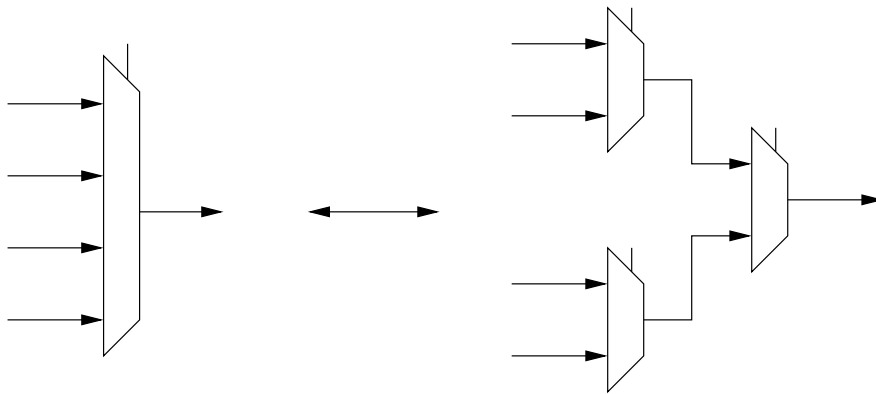
? 7

Figure 46: Two stage 4 to 1 multiplexer

6.2   Delay Estimation of IDEA

6.2.1   Delay of Direct Implementation

Estimate the computing time for the complete encryption process in case the algorithm is implemented directly as described in section 4.6. Use the module

? 8

delays from sections 6.1.1 to 6.1.5 (Remember: The delay time of the adder is just 10 ns). What is the longest path through one round? How can you compute the whole encryption time based on that? How many encryptions per second are possible with this implementation? Would it be possible — if necessary using additional registers — to start a new calculation while the current encryption is not yet finished? Justify your answer.

6.2.2 Delay of Resource Constrained Scheduling I

? 9 Find the shortest clock cycle for which the Resource Constrained Scheduling I works correctly (Remember: The delay of a 2:1 MUX is 0 ns). Considering section 5.2.4, how many clock cycles are needed to finish one complete encryption?

6.2.3 Delay of Resource Constrained Scheduling II

The performance of the realization of RCS II (Fig. 40) depends heavily on the calculation time of the extended round module. For an easier estimation of the total performance the short propagation delays of the round controller and the round counter are neglected. The delays in the wiring are also left out, since an estimation without a layout would be quite imprecise.

? 10 **Clocked Round**   Find the shortest clock cycle for which the clocked round module works correctly. Consider that depending on the control signal there are four different paths through each multiplexer. Furthermore, a register is only the end of a combinational path if it also accepts the result of the current calculation. Show how the longest possible path is created. How are the multiplexers switched in this case ? Which register is at the end of this path?

? 11 **Complete Encryption**   Find the shortest clock cycle for which RCS II (extended round module, register R1 to R4, input multiplexer S_i, round counter and key generator) still works correctly. Please note there is a different delay of a 4:1 MUX and a 2:1 MUX! Is the minimal clock cycle longer than the shortest cycle of the clocked round module if it is operated without external elements? If so, what causes the longer cycles? Trace and describe the critical path.

? 12 Find the number of clock cycles required by RCS II for a complete encryption. **Assume that the external start signal was active for the rising edge**
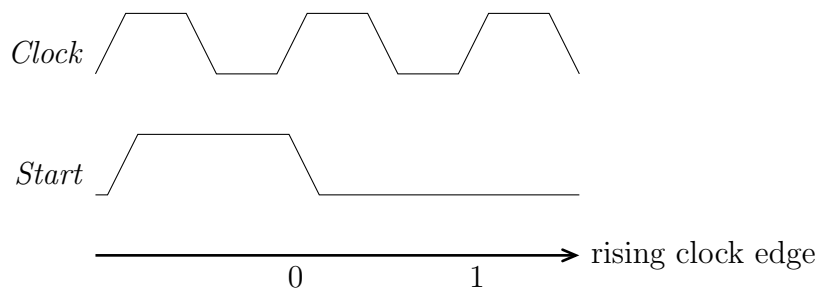
Figure 47: Start signal and Clock

of the *clock* **signal with the number 0**, starting an encryption (see figure 47). At which rising *clock* edge are the signals *init* and *result* set? At which rising *clock* edge does *ready* indicate the completion of the encryption? Please note the sections 5.3.2, 5.3.5 and 5.3.6!

How long does a complete encryption in RCS II of Section 5.3 take? How many encryptions per second are possible? | ? 13 |

RCS II+ How many clock cyles are needed for a complete encryption in RCS II+? How many encryptions per second are possible? | ? 14 |

6.3 Comparison of the Different Implementations

In the following the mentioned implementations are to be compared. Central points are the possible encryptions per second and the required hardware resources. Assume for the registers that they need extra SLICEs even if they are additionally included on the SLICEs. Another factor is the efficiency. We define it as the quotient of the number of encryptions per second and the number of required SLICEs. The area and delay in the **key generator**, the **controller** and the **round counter** are to be **neglected**.

Use your previous results or make estimates where necessary to complete the following table. Only giving numbers is not sufficient. You have to explain how you obtain the estimated numbers. Please note that in the direct implementation not all the Modulo Multiplier are mapped on a hardware Multiplier. | ? 15 |

Implementation	①	②	③	④	⑤
Encryptions / Second					
Required area in SLICEs					
Required input / output PINs					
Efficiency					

① Direct implementation (Section 6.2.1)

② Direct implementation with simultaneous calculation of several encryptions (Section 6.2.1)

③ Hardware oriented implementation - Resource Constrained Scheduling I (Section 5.2)

④ Hardware oriented implementation - Resource Constrained Scheduling II (Section 5.3.7)

⑤ Hardware oriented implementation - Resource Constrained Scheduling II+ (Section 5.4)

References

[1] ANSI, IEEE Standards Board, IEEE Standard VHDL Language Reference Manual: IEEE Std 1076-1993 , New York, 1988, ISBN 1559373768

[2] Peter J. Ashenden, Designer's Guide to Vhdl, Morgan Kaufmann Publishers, 1995, ISBN 1558602704

[3] Lehmann, Wunder, Selz, Schaltungsdesign mit VHDL, Franzis-Verlag, Poing, 1994, ISBN 3772361633, `http://www-itiv.etec.uni-karlsruhe.de/` `FORSCHUNG/VEROEFFENTLICHUNGEN/lws94/lws94.html`

[4] Giovanni De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill Book Company, 1994, ISBN 0070163332

[5] David R. Coelho, The VHDL Handbook, Kluwer Academic Publishers, Boston, 1989, ISBN 0792390318

[6] Jayaram Bhasker, A VHDL Primer, Prentice Hall, Englewood Cliffs, 1998, ISBN 0130965758

[7] Roger Lipsett, Carl Schaefer, Cary Ussery, VHDL: Hardware Description and Design, Kluwer Academic Publishers, Boston/Dordrecht/London, 1989, ISBN 079239030X

[8] Michael John Sebastian Smith, Application-Specific Integrated Circuits, Addison-Wesley Verlag, Juni 1997, ISBN 0201500221

[9] Bruce Schneider, Angewandte Kryptographie, Addison-Wesley Verlag, Bonn, 1996, ISBN 3893198547

[10] Xuejia Lai, On the Design and Security of Block Ciphers, Hartung-Gorre Verlag, Wissenschaftliche Publikationen, 1992

[11] Michael Welschenbach, Kryptographie in C und C++, Springer-Verlag, Berlin, 1998, ISBN 3540644040

[12] Richard E. Smith, Internet- Kryptographie, Addison-Wesley, Bonn, 1998, ISBN 3827313449

[13] Jens-Peter Kaps, Christof Paar, Fast DES Implementations for FPGAs and its application to a Universal Key-Search Machine, SAC' 98