GSI Helmholtzzentrum für Schwerionenforschung GmbH

WAVEFORM GENERATOR HADCON2

Remote controlled Pulser/Random Pulser on the HADCON2

Michail Pligouroudis
(Writer, Designer)
ATEI of Thessaloniki, Greece

Dr. Michael Traxler
(Supervisor)
GSI, Darmstadt, Germany

Darmstadt 2012

CONTENTS TABLE

Introduction 4
1. Hardware 5
2. FPGA Design (Main system) 6
2.1. ATMEL-FPGA communication 8
2.2. Address Decoder 9
2.3. PLL Frequency Multiplier 11
2.4. Pseudorandom Generator 12
2.5. Square Waveform Generator 13
2.6. Pulse Waveform Generator 14
2.7. Pulses with Pseudorandom timing delays 15
2.8. Reading Data 16
3. ATMEL Design 17
4. System Performance 18
5. How to connect it to PC / problem solutions 19
5.1. How to start 19
5.2. Connection problem fixing 19
6. Examples of use 20
7. Appendix A (FPGA 'VHDL' source code) 22
7.1. Main Design "Waveform Generator" 22
7.2. Component 0 "ATMEL-FPGA communication" 25
7.3. Component 1 "Address Decoder" 29
7.4. Component 2 "Pseudorandom Waveform Generator" 35
7.5. Component 4 "Square Waveform Generator" 37
7.6. Component 4 "Pulse Waveform Generator" 39
7.7. Component 5 "Pulse with Pseudorandom Delays Generator" 42
7.8. Component 6"Reading Data" 44
7.9. Component 7 "PLL clock multiplier" 50
8. Appendix B (ATMEL 'C' source code) 52
8.1. Header file "Waveform Generator Registers.h" 52
8.2. C file "Waveform Generator Registers.c" 53
9. Acknowledgments 57
10. References 57

Introduction

Waveform generator is a device that can produce variable signals (waves) at the output. This generator that we design is made in order to produce digital waveform outputs with variable programmable pulses. It can produce 4 types of digital signals (pseudorandom, pulse, square, pulse of 100ns with pseudorandom timing delays). It can produce all that signals with programmable variable timing steps of the waveform from 1,67s to 100ns or from 167ms to 10ns for the pseudorandom wave.

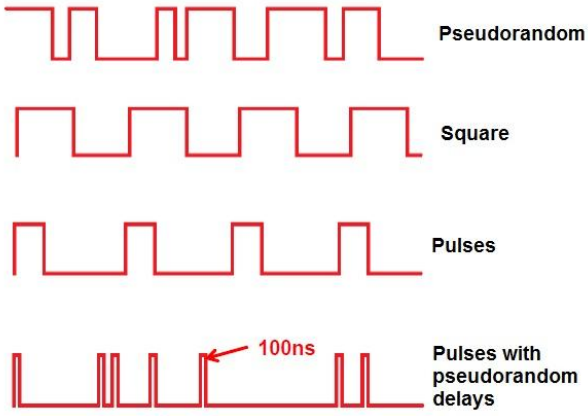


Fig. A: Waveforms

At the block diagram below you can see how we can use this generator remotely. We can connect from our pc through SSH tunnel to Raspberry Pi board. This board is a small cheap pc that runs Linux. So sending commands to Raspberry Pi we can communicate through USB 2.0 to Hadcon2 board. There the ATMEL receives our commands and sends them with RS232 protocol to the FPGA (MachX02) and form there we can have our waveform.

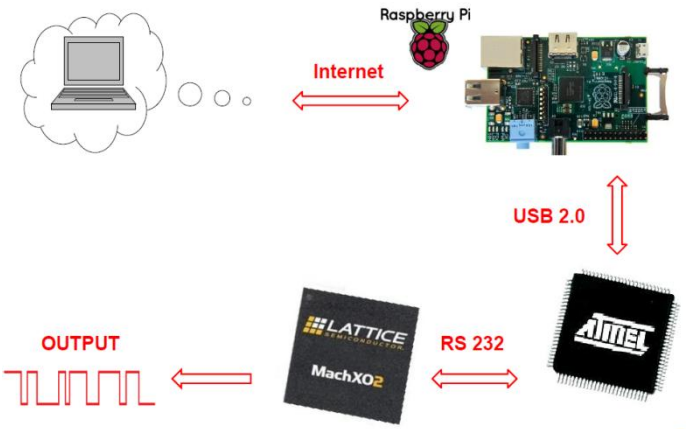


Fig. B: Block diagram

1. Hardware

The waveform generator is implemented on the PCB “HADCON2”. From this PCB we use only the FPGA (MachX02-1200), the ATMEL (AT90CAN128) and the FTDI chip for USB 2.0 communication with the pc. Also the oscillator of the board is 10MHz and the connection between FPGA and ATMEL is with 2 wires on board with RS232 protocol.

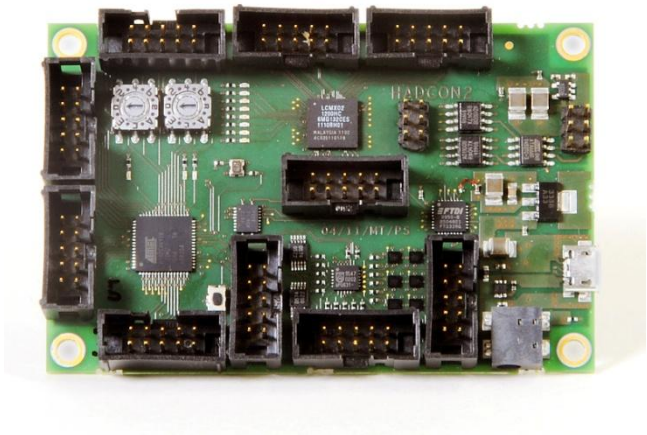


Fig. 1: Hadcon2 PCB

2. FPGA Design (Main system)

The FPGA design is consisted of eight different components.

I. **"ATMEL_FPGA_communication"** is made in order to connect with RS232 protocol the ATMEL with the FPGA.
II. **"address_decoder"** that is the heart of the system, it decodes the data and sends the correct enable signals and data to the other components to generate the correct waveform output.
III. **"reading_data"** that is the feedback loop of the system. It reads the values of the registers inside the address_decoder component and sends them through ATMEL_FPGA_communication to the pc.
IV. **"Pseudorandom"** it generates the variable pseudorandom waveform output.
V. **"square_waveform_generator"** it generates the variable square waveform output.
VI. **"pulse_generator"** it generates the variable pulses waveform output.
VII. **"Pseudorandom_delays"** it generates the pulses of 100ns in pseudorandom timing between them.
VIII. **"PLL clock multiplier"** it multiplies the oscillator frequency in order to produce 100MHz for the pseudorandom component.

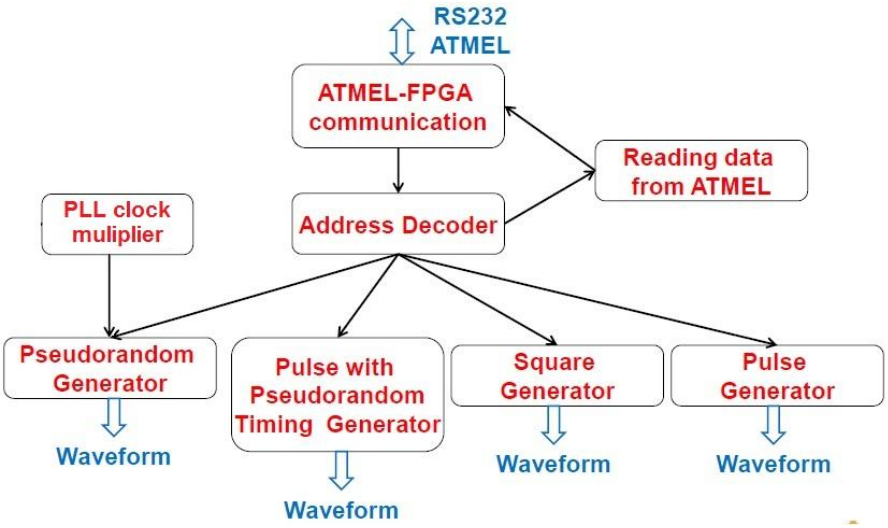


Fig. 2.1: FPGA design, block diagram

It can produce four types of digital programmable waveforms. It communicates with RS232 through ATMEL to the pc and from there you can send commands and use it as you want.

The whole system has 2 ports for UART communication with ATMEL,(**data_recieve** and **data_transmit)**, the **waveform** output, the synchronous **reset_button** to reset the whole system, the **clock** input for the oscillator 10MHz. It also has the **usb** output that is always at High Z, this output has no direct meaning for the architecture but it is made in order to protect the FPGA. It works like a buffer, because the hardware of HADCON2 has both in parallel the connection of USB and those 2 pins. If we do not put it in High Z we will have connection problem with the connection to the PC.

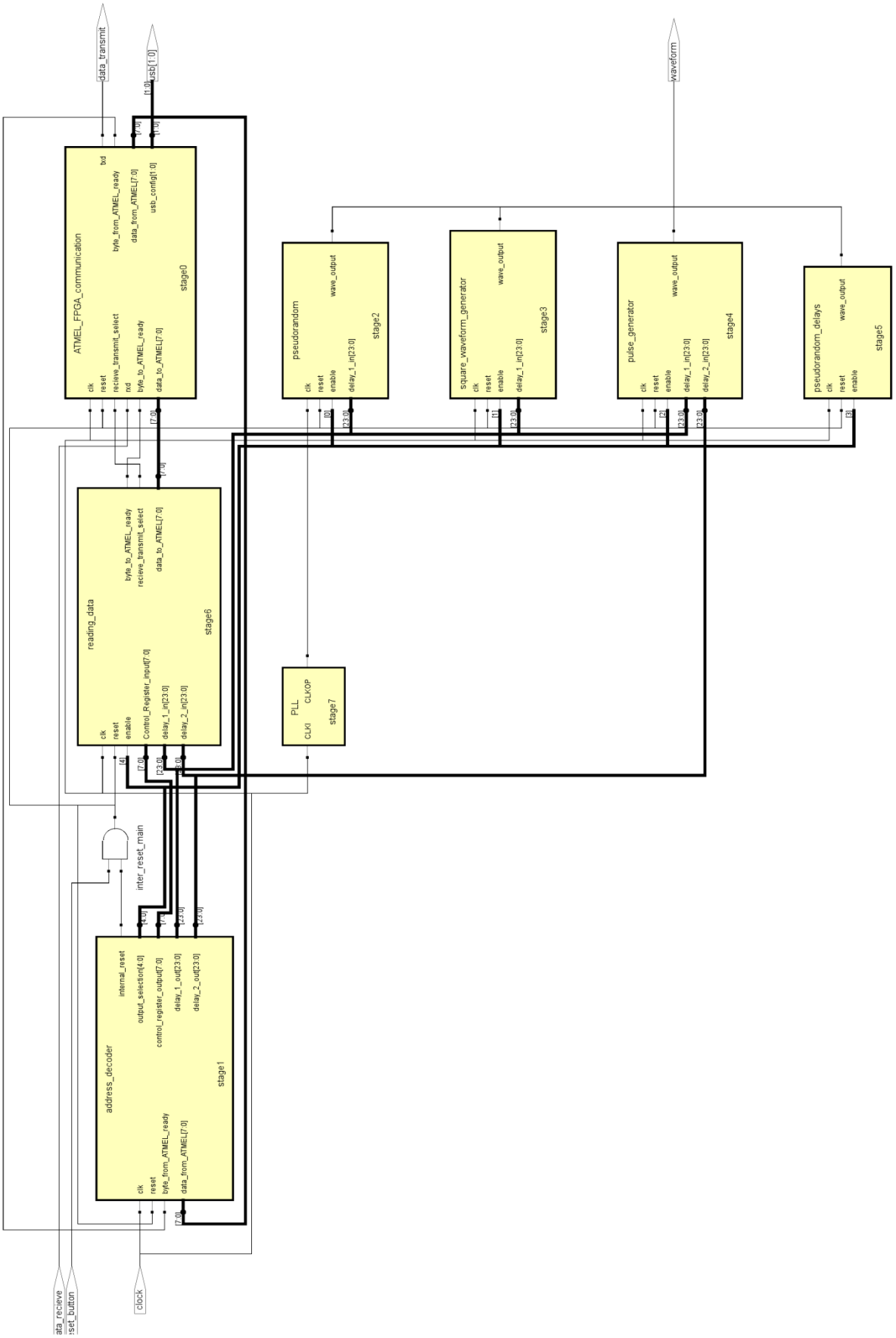


Fig. 2.2: FPGA design

2.1. ATMEL-FPGA communication

This component is the main input and output port of the FPGA that helps to receive and transmit data and have the RS232 communication between ATMEL and FPGA. It receives and sends data with baud rate 1,25Mbps. It uses 10bit per frame (1 start, 8 data and 1 stop).

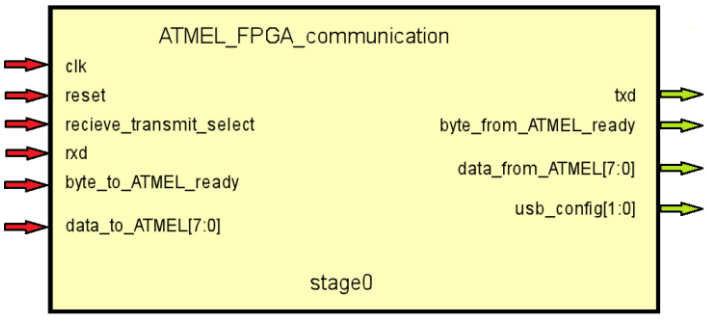


Fig. 2.1.1: ATMEL-FPGA communication component

It is designed with a state machine that waits synchronously for the receive_transimt_select signal the rxd signal and the byte_to_ATMEL_ready signal in order to take decision if it will start transmitting or receiving.

When the **receive_transmit_select** is '1' the system is ready for receiving data. To begin receiving it also waits the **rxd** signal from ATMEL. This signal is always '1' and when becomes '0' means that this is the start bit for the data stream. We receive the data from MSB to LSB and save them to a register for further use. When all data are received the signal **byte_from_ATMEL_ready** goes to '1' and the signal **data_from_ATMEL** has, in parallel, the value of the data.

When the **receive_transmit_select** is '0' the system is ready for transmitting data. To begin transmission it also waits the **byte_to_ATMEL_ready** signal to become '1'. This signal is always '0' and when it is '1' means that the component will start sending the value of the data that are at the input **data_to_ATMEL** with serial way.

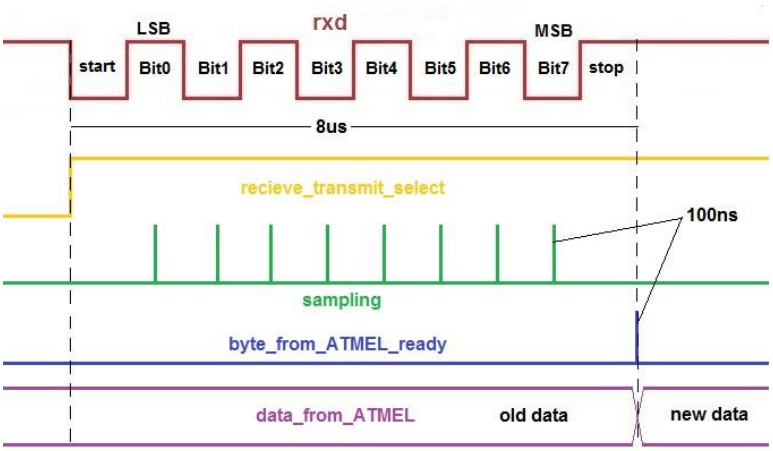


Fig. 2.1.2: Waveform for receiving data from ATMEL

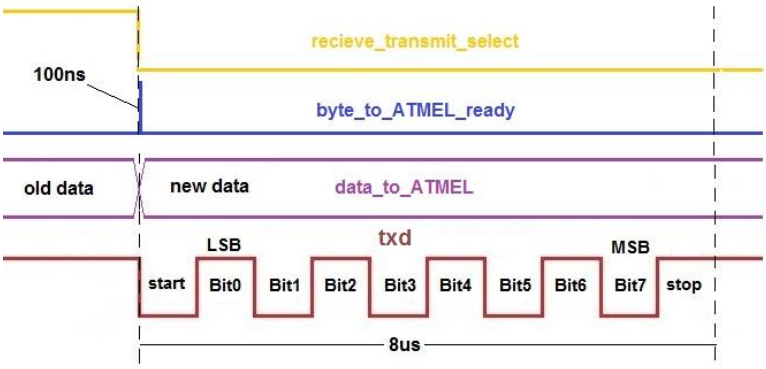


Fig. 2.1.3: Waveform for transmitting data to ATMEL

2.2. Address Decoder

The Address decoder is the "heart" of the Waveform Generator. With this component we can read and write the registers that it has inside it can decode them and after it can decide and enable the correct waveform-component and have the correct waveform output with the delays we want.

The decoder is made with a state machine that has two basic steps. The first step is to receive the address we have sent and after, the second step is to write the value we sent to that correct address. So every time we want to give instruction to the system we have to send both address and value.

Inside this component we have 3 Registers that we can use. The most important register is the **Control Register**, it is an 8bit register and with this we can control and choose what waveform we want. Also there are other 2 registers the **Delay1 Register** and the **Delay2 Register**. That two registers are used to fill them with the values of the delays we want to produce. That registers are smashed into 3 smaller one of 8bit in order to have access to them in better way with the RS232 communication. The registers map is shown at the following table with the addresses of each of them.

ADDRESS	REGISTER NAME
0x00	Control Register
0x01	Delay1 MSB
0x02	Delay1 MIDDLE
0x03	Delay1 LSB
0x04	Delay2 MSB
0x05	Delay2 MIDDLE
0x06	Delay2 LSB

Fig. 2.2.1: Registers map

The Delay1 register is used more often than the Delay2 because with this one we declare the delays of the pseudorandom waveform, the square waveform and also the period for the 'HIGH' of the pulse waveform. With the Delay2 we only declare the period for the 'LOW' of the pulse waveform.

In order to produce the delays at the waveforms we use counters with basic steps with same period of the clock. Our board has 100ns clock period and if we want to make big steps we have to count until very big numbers. This is the reason because that two registers use 24bits for the values. Now with that 24 bits

we can have waveforms with the smallest step T_{min}=100ns and biggest step $T_{max}=2^{24bit} \times 100ns = 1,677sec$

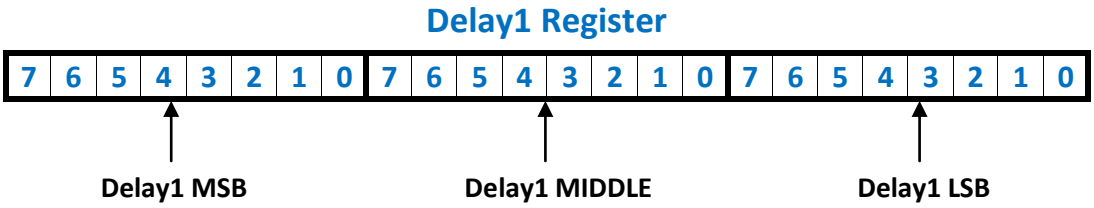


Fig. 2.2.2: Delay1 Register map

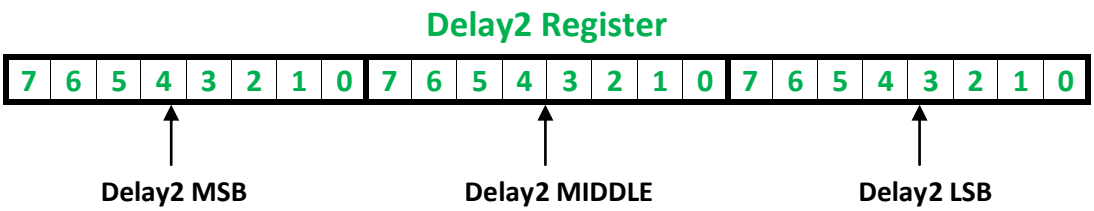


Fig. 2.2.3: Delay2 Register map

The Control Register is the main register of the system. With this one we can decide what waveform we want to enable. It is consisted of 8bits the first bit (from the MSB) is the enable bit. Every time we want to enable one component and have the waveform to the output, we must put it '1'. After that, we have the reset bit that resets internally by software the whole system if it is '1'. Next to this we have the read bit and when this bit is '1' we can enable the read component and read the values of the registers at the PC. We also have in the middle two free bits for future developing. The three LSB bits (select1, select2, select3) are used to select the waveform we want to produce.

Value	Result
0x40	Reset
0x81	Pseudorandom Wave
0x82	Square Wave
0x83	Pulse Wave
0x84	Pulses with pseudorandom delays
0xA0	Read Registers

Fig. 2.2.4: Control Register map

The address decoder component has an **internal_reset** output that resets the system by software, the **output_selection** that selects the waveform component every time we want new waveform, the **delay1_out** and **delay2_out** that have the values of the delays. Also it has a second **reset** that can externally reset this component by hardware, the **data_from_ATMEL** input that receives the addresses and values we want to write to the registers , the **byte_from_ATMEL_ready** that becomes '1' every time the data are ready to the input. Finally it has the **control_register_output** in order to be able to read the value of it with the read component to your PC screen.

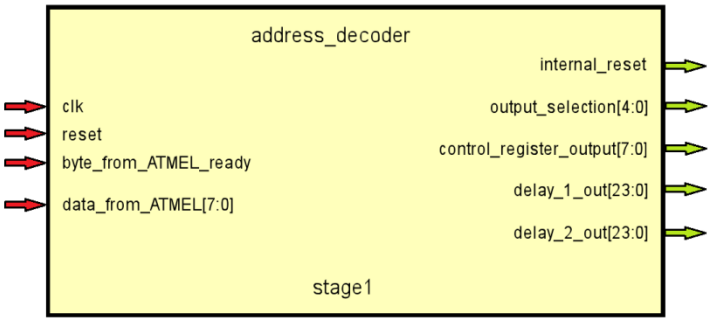


Fig. 2.2.1: Address decoder component

2.3. PLL Frequency Multiplier

The Hadcon2 PCB has oscillator crystal f=10MHz, and in order to make the Pseudorandom generator having smaller timing steps (the smallest step 10ns) we must have clock frequency 10 times bigger. This can be designed by using an internal PLL component inside FPGA that can multiply by 10 times the oscillator frequency.

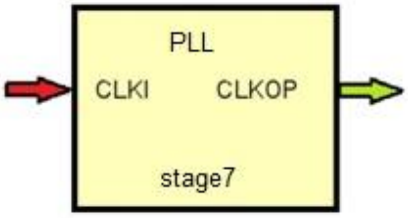


Fig. 2.3: PLL 100MHz component schematic

The PLL component is designed automatically by the IPexpress of Lattice Diamond software. It receives at the pin CLKI the oscillator clock signal f=10MHz and it multiplies it 10 times so we have at the output CLKOP a clock signal F=100MHz.

2.4. Pseudorandom Generator

The pseudorandom generator is used to generate the pseudorandom waveform with variable programmable steps of delays. To generate the pseudorandom sequence we use a Maximum-Length with Linear Feedback Shift Register of 32bits. The formula of this registers is:

$$Y = z^{32} + z^{22} + z^{2} + z + 1$$

This formula can produce $2^{32} - 1$ different pseudorandom digital numbers.

The component has an internal clock input from the PLL, and for this reason we have 100MHz clock in order to produce pseudorandom sequences with smaller step delays. So the maximum delay we can produce is **167,7ms** and the minimum **10ns**.

At the waveform below we have an example of a pseudorandom signal with step delay 1ms. To produce it we must give to delay1_in the following value:

$$\text{delay1_in}_{DEC} = \frac{1ms}{\frac{1}{100MHz}} = 100.000 \quad \rightarrow \quad \text{delay1_in}_{HEX} = 01\ 86\ A0$$

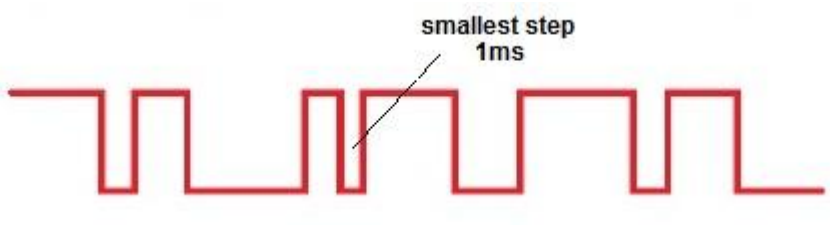


Fig. 2.4.1: Pseudorandom waveform with 1ms delay steps

In order to be programmed with the delay we want it is connected with the Delay1 register of the address decoder component and every time we change the value of this register we can directly change the delay steps. Also we have the **enable** input that must be '1' in order to be enabled and have the waveform at the output **wave_output**

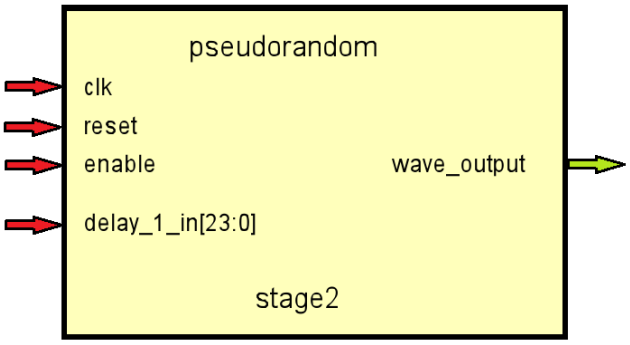


Fig. 2.4.2: Pseudorandom waveform generator component

2.5. Square Waveform Generator

The square generator is used to generate the square waveform with variable programmable steps.

The main idea of this design is based on a simple NOT gate that changes the waveform from 'HIGH' to 'LOW' every time the counter of the step delay expires.

At the waveform below we have an example of a square signal with step delay 500ns. In order to produce it we must give to delay1_in the following value:

$$\text{delay1_in}_{DEC} = \frac{500ns}{\frac{1}{10MHz}} = 5 \quad \rightarrow \quad \text{delay1_in}_{HEX} = 00\ 00\ 05$$

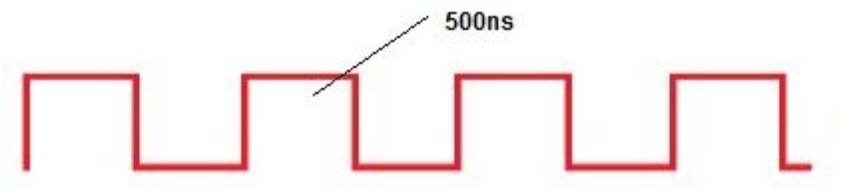


Fig. 2.5.1: Square waveform with 500ns delay steps

In order to be programmed with the delay we want, it is connected with the Delay1 register of the address decoder and every time we change the value of this register we can directly change the delay steps. Also we have the **enable** input that must be '1' in order to be enabled and have the waveform at the output **wave_output**

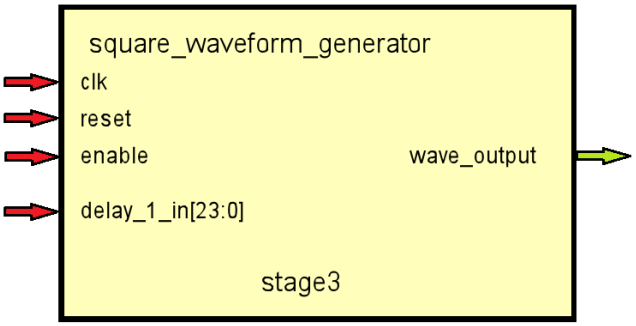


Fig. 2.5.2: Square waveform generator component

2.6. Pulse Waveform Generator

The pulse generator is used to generate the pulses with variable programmable periods for 'High' and 'Low'.

The main idea of this design is based on a simple NOT gate that changes the waveform from 'HIGH' to 'LOW' every time the counters of each step delay expires.

At the waveform below we have an example of a pulse signal with delay for 'High' 500ns and delay for 'Low' 1us. In order to produce it we must give to delay1_in and delay2_in the following values:

$$\text{delay1_in}_{DEC} = \frac{500ns}{\frac{1}{10MHz}} = 5 \quad \rightarrow \quad \text{delay1_in}_{HEX} = 00\ 00\ 05$$

$$\text{delay2_in}_{DEC} = \frac{1us}{\frac{1}{10MHz}} = 10 \quad \rightarrow \quad \text{delay2_in}_{HEX} = 00\ 00\ 0A$$

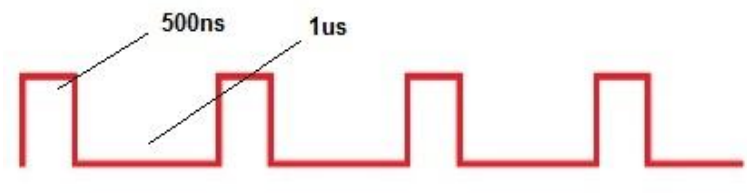


Fig. 2.6.1: Pulse waveform with 500ns 'High' & 1us 'Low'

In order to be programmed with the delays we want, it is connected with the Delay1 and Delay2 registers of the address decoder and every time we change the values of that registers we can directly change the delay steps. Also we have the **enable** input that must be '1' in order to be enabled and have the waveform at the output **wave_output**

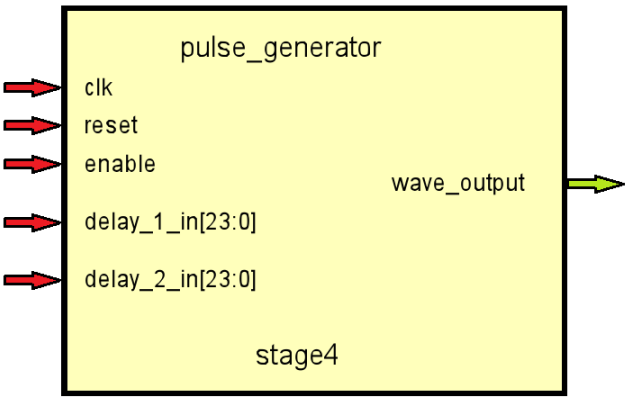


Fig. 2.6.2: Pulse waveform generator component

2.7. Pulses with Pseudorandom timing delays

The pseudorandom time component is used to produce small pulses of 100ns is a pseudorandom sequence between them.

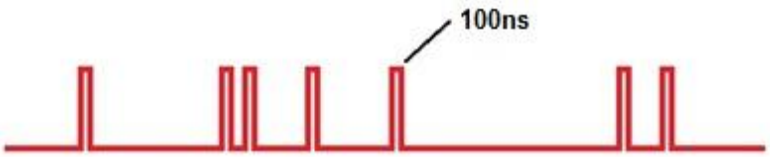


Fig. 2.7.1: Pseudorandom timing waveform

This pseudorandom generator is consisted of a Maximum-Length with Linear Feedback Shift Register of 32bits. The formula of this registers is:

$$Y = z^{32} + z^{22} + z^{2} + z + 1$$

This formula can produce $2^{32} - 1$ different pseudorandom delays between the pulses. So the maximum delay can be 1,677sec and the minimum 100ns.

From the block diagram of the component we can see that it has **clk** input of 10MHz, a synchronous **reset** input the **waveform output** where we can have the pseudorandom pulses and also an **enable** input in order to enable the component with '1' and send the pseudorandom pulses.

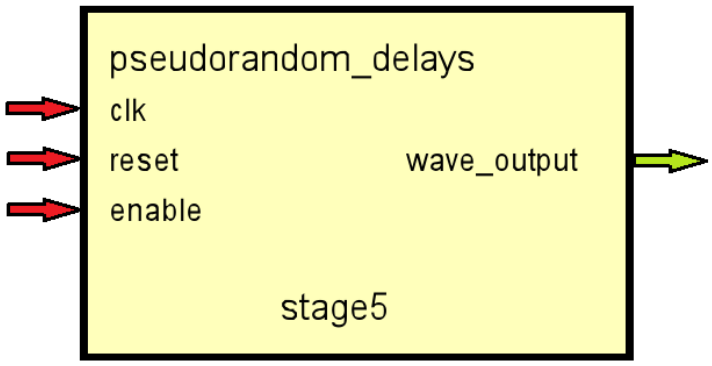


Fig. 2.7.2: Pulses with pseudorandom timing delays component

2.8. Reading Data

This component is the feedback of the system. With this one we can read all the values of the registers and send them through ATMEL to the PC. This can help us because with this way we can know in which situation is the Generator, we can know what waveform we have at the output with witch delays in our PC screen without measuring it with an oscilloscope.

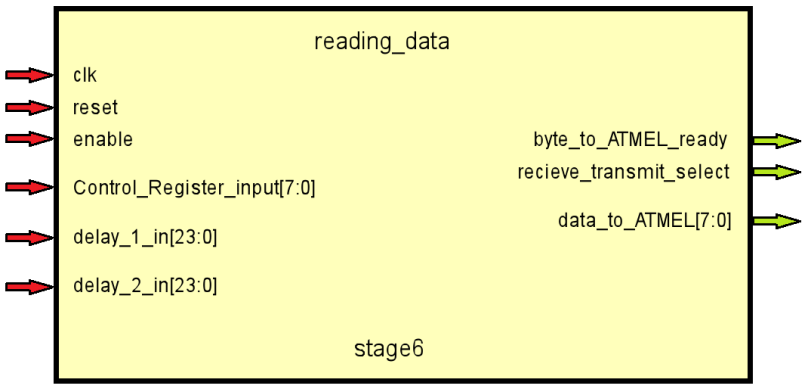


Fig. 2.8.1: Reading data component

The architecture of this component is consisted of a state machine that waits for the **enable** input to become '1'. When this pin is '1' the **receive_transmit_select** becomes from '1' to '0', that signal is connected with the ATMEL FPGA communication component and it means that we will start to transmit data from FPGA. After that it starts sending all the values of the registers one by one from the **data_to_ATMEL** pin and every time we have a new value of a register we send '1' from **byte_to_ATMEL_ready** pin in order the ATMEL FPGA communication component to know when to start send this value. The system reads the values of the registers in parallel way from the inputs **Control_Register_input**, **delay_1_in** and **delay_2_in** .

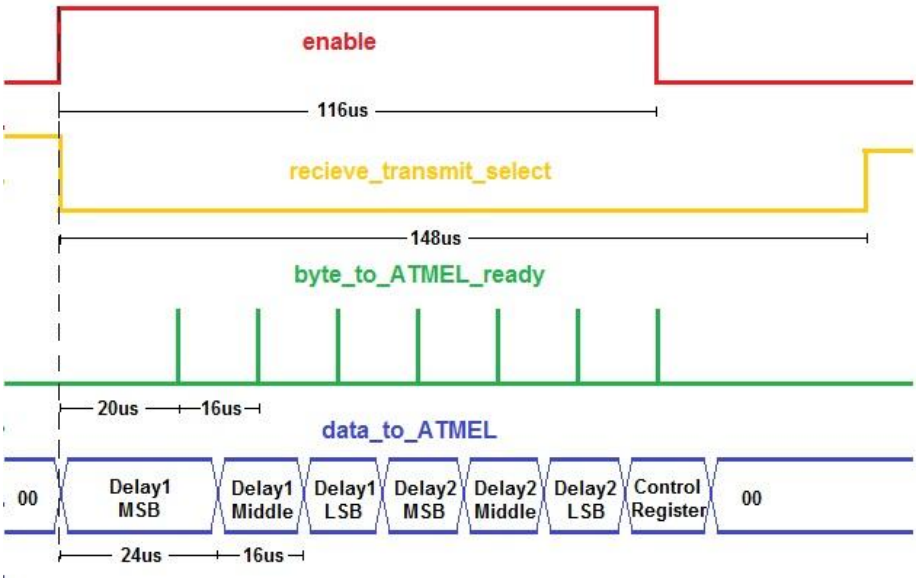


Fig. 2.8.2: Reading data waveform

3. ATMEL Design

We use the ATMEL microcontroller in order to make the connection between the PC and the FPGA through RS232 and USB protocol. The baud rate connection between the PC and ATMEL through USB2.0 is 115200bps and the baud rate between ATMEL and FPGA through RS232 is 1,25Mbps

We use the firmware API-Slave of the Hadcon2 board and inside this code we made 2 new commands **GNWR** (Generator Write) and **GNRE** (Generator Read).

GNWR

If we want to send and write a value to a register of the FPGA we must use the GNWR command. The syntax of this function is: **GNWR <address> <value>.**

For the address value we sent the number in hexadecimal system of the register's address we want to write and for the value we write again in Hexadecimal the value we want to write to that register.
With this way we can use and control the entire waveform generator by sending this command with the appropriate values and addresses each time.

At the picture below you can see the answer at your PC screen every time you use it.

```
RECV GNWR Control Register is written
```

Fig. 3.1: Answer at your screen by using the "GNWR 00 81"

GNRE

If we want to read the value of a register in the FPGA we must use the GNRE command. The syntax of this function is: **GNRE <address>**

For the address we sent the number in hexadecimal system of the register's address we want to read. In this way we can read only one register per time, if we want to read all the registers at the same time we can write the command GNRE with no giving any address.

At the picture below you can see the answer at your PC screen every time you use it. The first number is the address of the register after the value of the register and in the end the name of the register.

```
RECV GNRE 1 0 Delay1 MSB
RECV GNRE 2 0 Delay1 MIDDLE
RECV GNRE 3 4 Delay1 LSB
RECV GNRE 4 0 Delay2 MSB
RECV GNRE 5 0 Delay2 MIDDLE
RECV GNRE 6 1 Delay2 LSB
RECV GNRE 0 83 Control Register
```

Fig. 3.2: Answer at your screen by using the "GNRE"

4. System Performance

The system can produce digital waveforms with different delays between them. So the maximum delay that can be produced is **1,677sec** and the minimum **100ns** for the square pulse and pulses (100ns) with pseudorandom timing components. The pseudorandom component because of the different clock frequency can produce smaller delays (**10ns**) and the maximum **167,7ms**

The minimum period of the Pseudorandom signal with the smallest delay (100ns) can reach **7,16 minutes** and it can produce **4,3 billion** different number combinations.

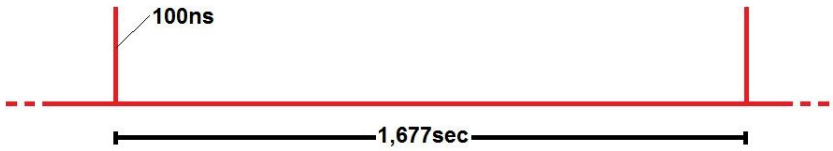


Fig. 4.1: Pulse Waveform with minimum and maximum delays for High and Low

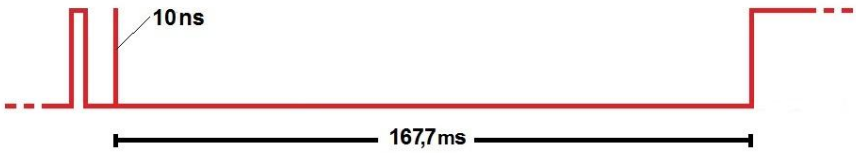


Fig. 4.2: Pseudorandom Waveform with minimum and maximum delays for High and Low

5. How to connect it to PC / problem solutions

5.1.How to start

To connect the Hadcon2 board to the pc you have to follow the steps below:

1. Plug in Hadcon2 board
2. Give command : **hadcon --device=/dev/hadcon2 –baudrate=115200**
3. Give command: **HELP** in order to have the control menu of the device

If at the screen you have the message: **"trying to read from device /dev/hadcon2"** you can try to chance the second step by giving the correct USB port in which Hadcon2 is connected.

ex. Give command : **hadcon --device=/dev/ttyUSB0 –baudrate=115200**

5.2.Connection problem fixing

If you have problems to connect the board to PC you can follow the steps below in order to fix the problem.

4. Give command: **ps aux |grep hadcon| grep username**
5. Plug off all USB devices
6. Give command: **/sbin/modprobe -r usbserial ftdi_sio**
7. Plug in Hadcon2 board
8. Give command: **hadcon --device=/dev/hadcon2 –baudrate=115200** or **hadcon --device=/dev/ttyUSBx –baudrate=115200**
9. Give command: **HELP** in order to have the control menu of the device

6. Examples of use

In this chapter we will have some examples about how we can use the Waveform generator and the commands you have to give. Firstly you have to follow the steps of chapter 5 to connect correctly to Hadcon2 main menu.

A. ***Pulse waveform with 400ns 'High' and 100ns 'Low'***

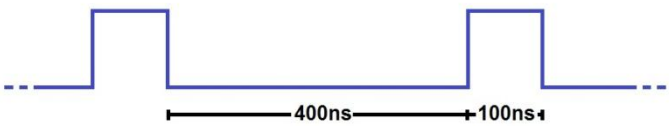
1. Write command ***GNWR 0x00 0x40*** to reset the system and previews data.
2. We must calculate the values of the register Delay1 and Delay2 for the HIGH and LOW. The clock frequency is 10MHz.

   $Delay1_{DEC} = \frac{400ns}{\frac{1}{10MHz}} = 4 \rightarrow Delay1_{HEX} = 00\ 00\ 04$

   $Delay2_{DEC} = \frac{100ns}{\frac{1}{10MHz}} = 1 \rightarrow Delay2_{HEX} = 00\ 00\ 01$

3. Write command ***GNWR 0x01 0x00*** write at the delay1 MSB register for 'High' (address 0x01 the value 0x00)
4. Write command ***GNWR 0x02 0x00*** write at the delay1 Middle register for 'High' (address 0x02 the value 0x00)
5. Write command ***GNWR 0x03 0x04*** write at the delay1 LSB register for 'High' (address 0x03 the value 0x04).
6. Write command ***GNWR 0x04 0x00*** write at the delay2 MSB register for 'Low' (address 0x04 the value 0x00)
7. Write command ***GNWR 0x05 0x00*** write at the delay2 Middle register for 'Low' (address 0x05 the value 0x00)
8. Write command ***GNWR 0x06 0x01*** write at the delay2 LSB register for 'Low' (address 0x06 the value 0x01)
9. Write command ***GNWR 0x00 0x83*** to enable from the Control Register the pulse waveform component (address 0x00 the value 0x83)

Now you have the Pulse waveform at the output.



B. ***Read the values of the Registers from the previews example.***

1. Write command ***GNRE*** to read all the values of all the registers at the same time.

At your PC screen you will have the following answer.

```
RECV GNRE 1 0 Delay1 MSB
RECV GNRE 2 0 Delay1 MIDDLE
RECV GNRE 3 4 Delay1 LSB
RECV GNRE 4 0 Delay2 MSB
RECV GNRE 5 0 Delay2 MIDDLE
RECV GNRE 6 1 Delay2 LSB
RECV GNRE 0 83 Control Register
```

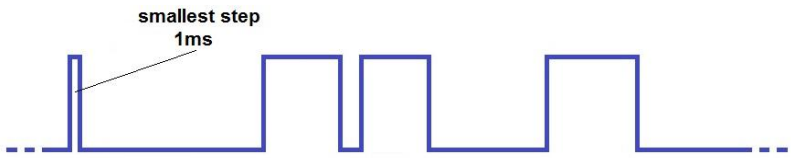
2. Write command ***GNRE 0x00*** to read the value only of the Control Register.
   At your PC screen you will have the following answer.

```
RECV GNRE 0 83 Control Register
```

C. ***Pseudorandom waveform with 1ms step delay***

1. Write command ***GNWR 0x00 0x40*** to reset the system and previews data.
2. We must calculate the value of the register Delay1. For the pseudorandom component because of the PLL clock we do not have any more clock frequency 10MHz, but 100MHz.
   $\text{Delay1}_{\text{DEC}} = \frac{1ms}{\frac{1}{100MHz}} = 100.000 \quad \rightarrow \quad \text{Delay1}_{\text{HEX}} = 01\ 86\ A0$
   After the calculation we must write that value to the Delay1 register, as you can see below.
3. Write command ***GNWR 0x01 0x01*** write at the delay1 MSB register (address 0x01 the value 0x01)
4. Write command ***GNWR 0x02 0x86*** write at the delay1 Middle register (address 0x02 the value 0x86)
5. Write command ***GNWR 0x03 0xA0*** write at the delay1 LSB register (address 0x03 the value 0xA0)
6. Write command ***GNWR 0x00 0x81*** to enable from the Control Register the pseudorandom component (address 0x00 the value 0x81)

Now you have the pseudorandom waveform at the output.



7. Appendix A (FPGA 'VHDL' source code)

7.1. Main Design "Waveform Generator"

```
-------------------------------------------------------------------------------
--                      SYSTEM WAVEFORM GENERATOR
--                        random_generator.vhd
--                                 ---
--                        Michail Pligouroudis
--                        M.Pligouroudis@gsi.de
--                                ---
--                       FPGA : MachXO2-1200HC
--                       ATMEL: AT90CAN128
--                       clock: f=10MHz, T=100ns
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- system description:   This system produces 4 types of digital waveform signals
--                       (pseudorandom, pulse, square, pulse 100ns with
--                       pseudorandom timing delays)
--                       It communicates through ATMEL to the pc and from there
--                       you can send commands in order to use it as you want.
--                       It can produse signal with programmable variable
--                       timing steps of the waveform (from 1,67s to 100ns)
-- components:           main system -> random_generator.vhd
--                       component0  -> ATMEL_FPGA_communication.vhd
--                       component1  -> address_decoder.vhd
--                       component2  -> pseudorandom.vhd
--                       component3  -> square_waveform_generator.vhd
--                       component4  -> pulse_generator.vhd
--                       component5  -> pseudorandom_delays.vhd
--                       component6  -> reading_data.vhd
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity random_generator is
  port (
    clock         : in  std_logic;  -- clock signal 10MHz
    reset_button  : in  std_logic;  -- reset button
    data_recieve  : in  std_logic;  -- recieve serial data from ATMEL (RXD pin)
    data_transmit : out std_logic;  -- transmit serial data to ATMEL (TXD pin)
    waveform      : out std_logic;  -- waveform output
    usb           : out std_logic_vector(1 downto 0));  -- output for the
                                                        -- parallel usb
                                                        -- connection to the pc

end random_generator;

architecture behave of random_generator is
  signal data_from_ATMEL_databus                     : std_logic_vector(7 downto 0)  := "00000000";
  signal data_to_ATMEL_databus                       : std_logic_vector(7 downto 0)  := "00000000";
  signal control_register_databus                    : std_logic_vector(7 downto 0)  := "00000000";
  signal ready                                       : std_logic                     := '0';
  signal delay_1_databus                             : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_2_databus                             : std_logic_vector(23 downto 0) := (others => '0');
  signal selection                                   : std_logic_vector(4 downto 0)  := "00000";
  signal inter_reset, inter_reset_main               : std_logic                     := '1';
  signal recieve_transmit_signal, data_ready_to_send : std_logic                     := '1';
  signal internal_clock : std_logic;

  component ATMEL_FPGA_communication
    port (
      clk                    : in  std_logic;  -- input clock f=10MHz
      reset                  : in  std_logic;  -- reset button
      recieve_transmit_select : in  std_logic;  -- input in order to select
                                                -- between recieve or transmmit
                                                -- (recieve -> 'HIGH', transmit -> 'LOW')
      rxd                    : in  std_logic;  -- recieve bitsream input pin
      txd                    : out std_logic;  -- transmmit bitstream output pin
      data_from_ATMEL        : out std_logic_vector(7 downto 0);  -- recieved byte from ATMEL
      byte_from_ATMEL_ready  : out std_logic;  -- 'HIGH' when the recieved byte
                                                -- from ATMEL is ready
      data_to_ATMEL          : in  std_logic_vector(7 downto 0);  -- value of FPGA registers
                                                                  -- in ordet to be send
                                                                  -- to ATMEL
      byte_to_ATMEL_ready    : in  std_logic;  -- 'HIGH' when the value of the
                                                -- registers is ready to the input
      usb_config             : out std_logic_vector(1 downto 0));  -- output will be set to 'Z' in order
                                                                   -- to be buffered from the parallelism
                                                                   -- of USB conection with
                                                                   -- the pc on hadcon2 pcb
  end component;

  component address_decoder
```

```
    port (
      clk                     : in  std_logic;  -- clock signal f=10MHz
      reset                   : in  std_logic;  -- reset signal
      internal_reset          : out std_logic;  -- internal reset signal
      data_from_ATMEL         : in  std_logic_vector(7 downto 0);  -- data and adresses input bus
      byte_from_ATMEL_ready   : in  std_logic;  -- 'HIGH' when data or addreses are ready to
                                                -- the input from component0
      output_selection        : out std_logic_vector(4 downto 0);  -- enable pins for selection
      control_register_output : out std_logic_vector(7 downto 0);  -- databus for the value of
                                                                   -- the Control Register
      delay_1_out             : out std_logic_vector(23 downto 0);  -- output databus for delay_1
      delay_2_out             : out std_logic_vector(23 downto 0));  -- output databus for dalay_2
  end component;

  component pseudorandom
    port (
      clk         : in  std_logic;      -- clock signal 10MHz
      reset       : in  std_logic;      -- system reset
      enable      : in  std_logic;      -- enable signal for the component
      delay_1_in  : in  std_logic_vector(23 downto 0);  -- input for the delay1 timing
      wave_output : out std_logic);     -- waveform output pin
  end component;

  component square_waveform_generator
    port (
      clk         : in  std_logic;      -- clock signal f=10MHz
      reset       : in  std_logic;      -- reset signal
      enable      : in  std_logic;      -- enable signal for the component
      delay_1_in  : in  std_logic_vector(23 downto 0);  -- input for the delay1 timing
      wave_output : out std_logic);     -- waveform output
  end component;

  component pulse_generator
    port (
      clk         : in  std_logic;      -- clock signal f=10MHz
      reset       : in  std_logic;      -- reset signal
      enable      : in  std_logic;      -- enable signal for the component
      delay_1_in  : in  std_logic_vector(23 downto 0);  -- delay for High
      delay_2_in  : in  std_logic_vector(23 downto 0);  -- delay for Low
      wave_output : out std_logic);     -- waveform output
  end component;

  component pseudorandom_delays
    port (
      clk         : in  std_logic;      -- clock signal 10MHz
      reset       : in  std_logic;      -- system reset
      enable      : in  std_logic;      -- enable signal for the component
      wave_output : out std_logic);     -- waveform output
  end component;

  component reading_data
    port (
      clk                     : in  std_logic;  -- clock signal f=10MHz
      reset                   : in  std_logic;  -- reset signal
      enable                  : in  std_logic;  -- enable signal for the component
      byte_to_ATMEL_ready     : out std_logic;  -- HIGH' when the value of the
                                                -- registers is ready to the output
      recieve_transmit_select : out std_logic;  -- output in order to select
                                                -- between recieve or transmmit
                                                -- (recieve -> 'HIGH', transmit -> 'LOW')
      data_to_ATMEL           : out std_logic_vector(7 downto 0);  -- value of FPGA registers
                                                                   -- in ordet to be send to ATMEL
      Control_Register_input  :     std_logic_vector(7 downto 0);  -- databus for the value of
                                                                   -- the Control Register
      delay_1_in              : in  std_logic_vector(23 downto 0);  -- input  for delay1 value
      delay_2_in              : in  std_logic_vector(23 downto 0));  -- input  for dalay2 value
  end component;

  component PLL
    port (
      CLKI: in  std_logic;              -- oscillator clock input 10MHz
      CLKOP: out  std_logic);           -- PLL internel clock output 100MHz
  end component;


begin  -- behave

  inter_reset_main <= reset_button and inter_reset;

  stage0 : ATMEL_FPGA_communication port map (
    clk                     => clock,
    reset                   => inter_reset_main,
    recieve_transmit_select => recieve_transmit_signal,  -- "high" recieve , "low" transmitt
    rxd                     => data_recieve,
    txd                     => data_transmit,
    data_from_ATMEL         => data_from_ATMEL_databus,
    byte_from_ATMEL_ready   => ready,
    data_to_ATMEL           => data_to_ATMEL_databus,
    byte_to_ATMEL_ready     => data_ready_to_send,
```

```
    usb_config              => usb);

  stage1 : address_decoder port map (
    clk                     => clock,
    reset                   => inter_reset_main,
    internal_reset          => inter_reset,
    data_from_ATMEL         => data_from_ATMEL_databus,
    byte_from_ATMEL_ready   => ready,
    output_selection        => selection,
    control_register_output => control_register_databus,
    delay_1_out             => delay_1_databus ,
    delay_2_out             => delay_2_databus);

  stage2 : pseudorandom port map (
    clk         => internal_clock,
    reset       => inter_reset_main,
    enable      => selection(0),
    delay_1_in  => delay_1_databus,
    wave_output => waveform);

  stage3 : square_waveform_generator port map (
    clk         => clock,
    reset       => inter_reset_main,
    enable      => selection(1),
    delay_1_in  => delay_1_databus,
    wave_output => waveform);

  stage4 : pulse_generator port map (
    clk         => clock,
    reset       => inter_reset_main,
    enable      => selection(2),
    delay_1_in  => delay_1_databus,
    delay_2_in  => delay_2_databus,
    wave_output => waveform);

  stage5 : pseudorandom_delays port map (
    clk         => clock,
    reset       => inter_reset_main,
    enable      => selection(3),
    wave_output => waveform);

  stage6 : reading_data port map(
    clk                     => clock,
    reset                   => inter_reset_main,
    enable                  => selection(4),
    byte_to_ATMEL_ready     => data_ready_to_send,
    recieve_transmit_select => recieve_transmit_signal,
    data_to_ATMEL           => data_to_ATMEL_databus,
    Control_Register_input  => control_register_databus,
    delay_1_in              => delay_1_databus,
    delay_2_in              => delay_2_databus);

  stage7 : PLL port map (
    CLKI     => clock,
    CLKOP    => internal_clock);
end behave;
```

7.2.Component 0 “ATMEL-FPGA communication”

```
-------------------------------------------------------------------------------
--                            COMPONENT 0
--                    ATMEL_FPGA_communication.vhd
--                                 ---
--                        Michail Pligouroudis
--                        M.Pligouroudis@gsi.de
--                                 ---
--                       FPGA : MachXO2-1200HC
--                       ATMEL: AT90CAN128
--                       clock: f=10MHz, T=100ns
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- component description:   This component creates the UART RS232 communication
--                          between ATMEL and FPGA. The communication protocol
--                          uses 10bits with baudrate 1.25Mbps (1 stop bit, 1
--                          finish bit and 8bit word)
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ATMEL_FPGA_communication is

  port (
    clk            : in std_logic;           -- input clock f=10MHz
    reset          : in std_logic;           -- reset button
    recieve_transmit_select: in  std_logic;  -- input in order to select
                                             -- between recieve or transmmit
                                             -- (recieve -> 'HIGH', transmit -> 'LOW')
    rxd            : in  std_logic;         -- recieve bitsream input pin
    txd            : out std_logic;          -- transmmit bitstream output pin
    data_from_ATMEL: out std_logic_vector(7 downto 0); -- recieved byte from ATMEL
    byte_from_ATMEL_ready  : out std_logic;  -- 'HIGH' when the recieved byte
                                             -- from ATMEL is ready
    data_to_ATMEL  : in std_logic_vector(7 downto 0);   -- value of FPGA registers
                                                        -- in ordet to be send
                                                        -- to ATMEL
    byte_to_ATMEL_ready    : in  std_logic;  -- 'HIGH' when the value of the
                                             -- registers is ready to the input
    usb_config     : out std_logic_vector(1 downto 0)); -- output will be set to 'Z' in order
                                                        -- to be buffered from the parallelism
                                                        -- of USB conection with
                                                        -- the pc on hadcon2 pcb
end ATMEL_FPGA_communication;


architecture behave of ATMEL_FPGA_communication is
  type states is (idle, decision, start_recieve, read0, read1, read2, read3, read4, read5, read6, read7,
finish_recieve,
                  start_transmit, send0, send1, send2, send3, send4, send5, send6, send7, end_transmit);
  signal present_state, next_state : states;
  signal data : std_logic_vector(7 downto 0):= "00000000";
  signal start_baudrate_delay, data_to_output, start_baudrate_delay1, finish_baudrate_delay: std_logic;
  signal saved_byte_to_send : std_logic_vector(7 downto 0):="00000000";
  signal counter : std_logic_vector(2 downto 0):="000";
  signal save_data : std_logic := '0';
  signal send_data : std_logic := '0';

begin
  usb_config <= "ZZ";                   -- pins N6,P6 are set to highZ in order
                                        -- to be buffered from the parallelism
                                        -- of USB conection with the pc

 ---------------------------------------------------------------------------
  -- purpose: selecting the state of the state machine both for recieving and transmitting
  -- type   : combinatorial
  -- inputs : present_state, recieve_transmit_select, rxd, finish_baudrate_delay, byte_to_ATMEL_ready
  -- outputs: next_state, data_to_output, send_data, txd, start_baudrate_delay1,
  --          start_baudrate_delay, save_data
 ---------------------------------------------------------------------------
  state_selection: process (present_state, recieve_transmit_select, rxd, finish_baudrate_delay,
byte_to_ATMEL_ready)
  begin  -- process state_selection
    start_baudrate_delay <= '0';
    start_baudrate_delay1 <= '0';
    save_data <= '0';
    data_to_output <= '0';
    send_data <= '0';

    case present_state is
      when idle =>
        next_state <= decision;
```

```
      when decision =>
        if recieve_transmit_select = '1' and rxd = '0' then
          next_state <= start_recieve;
          start_baudrate_delay1 <= '1';
        elsif recieve_transmit_select = '0' and byte_to_ATMEL_ready = '1' then
          next_state <= start_transmit;
          send_data <= '1';
          start_baudrate_delay <= '1';
        else
          next_state <= decision;
        end if;

          --RECIEVE--

      when start_recieve =>
        if finish_baudrate_delay = '1' then
          next_state <= read0;
          start_baudrate_delay1 <= '0';
          start_baudrate_delay <= '1';
          save_data <= '1';
        else
          next_state <= start_recieve;
          start_baudrate_delay1 <= '1';
        end if;

      when read0 to read7 =>
        if finish_baudrate_delay = '1' then
          save_data <= '1';
          next_state <= states'succ(present_state);
          start_baudrate_delay <= '1';
        else
          next_state <= present_state;
          start_baudrate_delay <= '1';
        end if;

      when finish_recieve =>
        if finish_baudrate_delay = '1' then
          next_state <= idle;
          data_to_output <= '1';
          start_baudrate_delay <= '0';
        else
          next_state <= finish_recieve;
          start_baudrate_delay <= '1';
        end if;

          --TRANSMIT--

      when start_transmit to end_transmit  =>
        if finish_baudrate_delay = '1' then
          send_data <= '1';
          if present_state = end_transmit then
            next_state <= idle;
            start_baudrate_delay <= '0';
            send_data <= '0';
          else
            next_state <= states'succ(present_state);
            start_baudrate_delay <= '1';
          end if;
        else
          start_baudrate_delay <= '1';
          next_state <= present_state;
        end if;

      when others => null;
    end case;
  end process state_selection;

 -----------------------------------------------------------------------------
  -- purpose: save 8bit data from the serial bitstream rxd
  -- type   : sequential
  -- inputs : clk, save_data, reset, rxd, data
  -- outputs: data
 -----------------------------------------------------------------------------
  store_data: process (clk, save_data, reset, rxd, data)
  begin  -- process store_data
    if reset = '0' then                 -- asynchronous reset (active low)
      data <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      if save_data = '1' then
        data <= rxd & data(7 downto 1);
      end if;
    end if;
  end process store_data;

 ------------------------------------------------------------------------------
   -- purpose: send data from FPGA to ATMEL with the outpout  pin txd
   -- type   : sequential
   -- inputs : clk, reset, send_data, data_to_ATMEL, next_state, saved_byte_to_send
```

```
   -- outputs: txd, saved_byte_to_send
 ------------------------------------------------------------------------------
  transmit_data: process (clk, reset, send_data, data_to_ATMEL, next_state, saved_byte_to_send)
  begin  -- process send_data
    if reset = '0' then                 -- asynchronous reset (active low)
      txd <= '1';
    elsif clk'event and clk = '1' then  -- rising clock edge
      if send_data = '1' then
        if next_state = start_transmit then
          txd <= '0';
          saved_byte_to_send <= data_to_ATMEL;
        elsif next_state  = end_transmit then
          txd <= '1';
        else
          txd <= saved_byte_to_send(0);
          saved_byte_to_send <= saved_byte_to_send(0) & saved_byte_to_send(7 downto 1);
          --shifting
        end if;
      end if;
    end if;
  end process transmit_data;

 -------------------------------------------------------------------------------
  -- purpose: show saved data from ATMEL to output
  -- type   : sequential
  -- inputs : clk, data_to_output, reset, data
  -- outputs: data_from_ATMEL, byte_from_ATMEL_ready
 -------------------------------------------------------------------------------
  show_saved_data_to_output: process (clk, data_to_output, reset, data)
  begin  -- process show_saved_data_to_output
    if reset = '0' then                 -- asynchronous reset (active low)
      data_from_ATMEL <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      byte_from_ATMEL_ready <= '0';
      if data_to_output = '1' then
        for i in 7 downto 0 loop
          data_from_ATMEL(i) <= data(i);
          byte_from_ATMEL_ready <= '1';
        end loop;  -- i
      end if;
    end if;
  end process show_saved_data_to_output;

 -------------------------------------------------------------------------------
  -- purpose: sychronus passing from present state to next state
  -- type   : sequential
  -- inputs : clk, reset, next_step, finish_baudrate_delay
  -- outputs: present_state
 -------------------------------------------------------------------------------
  steps: process (clk, reset, next_state)
  begin  -- process next_step
    if reset = '0' then                 -- asynchronous reset (active low)
      present_state <= idle;
    elsif clk'event and clk = '1' then  -- rising clock edge
      present_state <= next_state;
    end if;
  end process steps;

 ---------------------------------------------------------------------------
  -- purpose: generating dalays for correct synchronisation for sampling
  --          (delay1=800ns, delay2=300ns)
  -- type   : sequential
  -- inputs : clk, reset, start_baudrate_delay, start_baudrate_delay1, counter
  -- outputs: finish_baudrate_delay, counter
 ------------------------------------------------------------------ -----------
  delays: process (clk, reset, start_baudrate_delay, start_baudrate_delay1, counter)
  begin  -- process delays
    if reset = '0' then                 -- asynchronous reset (active low)
      counter <= "000";
      finish_baudrate_delay <= '1';
    elsif clk'event and clk = '1' then  -- rising clock edge
      if start_baudrate_delay = '1' then
       if counter = "111" then          -- delay1=800ns
        counter <= "000";
        finish_baudrate_delay <= '1';
       else
        counter <= counter + 1;
        finish_baudrate_delay <= '0';
       end if;
      elsif start_baudrate_delay1 = '1' then
       if counter = "010" then          -- delay2=300ns
        counter <= "000";
        finish_baudrate_delay <= '1';
       else
        counter <= counter + 1;
        finish_baudrate_delay <= '0';
       end if;
      end if;
    end if;
```

```
  end process delays;

end behave;
```

7.3.Component 1 “Address Decoder”

```
-------------------------------------------------------------------------------
--                            COMPONENT 1
--                         address_decoder.vhd
--                                ---
--                        Michail Pligouroudis
--                        M.Pligouroudis@gsi.de
--                                ---
--                       FPGA : MachXO2-1200HC
--                       ATMEL: AT90CAN128
--                       clock: f=10MHz, T=100ns
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- component description:   This component decodes the recieved data from ATMEL
--                          in order to understnd the whole system what
--                          waveform output we want and  with which delays. The
--                          decoder works with 2 bytes <address> <data>.
--
-- map of the decoder   :   <address> |   <name>
--                            0x00    | Control Register
--                            0x01    | Delay1 MSB
--                            0x02    | Delay1 Middle
--                            0x03    | Delay1 LSB
--                            0x04    | Delay2 MSB
--                            0x05    | Delay2 Middle
--                            0x06    | Delay2 LSB
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity address_decoder is
  port (
    clk      : in  std_logic;                -- clock signal f=10MHz
    reset    : in  std_logic;                -- reset signal
    internal_reset   : out std_logic;        -- internal reset signal
    data_from_ATMEL  : in  std_logic_vector(7 downto 0);        -- data and adresses input bus
    byte_from_ATMEL_ready   : in std_logic;  -- 'HIGH' when data or addreses are ready to
                                             -- the input from component0
    output_selection : out std_logic_vector(4 downto 0);        -- enable pins for selection
    control_register_output: out std_logic_vector(7 downto 0);  -- databus for the value of
                                                                -- the Control Register
    delay_1_out : out std_logic_vector(23 downto 0);            -- output databus for delay_1
    delay_2_out : out std_logic_vector(23 downto 0));           -- output databus for dalay_2
end address_decoder;

architecture behave of address_decoder is
  type states is(idle, address, data ) ;  -- states of the state machine
  signal next_state, present_state : states := idle;
  type states2 is(decision, send ) ;  -- states of the state machine
  signal next_state2, present_state2 : states2 := decision;
  signal address_register : std_logic_vector(7 downto 0) := "00000000";  -- address buffer register
  signal data_register : std_logic_vector(7 downto 0) := "00000000";     -- data buffer register
  signal control_register : std_logic_vector(7 downto 0) := "01000000";   -- CONTROL REGISTER
  signal control : std_logic_vector(7 downto 0) := "00000000";            --buffer from
                                                                         --control register
  signal delay1_MSB_register : std_logic_vector(7 downto 0) := "00000000";     -- DELAY1 REGISTER
  signal delay1_MIDDLE_register : std_logic_vector(7 downto 0) := "00000000";  -- DELAY1 REGISTER
  signal delay1_LSB_register : std_logic_vector(7 downto 0) := "00000000";     -- DELAY1 REGISTER
  signal delay2_MSB_register : std_logic_vector(7 downto 0) := "00000000";      -- DELAY2 REGISTER
  signal delay2_MIDDLE_register : std_logic_vector(7 downto 0) := "00000000";   -- DELAY2 REGISTER
  signal delay2_LSB_register : std_logic_vector(7 downto 0) := "00000000";      -- DELAY2 REGISTER
  signal finish_recieving, start_delay_for_sending, start_reading_signal : std_logic := '0';
  signal temp1, temp2 : std_logic_vector(7 downto 0) := "11111111";
  signal synch, finish_delay_for_sending : std_logic := '1';  -- synch= 'High' when data= 0x00 and
                                                              -- address=0x00 in order to
                                                              -- resynghronize the decoder
  signal count  : std_logic_vector(19 downto 0) := (others => '0');       -- counter
  signal output : std_logic_vector(4 downto 0) := "00000";    -- buffer for selection output

begin  -- behave

 ---------------------------------------------------------------------------
  -- purpose: selecting the state of the state machine
  -- type   : combinatorial
  -- inputs : present_state, data_from_ATMEL, synch, byte_from_ATMEL_ready,
  --          data_register, address_register, temp2, temp1
  -- outputs: finish_recieving, next_state, temp1, temp2, address_register,
  --          data_register
 ---------------------------------------------------------------------------
  state_machine: process (present_state, data_from_ATMEL, synch, byte_from_ATMEL_ready, data_register,
address_register, temp2, temp1)
  begin  -- process state_machine
```

```
  finish_recieving <= '0';

  case present_state is
    when idle =>
      if synch = '1' then
        next_state <= address;
        temp1 <= "11111111";
        temp2 <= "11111111";
        address_register <= "00000000";
        data_register <= "00000000";
      end if;

      if byte_from_ATMEL_ready = '1'  then
        next_state <= address;
        temp1 <= data_from_ATMEL;
        temp2 <= "11111111";
        address_register <= data_from_ATMEL;
        data_register <= data_register;
      else
        address_register <= address_register;
        data_register <= data_register;
        next_state <= idle;
        temp1 <= "11111111";
        temp2 <= "11111111";
      end if;

    when address =>
      if synch = '1' then
        next_state <= address;
        temp1 <= "11111111";
        temp2 <= "11111111";
        address_register <= "00000000";
        data_register <= "00000000";
      end if;

      if byte_from_ATMEL_ready = '1' then
        data_register <= data_from_ATMEL;
        address_register <= address_register;
        temp1 <= temp1;
        temp2 <= data_from_ATMEL;
        next_state <= data;
        finish_recieving <= '1';
      else
        address_register <= address_register;
        data_register <= data_register;
        temp1 <= temp1;
        temp2 <= temp2;
        next_state <= address;
      end if;

    when data =>
      if synch = '1' then
        address_register <= "00000000";
        data_register <= "00000000";
        next_state <= address;
        temp1 <= "11111111";
        temp2 <= "11111111";
      end if;

      if byte_from_ATMEL_ready = '1' then
        address_register <= data_from_ATMEL;
        data_register <= data_register;
        temp1 <= data_from_ATMEL;
        temp2 <= temp2;
        next_state <= address;
      else
        address_register <= address_register;
        data_register <= data_register;
        temp1 <= temp1;
        temp2 <= temp2;
        next_state <= data;
      end if;

     when others =>  null;

  end case;
  end process state_machine;

 ------------------------------------------------------------------------------
   -- purpose: sychronus passing from present state to next state
   -- type   : sequential
   -- inputs : clk, reset, next_state
   -- outputs: present_state
 ------------------------------------------------------------------------------
  step: process (clk, reset, next_state)
  begin  -- process steps
     if reset = '0' then                -- asynchronous reset (active low)
       present_state <=  idle;
     elsif clk'event and clk = '1' then  -- rising clock edge
```

```
        present_state <= next_state;
      end if;
  end process step;

 ---------------------------------------------------------------------------
  -- purpose: writing the correct data to the correct register decoding the address
  -- type   : sequential
  -- inputs : clk, reset, address_register, data_register, finish_recieving,
  --          count, control,bcontrol_register
  -- outputs: control_register, delay1_LSB_register, delay1_MIDDLE_register,
  --          delay1_LSB_register, delay2_LSB_register, delay2_MIDDLE_register,
  --          delay2_MSB_register
 ---------------------------------------------------------------------------
  decode1: process (clk, reset, address_register, data_register, finish_recieving, count, control,
control_register)
  begin  -- process decode
    if reset = '0' or control_register(6) = '1' then                    -- asynchronous reset (active low)
      control_register <= "00000000";
      delay1_MSB_register <= "00000000";
      delay1_MIDDLE_register <= "00000000";
      delay1_LSB_register <= "00000000";
      delay2_MSB_register <= "00000000";
      delay2_MIDDLE_register <= "00000000";
      delay2_LSB_register <= "00000000";
    elsif clk'event and clk = '1' then    -- rising clock edge
      if finish_recieving = '1' then
        case address_register is
          when "00000000" =>
            control_register <= data_register;
          when "00000001" =>
            delay1_MSB_register <= data_register;
          when "00000010" =>
            delay1_MIDDLE_register <= data_register;
          when "00000011" =>
            delay1_LSB_register <= data_register;
          when "00000100" =>
            delay2_MSB_register <= data_register;
          when "00000101" =>
            delay2_MIDDLE_register <= data_register;
          when "00000110" =>
            delay2_LSB_register <= data_register;
          when others => null;
        end case;
      end if;

      if count = x"00488" then
        control_register<= control;
      end if;

    end if;
  end process decode1;

 ----------------------------------------------------------------------------
  -- purpose: enabling the correct component in order to have the correct
  --          waveform output decoding from the data of the control register
  -- type   : sequential
  -- inputs : clk, reset, control_register, delay1_MSB_register, delay1_MIDDLE_register,
  --          delay1_LSB_register, delay2_MSB_register, delay2_MIDDLE_register,
  --          delay2_LSB_register, count, output
  -- outputs: delay_1_out, delay_2_out, output, start_reading_signal, output_selection
 ----------------------------------------------------------------------------
   decode2: process (clk, reset, control_register, delay1_MSB_register, delay1_MIDDLE_register,
delay1_LSB_register, delay2_MSB_register, delay2_MIDDLE_register, delay2_LSB_register, count, output)
  begin  -- process  decode2

    if reset = '0' or control_register(6) = '1' then                 -- asynchronous reset (active low)
      delay_1_out <= ( others => '0');
      delay_2_out <= ( others => '0');
      output(0) <= '0';
      output(1) <= '0';
      output(2) <= '0';
      output(3) <= '0';
      start_reading_signal <= '0';
    elsif clk'event and clk = '1' then  -- rising clock edge
      start_reading_signal <= '0';
      case control_register is
        when "10000001" =>
          output(0) <= '1';
          output(1) <= '0';
          output(2) <= '0';
          output(3) <= '0';
        when "10000010" =>
          output(0) <= '0';
          output(1) <= '1';
          output(2) <= '0';
          output(3) <= '0';
        when "10000011" =>
          output(0) <= '0';
          output(1) <= '0';
```

```
          output(2) <= '1';
          output(3) <= '0';
        when "10000100" =>
          output(0) <= '0';
          output(1) <= '0';
          output(2) <= '0';
          output(3) <= '1';
        when "10100000" =>
          start_reading_signal <= '1';
        when others =>
          output(0) <= '0';
          output(1) <= '0';
          output(2) <= '0';
          output(3) <= '0';

      end case;

      delay_1_out <= delay1_MSB_register & delay1_MIDDLE_register & delay1_LSB_register;
      delay_2_out <= delay2_MSB_register & delay2_MIDDLE_register & delay2_LSB_register;
      output_selection(0) <= output(0);
      output_selection(1) <= output(1);
      output_selection(2) <= output(2);
      output_selection(3) <= output(3);

      if count = x"00488" then
        start_reading_signal <= '0';
        output_selection(0) <= output(0);
        output_selection(1) <= output(1);
        output_selection(2) <= output(2);
        output_selection(3) <= output(3);
      end if;
    end if;
  end process  decode2;

 -----------------------------------------------------------------------------
  -- purpose: enabling component 6 "reading_data" in order to send the
  --          value of the registers to ATMEL
  -- type   : combinatorial
  -- inputs : present_state2, start_reading_signal, finish_delay_for_sending
  -- outputs: next_state2, start_delay_for_sending, output_selection
 -----------------------------------------------------------------------------
 reading: process (present_state2, start_reading_signal, finish_delay_for_sending)
 begin  -- process reading

   case present_state2 is
     when decision =>
       if start_reading_signal = '1' then
         next_state2 <= send;
         start_delay_for_sending <= '1';
         output_selection(4) <= '1';
       else
         next_state2 <= decision;
         start_delay_for_sending <= '0';
         output_selection(4) <= '0';
       end if;

     when send  =>
       if finish_delay_for_sending = '1' then
         next_state2 <= decision;
         start_delay_for_sending <= '0';
         output_selection(4) <= '0';
       else
         next_state2 <= send;
         start_delay_for_sending <= '1';
         output_selection(4) <= '1';
       end if;

     when others => null;

   end case;
 end process reading;

 -----------------------------------------------------------------------------
  -- purpose: sychronus passing from present state2 to next state2
  -- type   : sequential
  -- inputs : clk, reset, next_state2
  -- outputs: present_state2
 -----------------------------------------------------------------------------
  steps: process (clk, reset, next_state2)
  begin  -- process next_step
    if reset = '0' then                 -- asynchronous reset (active low)
      present_state2 <= decision;
    elsif clk'event and clk = '1' then  -- rising clock edge
      present_state2 <= next_state2;
    end if;
  end process steps;

 ---------------------------------------------------------------------------
  -- purpose: generating delay for the enabling signal for the component 6 "reading_data"
```

```
 --          in order to be synchronized (delay=116ums)
 -- type   : sequential
 -- inputs : clk, reset, start_delay_for_sending, count
 -- outputs: count, finish_delay_for_sending
-------------------------------------------------------------------------------
 generate_delay: process (clk, reset, start_delay_for_sending, count)
 begin  -- process generate_delay
   if reset = '0' then                  -- asynchronous reset (active low)
     count <= (others => '0');
   elsif clk'event and clk = '1' then  -- rising clock edge
     if start_delay_for_sending = '1' then
       if count = X"00488" then        -- delay=116us
         count <= (others => '0');
         finish_delay_for_sending <= '1';
       else
         count <= count + 1;
         finish_delay_for_sending <= '0';
       end if;
     end if;
   end if;
 end process generate_delay;

-------------------------------------------------------------------------------
 -- purpose: internal reset, in order to reset the whole system by software
 --          commands and not only with hardware way like reset buttons
 -- type   : sequential
 -- inputs : clk, reset, control_register
 -- outputs: internal_reset
-------------------------------------------------------------------------------
 intern_reset: process (clk, reset, control_register)
 begin  -- process intern_reset
   if reset = '0' then                  -- asynchronous reset (active low)
     internal_reset <= '1';
   elsif clk'event and clk = '1' then  -- rising clock edge
     if control_register(6) = '1' then
       internal_reset <= '0';
     else
       internal_reset <= '1';
     end if;
   end if;
 end process intern_reset;

-------------------------------------------------------------------------------
 -- purpose: synch= '1' when data= 0x00 and
 --          address=0x00 in order to
 --          resynghronize the decoder
 -- type   : sequential
 -- inputs : clk, reset, temp2, temp1
 -- outputs: synch
-------------------------------------------------------------------------------
 synchronize: process (clk, reset, temp2, temp1)
 begin  -- process synchronize
   if reset = '0' then                  -- asynchronous reset (active low)
     synch <= '1';
   elsif clk'event and clk = '1' then  -- rising clock edge
     if temp1 = "00000000" and temp2 = "00000000"  then
       synch <= '1';
     else
       synch <= '0';
     end if;
   end if;
 end process synchronize;

-------------------------------------------------------------------------------
 -- purpose: remain giving waveform to the main output while sending data to ATMEL
 -- type   : sequential
 -- inputs : clk, reset, start_reading_signal, output, control
 -- outputs: status, control_register_output,
-------------------------------------------------------------------------------
 allive: process (clk, reset, start_reading_signal, output, control)
 begin  -- process allive
   if reset = '0' then                  -- asynchronous reset (active low)
     control <= "00000000";
   elsif clk'event and clk = '1' then  -- rising clock edge
     control_register_output <= control;
     if start_reading_signal = '1' then
       case output is
         when "00001" =>
           control <= "10000001";
         when "00010" =>
           control <= "10000010";
         when "00100" =>
           control <= "10000011";
         when "01000" =>
           control <= "10000100";
         when others =>
           control <= "00000000";
       end case;
     end if;
```

```
    end if;
  end process allive;

end behave;
```

7.4.Component 2 “Pseudorandom Waveform Generator”

```
-----------------------------------------------------------------------------
--                            COMPONENT 2
--                          pseudorandom.vhd
--                                ---
--                       Michail Pligouroudis
--                       M.Pligouroudis@gsi.de
--                                ---
--                      FPGA : MachXO2-1200HC
--                      ATMEL: AT90CAN128
--                      clock: f=10MHz, T=100ns
-----------------------------------------------------------------------------

-----------------------------------------------------------------------------
-- component description:   This component generates the pseudorandom waveform
--                          with variable timins steps (from 1,677sec to 100ns).
--                          To generate pseudorandom sequense we use
--                          Maximum-Length with Linear Feedback Shift Register
--                          of 32 bits
--                          The formula of this generetor is :
--                          Y = z^32 + z^22 + z^2 + z + 1
-----------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity pseudorandom is
  port (
    clk          : in  std_logic;                    -- clock signal 10MHz
    reset        : in std_logic;                     -- system reset
    enable       : in std_logic;                     -- enable signal for the component
    delay_1_in   : in std_logic_vector(23 downto 0); -- input for the delay1 timing
    wave_output  : out std_logic);                   -- waveform output pin
end pseudorandom;

architecture behave of pseudorandom is
  signal rand_temp : std_logic_vector(31 downto 0) := (0 => '1', others => '0');
  signal temp : std_logic := '0';
  signal sync_reset : std_logic := '0';
  signal finish_delay : std_logic := '0';
  signal counter : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_time : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_change : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_change_flag : std_logic := '0';

begin  -- behave

-----------------------------------------------------------------------------
 -- purpose: Linear Feedback Shift Register of 32 bit in order to generate
 --          pseydorandom signal (Y = z^32 + z^22 + z^2 + z + 1 )
 -- type   : sequential
 -- inputs : clk, sync_reset, temp, rand_temp, finish_delay
 -- outputs: rand_temp, temp
-----------------------------------------------------------------------------
 shift_register: process (clk, sync_reset, temp, rand_temp, finish_delay)
 begin  -- process shift_register
   if sync_reset = '1' then                 -- asynchronous reset (active low)
      rand_temp <= (23 => '1', others => '0');
   elsif clk'event and clk = '1' then       -- rising clock edge
      if finish_delay = '1' then
         temp <= rand_temp(31) xnor rand_temp(21) xnor rand_temp(1) xnor rand_temp(0);
         rand_temp(31 downto 1) <= rand_temp((30) downto 0);
         rand_temp(0) <= temp;
      end if;
   end if;
 end process shift_register;

-----------------------------------------------------------------------------
 -- purpose: generating delay for the variable timing steps (from 1,677sec to 100ns)
 -- type   : sequential
 -- inputs : clk, sync_reset, counter, delay_time
 -- outputs: finish_delay, counter
-----------------------------------------------------------------------------
 delay: process (clk, sync_reset, counter, delay_time)
 begin  -- process delay
   if sync_reset = '1' then                 -- asynchronous reset (active low)
     counter <= (others => '0');
   elsif clk'event and clk = '1' then       -- rising clock edge
     counter <= counter + 1;
     finish_delay <= '0';
     if counter = delay_time then
       counter <= (0 => '1' , others => '0');
       finish_delay <= '1';
     end if;
```

```
    end if;
  end process delay;

 -----------------------------------------------------------------------------
  -- purpose: syghronous recieving timing data to produce delays only if the
  --          timing data is changed
  -- type   : sequential
  -- inputs : clk, sync_reset, delay_1_in, delay_time, delay_change
  -- outputs: delay_time, delay_change, delay_change_flag
 -----------------------------------------------------------------------------
  recieve_timing_data: process (clk, sync_reset, delay_1_in, delay_time, delay_change)
  begin  -- process recieve_timing_data
    if sync_reset = '1' then                 -- asynchronous reset (active low)
      delay_change_flag <= '0';
    elsif clk'event and clk = '1' then       -- rising clock edge
      delay_change <=  delay_1_in xnor delay_time;  -- checking if delay data changed
      if delay_change = X"FFFFFF"  then
        delay_time <= delay_1_in;
        delay_change_flag <= '0';
      else
        delay_change_flag <= '1';
        delay_time <= delay_1_in;
      end if;
    end if;
  end process recieve_timing_data;

 -----------------------------------------------------------------------------
  -- purpose: waveform to the output pin in the component is enabled, else 'Z'
  -- type   : sequential
  -- inputs : clk, sync_reset, enable, temp
  -- outputs: wave_output
 -----------------------------------------------------------------------------
  waveform_out: process (clk, sync_reset, enable, temp)
  begin  -- process waveform_out
    if sync_reset = '1' then                 -- asynchronous reset (active low)
      wave_output <= 'Z';
    elsif clk'event and clk = '1' then       -- rising clock edge
      if enable = '1' then
        wave_output <= temp;
      else
        wave_output <= 'Z';
      end if;
    end if;
  end process waveform_out;

 -----------------------------------------------------------------------------
  -- purpose: sychronus reset button, resetting also when delay data are changed
  -- type   : sequential
  -- inputs : clk, reset
  -- outputs: sync_reset
 -----------------------------------------------------------------------------
  synchronus_reset: process (clk, reset, delay_change_flag)
  begin  -- process synchronus_reset
    if clk'event and clk = '1' then  -- rising clock edge
      if reset = '0' or delay_change_flag = '1' then
        sync_reset <= '1';
      elsif reset = '1'  then
        sync_reset <= '0';
      end if;
    end if;
  end process synchronus_reset;

end behave;
```

7.5.Component 4 “Square Waveform Generator”

```
-------------------------------------------------------------------------------
 --                            COMPONENT 3
 --                    square_waveform_generator.vhd
 --                                 ---
 --                        Michail Pligouroudis
 --                        M.Pligouroudis@gsi.de
 --                                 ---
 --                       FPGA : MachXO2-1200HC
 --                       ATMEL: AT90CAN128
 --                       clock: f=10MHz, T=100ns
 -----------------------------------------------------------------------------

 -----------------------------------------------------------------------------
 -- component description:   This component generates the square waveform
 --                          with variable timing steps (from 1,677sec to 100ns).
 -----------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity square_waveform_generator is

  port (
    clk         : in  std_logic;   -- clock signal f=10MHz
    reset       : in  std_logic;   -- reset signal
    enable      : in  std_logic;   -- enable signal for the component
    delay_1_in  : in  std_logic_vector(23 downto 0);   -- input for the delay1 timing
    wave_output : out std_logic);  -- waveform output
end square_waveform_generator;

architecture behave of square_waveform_generator is
  signal delay_finish : std_logic := '0';
  signal wave : std_logic := '0';
  signal counter : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_change : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_time : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_change_flag : std_logic := '0';
  signal sync_reset : std_logic := '0';

begin  -- behave

 -----------------------------------------------------------------------------
  -- purpose: generating the square shape to waveform
  -- type   : sequential
  -- inputs : clk, sync_reset, delay_finish, wave
  -- outputs: wave
 -----------------------------------------------------------------------------
  square: process (clk, sync_reset, delay_finish, wave)
  begin  -- process square
    if sync_reset = '1' then                   -- asynchronous reset (active low)
      wave <= '0';
    elsif clk'event and clk = '1' then    -- rising clock edge
      if delay_finish = '1' then
        wave <= not wave;
      else
        wave <= wave;
      end if;
    end if;
  end process square;

 -----------------------------------------------------------------------------
  -- purpose: genarate delay for the variable timing steps (from 1,677sec to 100ns)
  -- type   : sequential
  -- inputs : clk, sync_reset, counter, delay_time
  -- outputs: delay_finish, counter
 -----------------------------------------------------------------------------
  delays: process (clk, sync_reset, counter, delay_time)
  begin  -- process delays
    if sync_reset = '1'  then                 -- asynchronous reset (active low)
      counter <= (others => '0');
      delay_finish <= '1';
    elsif clk'event and clk = '1' then  -- rising clock edge
      if counter = delay_time  then
        delay_finish <= '1';
        counter <= (0 => '1', others => '0');
      else
        counter <= counter + 1;
        delay_finish <= '0';
      end if;

    end if;
  end process delays;
```

```
 ---------------------------------------------------------------------------
  -- purpose: syghronous recieving timing data to produce delays only if the
  --          timing data is changed
  -- type   : sequential
  -- inputs : clk, sync_reset, delay_1_in, delay_time, delay_change
  -- outputs: delay_time, delay_change, delay_change_flag
 ---------------------------------------------------------------------------
  delay_changed: process (clk, sync_reset, delay_1_in, delay_time, delay_change)
  begin  -- process delay_data
    if sync_reset = '1' then                 -- asynchronous reset (active low)
      delay_change_flag <= '0';
    elsif clk'event and clk = '1' then       -- rising clock edge
      delay_change <=  delay_1_in xnor delay_time;  -- checking if delay data changed
      if delay_change = X"FFFFFF"  then
        delay_time <= delay_1_in;
        delay_change_flag <= '0';
      else
        delay_change_flag <= '1';
        delay_time <= delay_1_in;
      end if;
    end if;
  end process delay_changed;

 ---------------------------------------------------------------------------
  -- purpose: waveform to the output pin in the component is enabled, else 'Z'
  -- type   : sequential
  -- inputs : clk, sync_reset, enable, wave
  -- outputs: wave_output
 ---------------------------------------------------------------------------
  wave_out: process (clk, sync_reset, enable, wave)
  begin  -- process wave
    if sync_reset = '1' then                 -- asynchronous reset (active low)
      wave_output <= 'Z';
    elsif clk'event and clk = '1' then  -- rising clock edge
      if enable = '1' then
        wave_output <= wave;
      else
        wave_output <= 'Z';
      end if;
    end if;
  end process wave_out;

 --------------------------------------------------------------------------
  -- purpose: sychronus reset button, resetting also when delay data are changed
  -- type   : sequential
  -- inputs : clk, reset, delay_change_flag
  -- outputs: sync_reset
 --------------------------------------------------------------------------
  synchronus_reset: process (clk, reset, delay_change_flag)
  begin  -- process synchronus_reset
    if clk'event and clk = '1' then  -- rising clock edge
      if reset = '0' or delay_change_flag = '1' then
        sync_reset <= '1';
      elsif reset = '1'  then
        sync_reset <= '0';
      end if;
    end if;
  end process synchronus_reset;

end behave;
```

7.6.Component 4 “Pulse Waveform Generator”

```
-----------------------------------------------------------------------------
 --                            COMPONENT 4
 --                        pulse_generator.vhd
 --                               ---
 --                       Michail Pligouroudis
 --                       M.Pligouroudis@gsi.de
 --                               ---
 --                      FPGA : MachXO2-1200HC
 --                      ATMEL: AT90CAN128
 --                      clock: f=10MHz, T=100ns
 ----------------------------------------------------------------------------

 ----------------------------------------------------------------------------
 -- component description:   This component generates the pulse waveform
 --                          with variable timing steps (from 1,677sec to 100ns).
 --                          It can also generate waveform with different
 --                          timing delays for the 'High' and 'Low' using both
 --                          the delay1 and delay2 registers.
 ----------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity pulse_generator is

  port (
    clk         : in  std_logic;   -- clock signal f=10MHz
    reset       : in  std_logic;   -- reset signal
    enable      : in  std_logic;   -- enable signal for the component
    delay_1_in  : in  std_logic_vector(23 downto 0);   -- delay for High
    delay_2_in  : in  std_logic_vector(23 downto 0);   -- delay for Low
    wave_output : out std_logic);  -- waveform output
end pulse_generator;

architecture behave of pulse_generator is

  type states is (HIGH_state, LOW_state);
  signal present_state, next_state : states;
  signal delay_finish_H, delay_finish_L : std_logic := '0';
  signal wave : std_logic := '0';
  signal counter_H : std_logic_vector(23 downto 0) := (others => '0');
  signal counter_L : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_H_change : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_L_change : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_H : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_L : std_logic_vector(23 downto 0) := (others => '0');
  signal delay_H_change_flag : std_logic := '0';
  signal delay_L_change_flag : std_logic := '0';
  signal delay_H_start : std_logic := '0';
  signal delay_L_start : std_logic := '0';
  signal sync_reset : std_logic := '0';


begin
 ----------------------------------------------------------------------------
  -- purpose: state machine in order to produce 2 states of waveform output High-Low
  -- type   : combinatorial
  -- inputs : present_state, delay_finish_L, delay_finish_H
  -- outputs: delay_L_start, delay_H_start, wave, next_state
 ----------------------------------------------------------------------------
  state_machine: process (present_state, delay_finish_L, delay_finish_H)
  begin  -- process state_machine
    delay_L_start <= '0';
    delay_H_start <= '0';

    case present_state is
      when HIGH_state =>
        if delay_finish_H = '1'  then
          next_state <= LOW_state;
          delay_L_start <= '1';
          delay_H_start <= '0';
          wave <= '0';
        else
          wave <= '1';
          next_state <= HIGH_state;
          delay_L_start <= '0';
          delay_H_start <= '1';
        end if;

      when LOW_state =>
        if delay_finish_L = '1' then
          next_state <= HIGH_state;
```

```
          delay_L_start <= '0';
          delay_H_start <= '1';
          wave <= '1';
        else
          wave <= '0';
          next_state <= LOW_state;
          delay_L_start <= '1';
          delay_H_start <= '0';
        end if;

      when others => null;

    end case;
  end process state_machine;

 ---------------------------------------------------------------------------
  -- purpose: sychronus passing from next state to present state
  -- type   : sequential
  -- inputs : clk, sync_reset, next_state
  -- outputs: present_state
 ---------------------------------------------------------------------------
  change_states: process (clk, sync_reset, next_state)
  begin  -- process change_states
    if sync_reset = '1' then            -- asynchronous reset (active low)
      present_state <= HIGH_state;
    elsif clk'event and clk = '1' then  -- rising clock edge
      present_state <= next_state;
    end if;
  end process change_states;

 ---------------------------------------------------------------------------
  -- purpose: genarate variable delay for 'High' (from 1,677sec to 100ns)
  -- type   : sequential
  -- inputs : clk, sync_reset, delay_H, delay_H_start, counter_H
  -- outputs: delay_finish_H, counter_H
 ---------------------------------------------------------------------------
  delay_High: process (clk, sync_reset, delay_H, delay_H_start, counter_H)
  begin  -- process delays
    if sync_reset = '1'  then                 -- asynchronous reset (active low)
      counter_H <= (others => '0');
    elsif clk'event and clk = '1' then        -- rising clock edge
      if counter_H = delay_H and delay_H_start = '1'  then
        delay_finish_H <= '1';
        counter_H <= (0 => '1', others => '0');
      elsif delay_H_start = '1' then
        counter_H <= counter_H + 1;
        delay_finish_H <= '0';
      end if;
    end if;
  end process delay_High;

 ---------------------------------------------------------------------------
  -- purpose: genarate variable delay for 'Low' (from 1,677sec to 100ns)
  -- type   : sequential
  -- inputs : clk, sync_reset, delay_L, delay_L_start, counter_L
  -- outputs: delay_finish_L, counter_L
 ---------------------------------------------------------------------------
  delay_Low: process (clk, sync_reset, delay_L, delay_L_start, counter_L)
  begin  -- process delays
    if sync_reset = '1'  then                 -- asynchronous reset (active low)
      counter_L <= (others => '0');
    elsif clk'event and clk = '1' then        -- rising clock edge
      if counter_L = delay_L and delay_L_start = '1' then
        delay_finish_L <= '1';
        counter_L <= (0 => '1', others => '0');
      elsif delay_L_start = '1' then
        counter_L <= counter_L + 1;
        delay_finish_L <= '0';
      end if;
    end if;
  end process delay_Low;

 ---------------------------------------------------------------------------
  -- purpose: syghronous recieving timing data to produce delay for 'High' only
  --          if the timing data is changed
  -- type   : sequential
  -- inputs : clk, sync_reset, delay_1_in, delay_H_change, delay_H
  -- outputs: delay_H, delay_H_change, delay_H_change_flag
 ---------------------------------------------------------------------------
  delay_H_changed: process (clk, sync_reset, delay_1_in, delay_H_change, delay_H)
  begin  -- process delay_data
    if sync_reset = '1' then                 -- asynchronous reset (active low)
      delay_H_change_flag <= '0';
    elsif clk'event and clk = '1' then       -- rising clock edge
      delay_H_change <=  delay_H xnor delay_1_in;  --checking if delay data changed
      if delay_H_change = X"FFFFFF"  then
        delay_H <= delay_1_in;
        delay_H_change_flag <= '0';
      else
```

```
        delay_H_change_flag <= '1';
        delay_H <= delay_1_in;
      end if;
    end if;
  end process delay_H_changed;

 ---------------------------------------------------------------------------
  -- purpose: syghronous recieving timing data to produce delay for 'Low' only
  --          if the timing data is changed
  -- type   : sequential
  -- inputs : clk, sync_reset, delay_2_in, delay_L_change, delay_L
  -- outputs: delay_L, delay_L_change, delay_L_change_flag
 ---------------------------------------------------------------------------
  delay_L_changed: process (clk, sync_reset, delay_2_in, delay_L_change, delay_L)
  begin  -- process delay_data
    if sync_reset = '1' then                 -- asynchronous reset (active low)
      delay_L_change_flag <= '0';
    elsif clk'event and clk = '1' then       -- rising clock edge
      delay_L_change <=  delay_L xnor delay_2_in;  -- checking if delay data changed
      if delay_L_change = X"FFFFFF"  then
        delay_L <= delay_2_in;
        delay_L_change_flag <= '0';
      else
        delay_L_change_flag <= '1';
        delay_L <= delay_2_in;
      end if;
    end if;
  end process delay_L_changed;

 ---------------------------------------------------------------------------
  -- purpose: waveform to the output pin in the component is enabled, else 'Z'
  -- type   : sequential
  -- inputs : clk, sync_reset, enable, wave
  -- outputs: wave_output
 ---------------------------------------------------------------------------
  wave_out: process (clk, sync_reset, enable, wave)
  begin  -- process wave
    if sync_reset = '1' then                 -- asynchronous reset (active low)
      wave_output <= 'Z';
    elsif clk'event and clk = '1' then  -- rising clock edge
      if enable = '1' then
        wave_output <= wave;
      else
        wave_output <= 'Z';
      end if;
    end if;
  end process wave_out;

 --------------------------------------------------------------------------
  -- purpose: sychronus reset button, resetting also when delay data are changed
  -- type   : sequential
  -- inputs : clk, reset, delay_L_change_flag, delay_H_change_flag
  -- outputs: sync_reset
 --------------------------------------------------------------------------
  synchronus_reset: process (clk, reset, delay_L_change_flag, delay_H_change_flag)
  begin  -- process synchronus_reset
    if clk'event and clk = '1' then  -- rising clock edge
      if reset = '0' or delay_H_change_flag = '1' or delay_L_change_flag = '1' then
        sync_reset <= '1';
      elsif reset = '1'  then
        sync_reset <= '0';
      end if;
    end if;
  end process synchronus_reset;

end behave;
```

7.7.Component 5 “Pulse with Pseudorandom Delays Generator”

```
-------------------------------------------------------------------------------
 --                            COMPONENT 5
 --                      pseudorandom_delays.vhd
 --                                ---
 --                        Michail Pligouroudis
 --                        M.Pligouroudis@gsi.de
 --                                ---
 --                       FPGA : MachXO2-1200HC
 --                       ATMEL: AT90CAN128
 --                       clock: f=10MHz, T=100ns
 -----------------------------------------------------------------------------

 -----------------------------------------------------------------------------
 -- component description:   This component generates pulses of 100ns with
 --                          pseudorandom timing delays between that pulses.
 --                          This component has no programmable variable timing.
 --                          To generate pseudorandom timing we use
 --                          Maximum-Length with Linear Feedback Shift Register
 --                          of 32 bits
 --                          The formula of this generetor is :
 --                          Y = z^32 + z^22 + z^2 + z + 1
 -----------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity pseudorandom_delays is
  port (
    clk : in  std_logic;                        -- clock signal 10MHz
    reset : in std_logic;                       -- system reset
    enable : in std_logic;                      -- enable signal for the component
    wave_output   : out std_logic);             -- waveform output
end pseudorandom_delays;

architecture behave of pseudorandom_delays is
  signal rand_temp : std_logic_vector(31 downto 0) := (0 => '1', others => '0');
  signal temp  : std_logic := '0';
  signal finish_delay : std_logic := '0';
  signal counter : std_logic_vector(23 downto 0) := (others => '0');

begin  -- behave

 ---------------------------------------------------------------------------
  -- purpose: Linear Feedback Shift Register of 32 bit in order to generate
  --          pseydorandom signal (Y = z^32 + z^22 + z^2 + z + 1 )
  -- type   : sequential
  -- inputs : clk, reset, finish_delay, rand_temp, temp
  -- outputs: rand_temp, temp
 ---------------------------------------------------------------------------
  shift_register: process (clk, reset, finish_delay, rand_temp, temp)
  begin  -- process shift_register
    if reset = '0' then                 -- asynchronous reset (active low)
      rand_temp <= (31 => '1', others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      if finish_delay = '1' then
        temp <= rand_temp(31) xnor rand_temp(21) xnor rand_temp(1) xnor rand_temp(0);
        rand_temp(31 downto 1) <= rand_temp((30) downto 0);
        rand_temp(0) <= temp;
      end if;
    end if;
  end process shift_register;

 -----------------------------------------------------------------------------
  -- purpose: generating the pseudorandom timing delay
  -- type   : sequential
  -- inputs : clk, reset, rand_temp, counter, rand_temp
  -- outputs: finish_delay, counter
 -----------------------------------------------------------------------------
  delay: process (clk, reset, rand_temp, counter, rand_temp)
  begin  -- process delay
    if reset = '0' then                 -- asynchronous reset (active low)
      counter <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      counter <= counter + 1;
      finish_delay <= '0';
      if counter = rand_temp(23 downto 0) then
        counter <= ( others => '0');
        finish_delay <= '1';
      end if;
    end if;
  end process delay;

 -----------------------------------------------------------------------------
```

```
  -- purpose: waveform to the output pin in the component is enabled, else 'Z'
  -- type   : sequential
  -- inputs : clk, reset, enable, finish_delay
  -- outputs: wave_output
 ----------------------------------------------------------------------------
  waveform_out: process (clk, reset, enable, finish_delay)
  begin  -- process waveform_out
    if reset = '0' then            -- asynchronous reset (active low)
      wave_output <= 'Z';
    elsif clk'event and clk = '1' then  -- rising clock edge
      if enable = '1' then
        wave_output <= finish_delay;
      else
        wave_output <= 'Z';
      end if;
    end if;
  end process waveform_out;

end behave;
```

7.8.Component 6“Reading Data”

```
-------------------------------------------------------------------------------
 --                            COMPONENT 6
 --                          reading_data.vhd
 --                                ---
 --                        Michail Pligouroudis
 --                        M.Pligouroudis@gsi.de
 --                                ---
 --                       FPGA : MachXO2-1200HC
 --                       ATMEL: AT90CAN128
 --                       clock: f=10MHz, T=100ns
 -----------------------------------------------------------------------------

 -----------------------------------------------------------------------------
 -- component description:   This component is enabled only when you want to
 --                          send all the values of the registers to ATMEL in
 --                          order to read them from the pc. Every time it
 --                          sends 7 bytes of the values with the following
 --                          sequense (delay1 MSB, delay1 Middle, delay1 LSB,
 --                          delay2 MSB, delay2 Middle, delay2 LSB, Control Register)
 -----------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity reading_data is
  port (
    clk      : in  std_logic;           -- clock signal f=10MHz
    reset    : in  std_logic;           -- reset signal
    enable   : in  std_logic;           -- enable signal for the component
    byte_to_ATMEL_ready : out std_logic;       -- HIGH' when the value of the
                                               -- registers is ready to the output
    recieve_transmit_select : out std_logic;   -- output in order to select
                                               -- between recieve or transmmit
                                               -- (recieve -> 'HIGH', transmit -> 'LOW')
    data_to_ATMEL : out std_logic_vector(7 downto 0);   -- value of FPGA registers
                                                        -- in ordet to be send to ATMEL
    Control_Register_input: std_logic_vector(7 downto 0); -- databus for the value of
                                                            -- the Control Register
    delay_1_in : in std_logic_vector(23 downto 0);   -- input  for delay1 value
    delay_2_in : in std_logic_vector(23 downto 0));  -- input  for dalay2 value
end reading_data;

architecture behave of reading_data is

  type states is (decision, delay1_MSB, delay1_MIDDLE, delay1_LSB,delay2_MSB, delay2_MIDDLE, delay2_LSB,
Control_Register);
  signal present_state, next_state : states := decision;
  type states3 is (decision3, MSB_1, MIDDLE_1, LSB_1, MSB_2, MIDDLE_2, LSB_2, Control_Register3);
  signal present_state3, next_state3 : states3 := decision3;
  type states2 is (decision2, enable_low);
  signal present_state2, next_state2 : states2 := decision2;
  signal counter, counter2, counter3 : std_logic_vector(19 downto 0) := (others => '0');
  signal finish_delay,finish_counting,finish_counting3 : std_logic := '0';
  signal start_delay1, start_delay2, start_counting,start_counting3, start_counting4 : std_logic := '0';

begin  -- behave

 -----------------------------------------------------------------------------
  -- purpose: selecting the state of the state machine in order to send
  --          seperated the value of each register to ATMEL
  -- type   : combinatorial
  -- inputs : present_state, enable, finish_delay, delay_1_in, delay_2_in, Control_Register_input
  -- outputs: start_delay1, start_delay2, next_state, data_to_ATMEL
 -----------------------------------------------------------------------------
  state_selection: process (present_state, enable, finish_delay, delay_1_in, delay_2_in, Control_Register_input)
  begin  -- process state_selection
    start_delay1 <= '0';
    start_delay2 <= '0';

    case present_state is
      when decision =>
        if enable = '1' then
          next_state <= delay1_MSB;
          start_delay1 <= '1';
          data_to_ATMEL <= delay_1_in(23 downto 16);
        else
          next_state <= decision;
          data_to_ATMEL <= "00000000";
        end if;

      when delay1_MSB =>
        if finish_delay = '1' then
          next_state <= delay1_MIDDLE;
```

```
          start_delay1 <= '0';
          start_delay2 <= '1';
          data_to_ATMEL <= delay_1_in(15 downto 8);
        else
          next_state <= delay1_MSB;
          start_delay1 <= '1';
          data_to_ATMEL <= delay_1_in(23 downto 16);
        end if;

      when delay1_MIDDLE =>
        if finish_delay = '1' then
          next_state <= states'succ(present_state);
          start_delay2 <= '1';
          data_to_ATMEL <= delay_1_in(7 downto 0);
        else
          next_state <= present_state;
          start_delay2 <= '1';
          data_to_ATMEL <= delay_1_in(15 downto 8);
        end if;

      when delay1_LSB =>
        if finish_delay = '1' then
          next_state <= states'succ(present_state);
          start_delay2 <= '1';
          data_to_ATMEL <= delay_2_in(23 downto 16);
        else
          next_state <= present_state;
          start_delay2 <= '1';
          data_to_ATMEL <= delay_1_in(7 downto 0);
        end if;

      when delay2_MSB =>
        if finish_delay = '1' then
          next_state <= states'succ(present_state);
          start_delay2 <= '1';
          data_to_ATMEL <= delay_2_in(15 downto 8);
        else
          next_state <= present_state;
          start_delay2 <= '1';
          data_to_ATMEL <= delay_2_in(23 downto 16);
        end if;

      when delay2_MIDDLE =>
        if finish_delay = '1' then
          next_state <= states'succ(present_state);
          start_delay2 <= '1';
          data_to_ATMEL <= delay_2_in(7 downto 0);
        else
          next_state <= present_state;
          start_delay2 <= '1';
          data_to_ATMEL <= delay_2_in(15 downto 8);
        end if;

      when delay2_LSB =>
        if finish_delay = '1' then
          next_state <= states'succ(present_state);
          start_delay2 <= '1';
          data_to_ATMEL <= Control_Register_input;
        else
          next_state <= delay2_LSB;
          start_delay2 <= '1';
          data_to_ATMEL <= delay_2_in(7 downto 0);
        end if;

      when Control_Register =>
        if finish_delay = '1' then
          next_state <= decision;
          start_delay2 <= '0';
          data_to_ATMEL <= "00000000";
        else
          next_state <= Control_Register;
          start_delay2 <= '1';
          data_to_ATMEL <= Control_Register_input;
        end if;

      when others => null;
    end case;
  end process state_selection;

 -------------------------------------------------------------------------------
  -- purpose: sychronus passing from present state to next state
  -- type   : sequential
  -- inputs : clk, reset, next_step, finish_delay
  -- outputs: present_state
 -------------------------------------------------------------------------------
  steps: process (clk, reset, next_state)
  begin  -- process next_step
    if reset = '0' then                 -- asynchronous reset (active low)
      present_state <= decision;
```

```
    elsif clk'event and clk = '1' then  -- rising clock edge
      present_state <= next_state;
    end if;
  end process steps;

 ---------------------------------------------------------------------------
  -- purpose: generating dalays for correct synchronisation between component6
  --          and component0. Delay for the first byte  sending (24ums).
  --          Delay between each byte (16us)
  -- type   : sequential
  -- inputs : clk, reset, start_delay2, start_delay1, counter
  -- outputs: finish_delay, counter
 ------------------------------------------------------------------ -----------
  delays: process (clk, reset, start_delay2, start_delay1, counter)
  begin  -- process delays
    if reset = '0' then                 -- asynchronous reset (active low)
      counter <= (others => '0');
      finish_delay <= '1';
    elsif clk'event and clk = '1' then  -- rising clock edge
      if start_delay1 = '1' then
       if counter = X"000f0" then -- delay for the first byte sending (24us)
        counter <= (others => '0');
        finish_delay <= '1';
       else
        counter <= counter + 1;
        finish_delay <= '0';
       end if;
      elsif start_delay2 = '1' then
       if counter = X"000a0" then -- delay between each byte (16us)
        counter <= (others => '0');
        finish_delay <= '1';
       else
        counter <= counter + 1;
        finish_delay <= '0';
       end if;
      end if;
    end if;
  end process delays;

 ------------------------------------------------------------------------------
  -- purpose: selecting the state of the state machine in order to be
  --          synchronized with the component0 "FPGA-ATMEL-communication" if
  --          the system will recieve or transmitt data.
  -- type   : compinatorial
  -- inputs : present_state2, enable, finish_counting
  -- outputs: next_state2, start_counting, recieve_transmit_select
 ------------------------------------------------------------------------------
  state_selection2: process (present_state2, enable, finish_counting)
  begin  -- process state_selection

    case present_state2 is

      when decision2 =>
        if enable = '1' then
          next_state2 <=enable_low;
          start_counting <= '1';
          recieve_transmit_select <= '0';
        else
          next_state2 <= decision2;
          start_counting <= '0';
          recieve_transmit_select <= '1';
        end if;

      when enable_low =>
        if finish_counting = '1' then
          next_state2 <= decision2 ;
          start_counting <= '0';
          recieve_transmit_select <= '1';
        else
          next_state2 <= enable_low;
          recieve_transmit_select <= '0';
          start_counting <= '1';
        end if;

      when others => null;
    end case;
  end process state_selection2;

 ------------------------------------------------------------------------------
  -- purpose: sychronus passing from present state 2 to next state 2
  -- type   : sequential
  -- inputs : clk, reset, next_state2
  -- outputs: present_state2
 ------------------------------------------------------------------------------
  steps2: process (clk, reset, next_state2)
  begin  -- process next_step
    if reset = '0' then                 -- asynchronous reset (active low)
      present_state2 <= decision2;
    elsif clk'event and clk = '1' then  -- rising clock edge
```

```
      present_state2 <= next_state2;
    end if;
  end process steps2;

 ---------------------------------------------------------------------------
  -- purpose: generating delay for correct synchronisation during transmitting
  --          period (148us)
  -- type   : sequential
  -- inputs : clk, reset, start_counting, counter2
  -- outputs: finish_counting, counter2
 ------------------------------------------------------------------ -----------
  delays2: process (clk, reset, start_counting, counter2)
  begin  -- process delays
    if reset = '0' then                 -- asynchronous reset (active low)
      counter2 <= (others => '0');
      finish_counting <= '1';
    elsif clk'event and clk = '1' then  -- rising clock edge
      if start_counting = '1' then
        if counter2 = X"005C8"  then    -- delay for transmitting data (148us)
          counter2 <= (others => '0');
          finish_counting <= '1';
        else
          counter2 <= counter2 + 1;
          finish_counting <= '0';
        end if;
      end if;
    end if;
  end process delays2;

 ------------------------------------------------------------------------------
  -- purpose: selecting the state of the state machine in order to send signal when
  --          the value of each register is ready at the output of this component
  -- type   : combinatorial
  -- inputs : present_state3, enable, finish_counting3
  -- outputs: start_counting4, start_counting3, next_state3, byte_to_ATMEL_ready
 ------------------------------------------------------------------------------
  state_selection3: process (present_state3, enable, finish_counting3)
  begin  -- process state_selection
    start_counting3 <= '0';
    start_counting4 <= '0';

    case present_state3 is
      when decision3 =>
        if enable = '1' then
          next_state3 <= MSB_1;
          start_counting3 <= '1';
          byte_to_ATMEL_ready <= '0';
        else
          next_state3 <= decision3;
          byte_to_ATMEL_ready <= '0';
        end if;

      when MSB_1 =>
        if finish_counting3 = '1' then
          next_state3 <= MIDDLE_1;
          start_counting3 <= '0';
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '1';
        else
          next_state3 <= MSB_1;
          start_counting3 <= '1';
          byte_to_ATMEL_ready <= '0';
        end if;

      when MIDDLE_1 =>
        if finish_counting3 = '1' then
          next_state3 <= states3'succ(present_state3);
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '1';
        else
          next_state3 <= present_state3;
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '0';
        end if;

      when LSB_1 =>
        if finish_counting3 = '1' then
          next_state3 <= states3'succ(present_state3);
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '1';
        else
          next_state3 <= present_state3;
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '0';
        end if;

      when MSB_2 =>
        if finish_counting3 = '1' then
          next_state3 <= states3'succ(present_state3);
```

```
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '1';
        else
          next_state3 <= present_state3;
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '0';
        end if;

      when MIDDLE_2 =>
        if finish_counting3 = '1' then
          next_state3 <= states3'succ(present_state3);
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '1';
        else
          next_state3 <= present_state3;
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '0';
        end if;

      when LSB_2 =>
        if finish_counting3 = '1' then
          next_state3 <= states3'succ(present_state3);
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '1';
        else
          next_state3 <= LSB_2;
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '0';
        end if;

      when Control_Register3 =>
        if finish_counting3 = '1' then
          next_state3 <= decision3;
          start_counting4 <= '0';
          byte_to_ATMEL_ready <= '1';
        else
          next_state3 <= Control_Register3;
          start_counting4 <= '1';
          byte_to_ATMEL_ready <= '0';
        end if;

      when others => null;
    end case;
  end process state_selection3;

 ------------------------------------------------------------------------------
  -- purpose: sychronus passing from present state3 to next state3
  -- type   : sequential
  -- inputs : clk, reset, next_state3
  -- outputs: present_state3
 ------------------------------------------------------------------------------
  steps3: process (clk, reset, next_state3)
  begin  -- process next_step
    if reset = '0' then                -- asynchronous reset (active low)
      present_state3 <= decision3;
    elsif clk'event and clk = '1' then  -- rising clock edge
      present_state3 <= next_state3;
    end if;
  end process steps3;

 ---------------------------------------------------------------------------
  -- purpose: generating dalays for correct synchronisation Delay for the first
  --          byte  sending when it is ready to output (20us). Delay between each
  --          byte when it is ready to output (16ums)
  -- type   : sequential
  -- inputs : clk, reset
  -- outputs: finish_delay, counter
 ------------------------------------------------------------------ -----------
  delays3: process (clk, reset, start_counting3, start_counting4)
  begin  -- process delays
    if reset = '0' then                -- asynchronous reset (active low)
      counter3 <= (others => '0');
      finish_counting3 <= '1';
    elsif clk'event and clk = '1' then  -- rising clock edge
      if start_counting3 = '1' then
        if counter3 = X"000c8" then -- delay in order to start the 1st byte (20us)
          counter3 <= (others => '0');
          finish_counting3 <= '1';
        else
          counter3 <= counter3 + 1;
          finish_counting3 <= '0';
        end if;
      elsif start_counting4 = '1' then
        if counter3 = X"000a0" then -- delay between every byte (16us)
          counter3 <= (others => '0');
          finish_counting3 <= '1';
        else
          counter3 <= counter3 + 1;
          finish_counting3 <= '0';
```

```
        end if;
      end if;
    end if;
  end process delays3;

end behave;
```

7.9.Component 7 “PLL clock multiplier”

```
-- VHDL netlist generated by SCUBA Diamond_1.4_Production (87)
-- Module  Version: 5.2
--/opt/lattice/diamond/1.4/ispfpga/bin/lin/scuba -w -n PLL -lang vhdl -synth synplify -arch xo2c00 -type pll -fin
10 -fclkop 100 -fclkop_tol 0.0 -trimp 0 -phasep 0 -trimp_r -phase_cntl STATIC -fb_mode 1 -e

-- Tue Jun 12 16:47:05 2012

library IEEE;
use IEEE.std_logic_1164.all;
-- synopsys translate_off
library MACHXO2;
use MACHXO2.components.all;
-- synopsys translate_on

entity PLL is
    port (
        CLKI: in  std_logic;
        CLKOP: out  std_logic);
 attribute dont_touch : boolean;
 attribute dont_touch of PLL : entity is true;
end PLL;

architecture Structure of PLL is

    -- internal signal declarations
    signal LOCK: std_logic;
    signal CLKOP_t: std_logic;
    signal scuba_vlo: std_logic;

    -- local component declarations
    component VLO
        port (Z: out  std_logic);
    end component;
    component EHXPLLJ
        generic (INTFB_WAKE : in String; DDRST_ENA : in String;
                DCRST_ENA : in String; MRST_ENA : in String;
                PLLRST_ENA : in String; DPHASE_SOURCE : in String;
                OUTDIVIDER_MUXD2 : in String;
                OUTDIVIDER_MUXC2 : in String;
                OUTDIVIDER_MUXB2 : in String;
                OUTDIVIDER_MUXA2 : in String;
                PREDIVIDER_MUXD1 : in Integer;
                PREDIVIDER_MUXC1 : in Integer;
                PREDIVIDER_MUXB1 : in Integer;
                PREDIVIDER_MUXA1 : in Integer; PLL_USE_WB : in String;
                PLL_LOCK_MODE : in Integer;
                CLKOS_TRIM_DELAY : in Integer;
                CLKOS_TRIM_POL : in String;
                CLKOP_TRIM_DELAY : in Integer;
                CLKOP_TRIM_POL : in String; FRACN_DIV : in Integer;
                FRACN_ENABLE : in String; FEEDBK_PATH : in String;
                CLKOS3_FPHASE : in Integer; CLKOS2_FPHASE : in Integer;
                CLKOS_FPHASE : in Integer; CLKOP_FPHASE : in Integer;
                CLKOS3_CPHASE : in Integer; CLKOS2_CPHASE : in Integer;
                CLKOS_CPHASE : in Integer; CLKOP_CPHASE : in Integer;
                VCO_BYPASS_D0 : in String; VCO_BYPASS_C0 : in String;
                VCO_BYPASS_B0 : in String; VCO_BYPASS_A0 : in String;
                CLKOS3_ENABLE : in String; CLKOS2_ENABLE : in String;
                CLKOS_ENABLE : in String; CLKOP_ENABLE : in String;
                CLKOS3_DIV : in Integer; CLKOS2_DIV : in Integer;
                CLKOS_DIV : in Integer; CLKOP_DIV : in Integer;
                CLKFB_DIV : in Integer; CLKI_DIV : in Integer);
        port (CLKI: in  std_logic; CLKFB: in  std_logic;
            PHASESEL1: in  std_logic; PHASESEL0: in  std_logic;
            PHASEDIR: in  std_logic; PHASESTEP: in  std_logic;
            LOADREG: in  std_logic; STDBY: in  std_logic;
            PLLWAKESYNC: in  std_logic; RST: in  std_logic;
            RESETM: in  std_logic; RESETC: in  std_logic;
            RESETD: in  std_logic; ENCLKOP: in  std_logic;
            ENCLKOS: in  std_logic; ENCLKOS2: in  std_logic;
            ENCLKOS3: in  std_logic; PLLCLK: in  std_logic;
            PLLRST: in  std_logic; PLLSTB: in  std_logic;
            PLLWE: in  std_logic; PLLADDR4: in  std_logic;
            PLLADDR3: in  std_logic; PLLADDR2: in  std_logic;
            PLLADDR1: in  std_logic; PLLADDR0: in  std_logic;
            PLLDATI7: in  std_logic; PLLDATI6: in  std_logic;
            PLLDATI5: in  std_logic; PLLDATI4: in  std_logic;
            PLLDATI3: in  std_logic; PLLDATI2: in  std_logic;
            PLLDATI1: in  std_logic; PLLDATI0: in  std_logic;
            CLKOP: out  std_logic; CLKOS: out  std_logic;
            CLKOS2: out  std_logic; CLKOS3: out  std_logic;
            LOCK: out  std_logic; INTLOCK: out  std_logic;
```

```
            REFCLK: out  std_logic; CLKINTFB: out  std_logic;
            DPHSRC: out  std_logic; PLLACK: out  std_logic;
            PLLDATO7: out  std_logic; PLLDATO6: out  std_logic;
            PLLDATO5: out  std_logic; PLLDATO4: out  std_logic;
            PLLDATO3: out  std_logic; PLLDATO2: out  std_logic;
            PLLDATO1: out  std_logic; PLLDATO0: out  std_logic);
    end component;
    attribute STDBY_ENABLE : string;
    attribute FREQUENCY_PIN_CLKOP : string;
    attribute FREQUENCY_PIN_CLKI : string;
    attribute ICP_CURRENT : string;
    attribute LPF_RESISTOR : string;
    attribute STDBY_ENABLE of PLLInst_0 : label is "DISABLED";
    attribute FREQUENCY_PIN_CLKOP of PLLInst_0 : label is "100.000000";
    attribute FREQUENCY_PIN_CLKI of PLLInst_0 : label is "10.000000";
    attribute ICP_CURRENT of PLLInst_0 : label is "9";
    attribute LPF_RESISTOR of PLLInst_0 : label is "8";
    attribute syn_keep : boolean;
    attribute syn_noprune : boolean;
    attribute syn_noprune of Structure : architecture is true;

begin
    -- component instantiation statements
    scuba_vlo_inst: VLO
        port map (Z=>scuba_vlo);

    PLLInst_0: EHXPLLJ
        generic map (DDRST_ENA=> "DISABLED", DCRST_ENA=> "DISABLED",
        MRST_ENA=> "DISABLED", PLLRST_ENA=> "DISABLED", INTFB_WAKE=> "DISABLED",
        DPHASE_SOURCE=> "DISABLED", PLL_USE_WB=> "DISABLED",
        CLKOS3_FPHASE=>  0, CLKOS3_CPHASE=>  0, CLKOS2_FPHASE=>  0,
        CLKOS2_CPHASE=>  0, CLKOS_FPHASE=>  0, CLKOS_CPHASE=>  0,
        CLKOP_FPHASE=>  0, CLKOP_CPHASE=>  5, PLL_LOCK_MODE=>  0,
        CLKOS_TRIM_DELAY=>  0, CLKOS_TRIM_POL=> "FALLING",
        CLKOP_TRIM_DELAY=>  0, CLKOP_TRIM_POL=> "RISING", FRACN_DIV=>  0,
        FRACN_ENABLE=> "DISABLED", OUTDIVIDER_MUXD2=> "DIVD",
        PREDIVIDER_MUXD1=>  0, VCO_BYPASS_D0=> "DISABLED", CLKOS3_ENABLE=> "DISABLED",
        OUTDIVIDER_MUXC2=> "DIVC", PREDIVIDER_MUXC1=>  0, VCO_BYPASS_C0=> "DISABLED",
        CLKOS2_ENABLE=> "DISABLED", OUTDIVIDER_MUXB2=> "DIVB",
        PREDIVIDER_MUXB1=>  0, VCO_BYPASS_B0=> "DISABLED", CLKOS_ENABLE=> "DISABLED",
        OUTDIVIDER_MUXA2=> "DIVA", PREDIVIDER_MUXA1=>  0, VCO_BYPASS_A0=> "DISABLED",
        CLKOP_ENABLE=> "ENABLED", CLKOS3_DIV=>  1, CLKOS2_DIV=>  1,
        CLKOS_DIV=>  1, CLKOP_DIV=>  6, CLKFB_DIV=>  10, CLKI_DIV=>  1,
        FEEDBK_PATH=> "CLKOP")
        port map (CLKI=>CLKI, CLKFB=>CLKOP_t, PHASESEL1=>scuba_vlo,
            PHASESEL0=>scuba_vlo, PHASEDIR=>scuba_vlo,
            PHASESTEP=>scuba_vlo, LOADREG=>scuba_vlo, STDBY=>scuba_vlo,
            PLLWAKESYNC=>scuba_vlo, RST=>scuba_vlo, RESETM=>scuba_vlo,
            RESETC=>scuba_vlo, RESETD=>scuba_vlo, ENCLKOP=>scuba_vlo,
            ENCLKOS=>scuba_vlo, ENCLKOS2=>scuba_vlo, ENCLKOS3=>scuba_vlo,
            PLLCLK=>scuba_vlo, PLLRST=>scuba_vlo, PLLSTB=>scuba_vlo,
            PLLWE=>scuba_vlo, PLLADDR4=>scuba_vlo, PLLADDR3=>scuba_vlo,
            PLLADDR2=>scuba_vlo, PLLADDR1=>scuba_vlo,
            PLLADDR0=>scuba_vlo, PLLDATI7=>scuba_vlo,
            PLLDATI6=>scuba_vlo, PLLDATI5=>scuba_vlo,
            PLLDATI4=>scuba_vlo, PLLDATI3=>scuba_vlo,
            PLLDATI2=>scuba_vlo, PLLDATI1=>scuba_vlo,
            PLLDATI0=>scuba_vlo, CLKOP=>CLKOP_t, CLKOS=>open,
            CLKOS2=>open, CLKOS3=>open, LOCK=>LOCK, INTLOCK=>open,
            REFCLK=>open, CLKINTFB=>open, DPHSRC=>open, PLLACK=>open,
            PLLDATO7=>open, PLLDATO6=>open, PLLDATO5=>open,
            PLLDATO4=>open, PLLDATO3=>open, PLLDATO2=>open,
            PLLDATO1=>open, PLLDATO0=>open);

    CLKOP <= CLKOP_t;
end Structure;

-- synopsys translate_off
library MACHXO2;
configuration Structure_CON of PLL is
    for Structure
        for all:VLO use entity MACHXO2.VLO(V); end for;
        for all:EHXPLLJ use entity MACHXO2.EHXPLLJ(V); end for;
    end for;
end Structure_CON;

-- synopsys translate_on
```

8. Appendix B (ATMEL 'C' source code)

8.1.Header file "Waveform Generator Registers.h"

```
#ifndef WAVEFORM_GENERATOR_REGISTERS_H
#define WAVEFORM_GENERATOR_REGISTERS_H
#include "api_define.h"
#include "api_global.h"

#include "api_debug.h"
#include "api.h"

#define UCSR1C_register_of_ATMEL_address 0xCA
#define UBRR1L_register_of_ATMEL_address 0xCC
#define UBRR1H_register_of_ATMEL_address 0xCD
#define UCSR1B_register_of_ATMEL_address 0xC9
#define UCSR1A_register_of_ATMEL_address 0xC8
#define UDR1_register_of_ATMEL_address   0xCE

#define UCSR1C_register_of_ATMEL_value 0x06
#define UBRR1L_register_of_ATMEL_value 0x00
#define UBRR1H_register_of_ATMEL_value 0x00
#define UCSR1B_register_of_ATMEL_value_transmit 0x08
#define UCSR1B_register_of_ATMEL_value_receive 0x10
#define UCSR1A_register_of_ATMEL_value 0x22

#define CONTROL_REGISTER       0x00
#define DELAY1_MSB_REGISTER    0x01
#define DELAY1_MIDDLE_REGISTER 0x02
#define DELAY1_LSB_REGISTER    0x03
#define DELAY2_MSB_REGISTER    0x04
#define DELAY2_MIDDLE_REGISTER 0x05
#define DELAY2_LSB_REGISTER    0x06
#define SHOW_ALL               0xFF

#define DEBUG 0x00
#define RESET 0x40
#define READ  0xa0
#define PSEUDORANDOM 0x81
#define SQUARE 0x82
#define PULSE 0x83
#define PSEUDORAND_TIME_PULSE 0x84

enum steps{
   step1=0,
   step2,
   step3,
   step4,
   step5,
   step6,
   step7,
   step8,
   step9,
   step10,
   step11,
   step12
};

enum registerIndex {
         DELAY1_MSB_REGISTER_INDEX = 0,
         DELAY1_MIDDLE_REGISTER_INDEX,
         DELAY1_LSB_REGISTER_INDEX,
         DELAY2_MSB_REGISTER_INDEX,
         DELAY2_MIDDLE_REGISTER_INDEX,
         DELAY2_LSB_REGISTER_INDEX,
         CONTROL_REGISTER_INDEX,
         MAX_REGISTER_INDEX
};

void waveformGeneratorReadRegister( struct uartStruct *ptr_uartStruct ); /*this function reads the status of a
register with the of his adress */
void waveformGeneratorWriteRegister( struct uartStruct *ptr_uartStruct );/* this function writes a value the
status in the register with the of his adress */
void waveformGeneratorPrintSingleRegister( uint8_t registerId, uint8_t registers[], uint8_t size );
void waveformGeneratorDeclareUARTtoSendData(void);
void waveformGeneratorDeclareUARTtoReceiveData(void);

#endif
```

8.2. C file “Waveform Generator Registers.c”

```
/*
 * Author: Michail Pligouroudis 30/05/2012
 * modified (heavily rather rebuild): Peter Zumbruch
 */
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

#include "read_write_register.h"
#include "waveform_generator_registers.h"

#include "adc.h"
#include "api.h"
#include "api_define.h"
#include "api_global.h"

#include "api_debug.h"


void waveformGeneratorWriteRegister(struct uartStruct *ptr_uartStruct)
{
        uint8_t status = 0;
        uint8_t waveformGeneratorRegisterAddress, value;

        switch(ptr_uartStruct->number_of_arguments)
        {
                case 0:
                case 1:
                        CommunicationError(ERRG, -1, 1, PSTR("too few arguments"), -1000);
                        status = 1;
                        break;
                default:
                        break;
        }

        if ( 0 != status )
        {
                return;
        }

        waveformGeneratorRegisterAddress = ptr_uartStruct->Uart_Message_ID;
        value = ptr_uartStruct->Uart_Mask;

        //declare UART in order to send data with baud=1,025Mbps
        waveformGeneratorDeclareUARTtoSendData();

        //sending your new data
        writeInto8bitRegister(UDR1_register_of_ATMEL_address, waveformGeneratorRegisterAddress);
        writeInto8bitRegister(UDR1_register_of_ATMEL_address, value);

        // print response
        clearString(uart_message_string, BUFFER_SIZE);
        createReceiveHeader(ptr_uartStruct, uart_message_string, BUFFER_SIZE);
        switch (waveformGeneratorRegisterAddress)
        {
                case CONTROL_REGISTER:
                {
                        switch (value)
                        {
                                case DEBUG:
                                        snprintf_P(uart_message_string, BUFFER_SIZE - 1, PSTR("%s DEBUG"),
                                                        uart_message_string);
                                        break;
                                case RESET:
                                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                                        PSTR("%sRESET STATUS"), uart_message_string);
                                        break;
                                case PSEUDORANDOM:
                                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                                        PSTR("%sPSEUDORANDOM WAVEFORM OUTPUT"),
uart_message_string);
                                        break;
                                case SQUARE:
                                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                                        PSTR("%sSQUARE WAVEFORM OUTPUT"),
uart_message_string);
                                        break;
                                case PULSE:
                                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                                        PSTR("%sPULSE WAVEFORM OUTPUT"), uart_message_string);
                                        break;
```

```
                                case PSEUDORAND_TIME_PULSE:
                                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                                        PSTR("%sPSEUDORANDOM TIME PULSE(100ns) WAVEFORM
OUTPUT"),  uart_message_string);
                                        break;
                                default:
                                        CommunicationError(ERRG, -1, 1, PSTR("invalid argument"), -1000);
                                        status = 1;
                                        break;
                        }
                        break;
                }
                case DELAY1_MSB_REGISTER:
                {
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%sDelay1 MSB is written"),
                                        uart_message_string);
                        break;
                }
                case DELAY1_MIDDLE_REGISTER:
                {
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%sDelay1 MIDDLE is written"),
                                        uart_message_string);
                        break;
                }
                case DELAY1_LSB_REGISTER:
                {
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%sDelay1 LSB is written"),
                                        uart_message_string);
                        break;
                }
                case DELAY2_MSB_REGISTER:
                {
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%sDelay2 MSB is written"),
                                        uart_message_string);
                        break;
                }
                case DELAY2_MIDDLE_REGISTER:
                {
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%sDelay2 MIDDLE is written"),
                                        uart_message_string);
                        break;
                }
                case DELAY2_LSB_REGISTER:
                {
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%sDelay2 LSB is written"),
                                        uart_message_string);
                        break;
                }
                default:
                        CommunicationError(ERRG, -1, 1, PSTR("invalid address"), -1000);
                        status = 1;
                        break;
        }
        if ( 0 == status)
        {
                UART0_Send_Message_String(NULL, 0);
        }
        return;

} //END of write_double_Register function

void waveformGeneratorReadRegister(struct uartStruct *ptr_uartStruct)
{
        int step = 0;
        uint8_t registers[7];

        //declare UART in order to send data with baud=1,025Mbps
        waveformGeneratorDeclareUARTtoSendData();

        //control register address
        writeInto8bitRegister(  UDR1_register_of_ATMEL_address, CONTROL_REGISTER); //0x00

        //control register in read status
        writeInto8bitRegister(  UDR1_register_of_ATMEL_address, READ); //0xA0

        //declare UART in order to receive data with baud=1,025Mbps
        waveformGeneratorDeclareUARTtoReceiveData();

        _delay_us(32); //delay in order to synchronize the first recieving byte

        // saving value of the registers from FPGA at the table "registers[]"
        for (step = 0; step < MAX_REGISTER_INDEX; step++)
        {
```

```
                registers[step] = readFrom8bitRegister((uint8_t) UDR1_register_of_ATMEL_address & 0xFF );
                _delay_us(16); // delay between each byte
        } //end for loop

        //declare UART in order to send data with baud=1,025Mbps (twice on purpose!)
        waveformGeneratorDeclareUARTtoSendData();
        waveformGeneratorDeclareUARTtoSendData();


        // print results
        switch(ptr_uartStruct->number_of_arguments)
        {
                case 0:
                        waveformGeneratorPrintSingleRegister( ( uint8_t) SHOW_ALL ,

registers, sizeof(registers)/sizeof(uint8_t) );
                        break;
                default:
                        waveformGeneratorPrintSingleRegister( ( uint8_t) (ptr_uartStruct->Uart_Message_ID &
0xFF),

registers, sizeof(registers)/sizeof(uint8_t) );
                        break;
        }
} //end of waveformGeneratorReadRegister function

void waveformGeneratorPrintSingleRegister( uint8_t registerId, uint8_t registers[], uint8_t size )
{
        uint8_t status = 0 ;
        clearString(uart_message_string, BUFFER_SIZE);
        createReceiveHeader(ptr_uartStruct, uart_message_string, BUFFER_SIZE);
        switch (registerId)
        {
                case DELAY1_MSB_REGISTER:
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%s%x %x Delay1 MSB "), uart_message_string,
                                        DELAY1_MSB_REGISTER, registers[DELAY1_MSB_REGISTER_INDEX]);
                        break;
                case DELAY1_MIDDLE_REGISTER:
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%s%x %x Delay1 MIDDLE "), uart_message_string,
                                        DELAY1_MIDDLE_REGISTER, registers[DELAY1_MIDDLE_REGISTER_INDEX]);
                        break;
                case DELAY1_LSB_REGISTER:
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%s%x %x Delay1 LSB "), uart_message_string,
                                        DELAY1_LSB_REGISTER, registers[DELAY1_LSB_REGISTER_INDEX]);
                        break;
                case DELAY2_MSB_REGISTER:
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%s%x %x Delay2 MSB "), uart_message_string,
                                        DELAY2_MSB_REGISTER, registers[DELAY2_MSB_REGISTER_INDEX]);
                        break;
                case DELAY2_MIDDLE_REGISTER:
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%s%x %x Delay2 MIDDLE "), uart_message_string,
                                        DELAY2_MIDDLE_REGISTER, registers[DELAY2_MIDDLE_REGISTER_INDEX]);
                        break;
                case DELAY2_LSB_REGISTER:
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%s%x %x Delay2 LSB "), uart_message_string,
                                        DELAY2_LSB_REGISTER, registers[DELAY2_LSB_REGISTER_INDEX]);
                        break;
                case CONTROL_REGISTER:
                        snprintf_P(uart_message_string, BUFFER_SIZE - 1,
                                        PSTR("%s%x %x Control Register "), uart_message_string,
                                        CONTROL_REGISTER, registers[CONTROL_REGISTER_INDEX]);
                        break;
                case SHOW_ALL: /*recursive call*/
                        waveformGeneratorPrintSingleRegister(DELAY1_MSB_REGISTER,    registers, size);
                        waveformGeneratorPrintSingleRegister(DELAY1_MIDDLE_REGISTER, registers, size);
                        waveformGeneratorPrintSingleRegister(DELAY1_LSB_REGISTER,    registers, size);
                        waveformGeneratorPrintSingleRegister(DELAY2_MSB_REGISTER,    registers, size);
                        waveformGeneratorPrintSingleRegister(DELAY2_MIDDLE_REGISTER, registers, size);
                        waveformGeneratorPrintSingleRegister(DELAY2_LSB_REGISTER,    registers, size);
                        waveformGeneratorPrintSingleRegister(CONTROL_REGISTER,       registers, size);
                        break;
                default:
                        CommunicationError(ERRG, -1, 1, PSTR("wrong register address"), -1000);
                        status = 1;
                        break;
        }
        if ( 0 == status)
        {
                UART0_Send_Message_String(NULL, 0);
        }
    return;
}
```

```
void waveformGeneratorDeclareUARTtoSendData(void)
{
        //declare UART in order to send data with baud=1,025Mbps
        writeInto8bitRegister(UCSR1C_register_of_ATMEL_address, UCSR1C_register_of_ATMEL_value);
        writeInto8bitRegister(UBRR1L_register_of_ATMEL_address, UBRR1L_register_of_ATMEL_value);
        writeInto8bitRegister(UBRR1H_register_of_ATMEL_address, UBRR1H_register_of_ATMEL_value);
        writeInto8bitRegister(UCSR1B_register_of_ATMEL_address, UCSR1B_register_of_ATMEL_value_transmit);
        writeInto8bitRegister(UCSR1A_register_of_ATMEL_address, UCSR1A_register_of_ATMEL_value);
}

void waveformGeneratorDeclareUARTtoReceiveData(void)
{
        //declare UART in order to receive data with baud=1,025Mbps
        writeInto8bitRegister(UCSR1C_register_of_ATMEL_address, UCSR1C_register_of_ATMEL_value);
        writeInto8bitRegister(UBRR1L_register_of_ATMEL_address, UBRR1L_register_of_ATMEL_value);
        writeInto8bitRegister(UBRR1H_register_of_ATMEL_address, UBRR1H_register_of_ATMEL_value);
        writeInto8bitRegister(UCSR1A_register_of_ATMEL_address, UCSR1A_register_of_ATMEL_value);
        writeInto8bitRegister(UCSR1B_register_of_ATMEL_address, UCSR1B_register_of_ATMEL_value_receive);
}
```

9. Acknowledgments

Firstly I would like to thank GSI and exactly the department of Experiment Electronics for their hospitality, during my internship. I would also like to thank my tutor Dr. Michael Traxler and Dr. Peter Zumbruch for their guidance and their help.

10. References

[1] "VHDL for programmable logic", Kevin Skahill , Cypress Semiconductor (1996)

[2] Epics Hadcon2 http://wiki.gsi.de/Epics/HadCon2?sortcol=1&table=11&up=0

[3] Lattice Semiconductor http://www.latticesemi.com/documents/38834.pdf

[4] Pseudorandom Generator http://en.wikipedia.org/wiki/Linear_feedback_shift_register

[5] http://www.xilinx.com/support/documentation/application_notes/xapp210.pdf

[6] "Σχεδίαση Ψηφιακών Συστημάτων με γλώσσα VHDL", S. Brown - Z. Vranesic, Tziola (2000)