VHDL Project

Range Measurement System

Digital System Design with VHDL (ET062G)

Mid Sweden University, Sundsvall

Student:
Jørgen Ryther Hoem

June 13, 2016

NTNU – Trondheim
Norwegian University of
Science and Technology

Abstract

This report describes the implementation, testing and results of a range measurement system using FPGA and VHDL. Besides the FPGA, the system is built using hardware such as a range sensor, VGA display, LEDs and 7-segement display. The system also makes use of hardware such as switches and push buttons for user interaction. The report is focused on the software side of the system in terms of VHDL and simulation and shows that the final system is working accordingly to the project instructions.
The project is part of the «Digital System Design with VHDL» course (ET062G) at Mid Sweden University in Sundsvall, Sweden.

Contents

Abbreviations

FPGA	Field Programmable Gate Array
LUT	LookUp Table
FF	Flip-Flop
RAM	Random Access Memory
BRAM	Block RAM
LUTRAM	LookUp Table RAM (Distributed RAM)
FIFO	First In, First Out
MMCM	Mixed-Mode Clock Manager
BCD	Binary-Coded Decimal
PLL	Phase-Locked Loop

1 Introduction

The main objective of this project is to learn how to interact with different kind of hardware using FPGA and VHDL. And gain experience in how to simulate, verify and test code as well as how to solve problems with VHDL.

The goal is to create a range measurement system using an ultrasonic range sensor, represent the data as a plot in a user-friendly manner on a VGA display, where the plot is surrounded by the flag of the student's home country.

For the sake of the length of this report, the report will assume that the reader has a basic understanding of VHDL, FPGA, VGA and Ultrasonic Range Sensors. This means that the report will not go in-depth on the hardware side of the project but instead focus on the software side in regards of implementation, simulation and verification.

The report will begin with a brief description of the required hardware. It will then present an overview of the design, and describe the code structure and functionality of the system.

All of the modules created and used in this project will be described before the simulations of the system is presented. The last part of the report will show how the testing of the system have been done, and then present the results in regards of accuracy and resource usage. The system as a whole will then be discussed before a short conclusion is given.

List of figures, tables and references can be found at the end of the report.

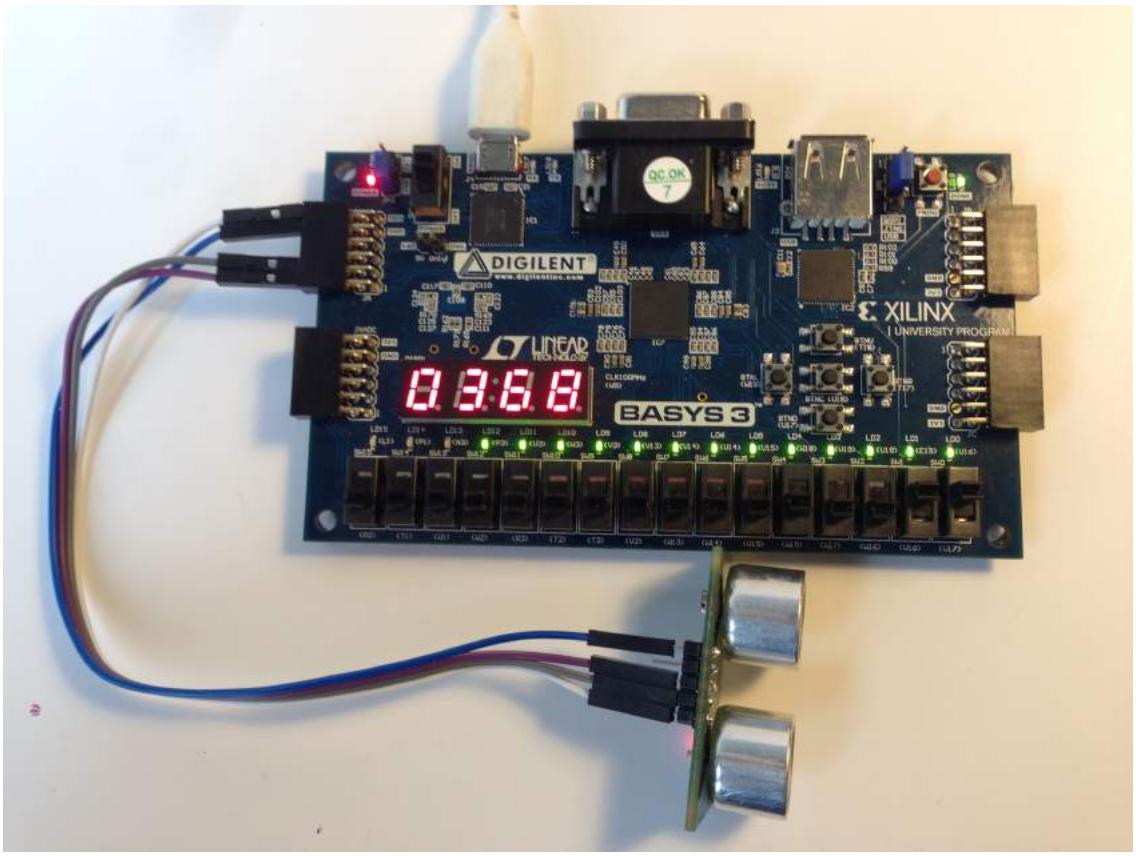


Figure 1: FPGA with the SRF05 connected

2 Hardware

This section will give a brief description of the hardware that is both required and used in this project.

2.1 The FPGA Development Board

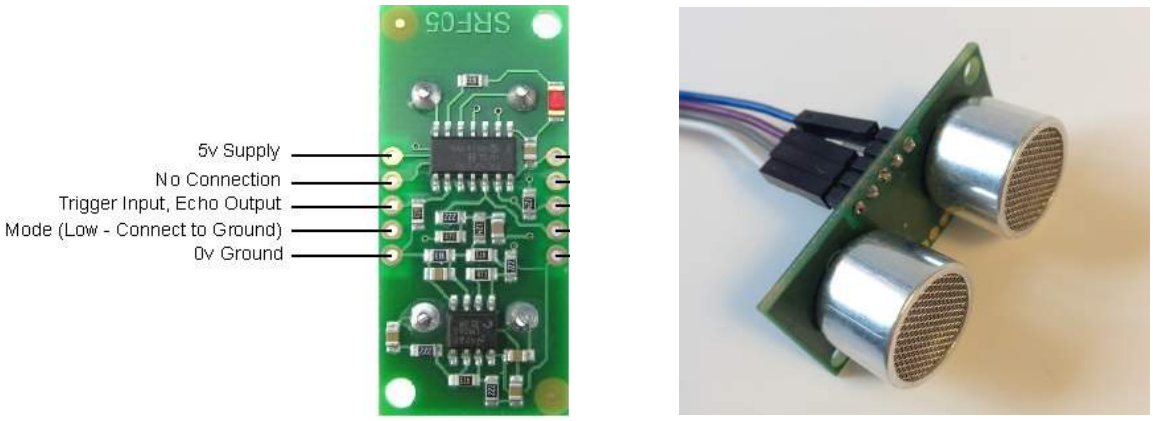
The FPGA kit used in this project is the Basys 3 Artix-7 FPGA Trainer Board (Figure 1). This board contains the Xilinx Series-7 chip XC7A35T-1CPG236. Features that are most relevant to the project is listed in Table 1.

Basys 3 Artix-7 FPGA Trainer Board
Xilinx Artix-7 FPGA: XC7A35T-1CPG236
33,280 logic cells in 5200 slices
USB-powered (micro-B)
1,800 Kbits of fast block RAM
12-bit VGA output
16 user switches
16 user LEDs
5 user pushbuttons
4-digit 7-segment display

Table 1: Basys3 FPGA features

One of the goals have been to use as many of the boards peripherals as possible, such as the switches, LEDs, push buttons, 7-segment display and so forth.

2.2 Ultrasonic Range Sensor



(a) SRF05 pinout (b) SRF05

Figure 2: SRF05 ultrasonic range sensor

The range sensor used in this project is an ultrasonic range sensor called «SRF05». This sensor is used in mode 1 which means that separate pins controls the sending and receiving of the trigger- and echo pulse.

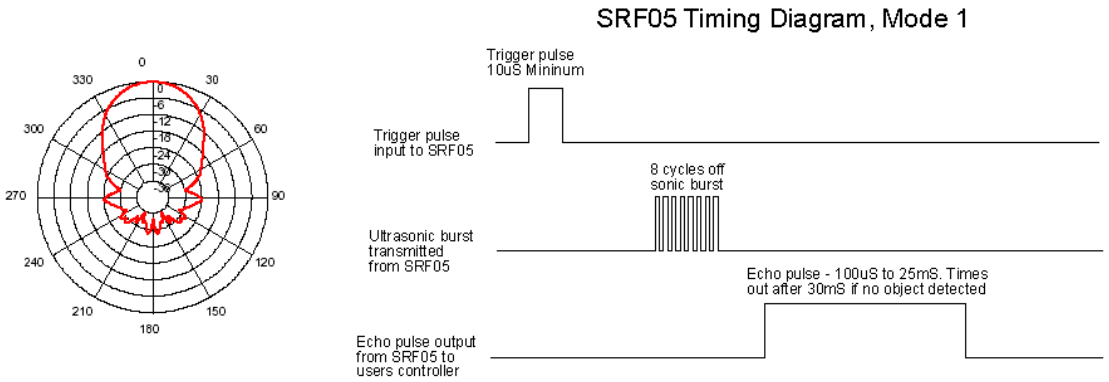


Figure 3: SRF05 beam profile and timing diagram

This sensor works by sending out an ultrasonic burst of 8 pulses at 40 kHz each time it receives a trigger pulse between 10 µs and 20 µs. The duration of the echo pulse will then represent the range. The range are then converted to centimeters by dividing the echo pulse length with 58.

Even though the SRF05 is rated for distances up to 400 cm (others claim 450 cm[1]), the timeout of the sensor is 30 ms which gives a *theoretical* range of up to 517 cm.

The sensor is rated for 5 VDC but have been supplied with 3.3 VDC directly from the FPGA board without any problems. This have been done to ease the process of testing.

2.3 VGA display

The FPGA provides an 12-bit VGA port which is used in this project. 12-bit VGA means that the RGB colors are represented with 4 bits each, hence RGB444.

General timing

Screen refresh rate	60 Hz
Vertical refresh	63.981042654028 kHz
Pixel freq.	108.0 MHz

Horizontal timing (line)

Polarity of horizontal sync pulse is positive.

Scanline part	Pixels	Time [µs]
Visible area	1280	11.851851851852
Front porch	48	0.44444444444444
Sync pulse	112	1.037037037037
Back porch	248	2.2962962962963
Whole line	1688	15.62962962963

Vertical timing (frame)

Polarity of vertical sync pulse is positive.

Frame part	Lines	Time [ms]
Visible area	1024	16.004740740741
Front porch	1	0.01562962962963
Sync pulse	3	0.046888888888889
Back porch	38	0.59392592592593
Whole frame	1066	16.661185185185

Figure 4: VGA timing data for 1280x1024 resolution[1]

The project is built using the VGA resolution 1280x1024 at a 60 Hz timing frequency. This requires a pixel clock of 108 MHz. Other key parameters used from Figure 4 is the vertical and horizontal front and back porch, and sync pulses.

[1] http://www.f15ijp.com/2012/09/arduino-ultrasonic-sensor-hc-sr04-or-hy-srf05/

3 Design and functionality

3.1 Design overview

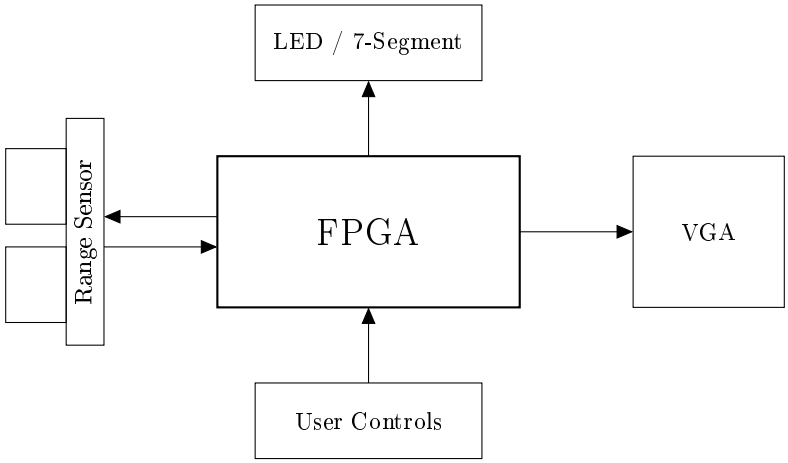


Figure 5: System overview, block diagram

Figure 5 show the basic overview of the system and how the different hardware is interacting and connected together.

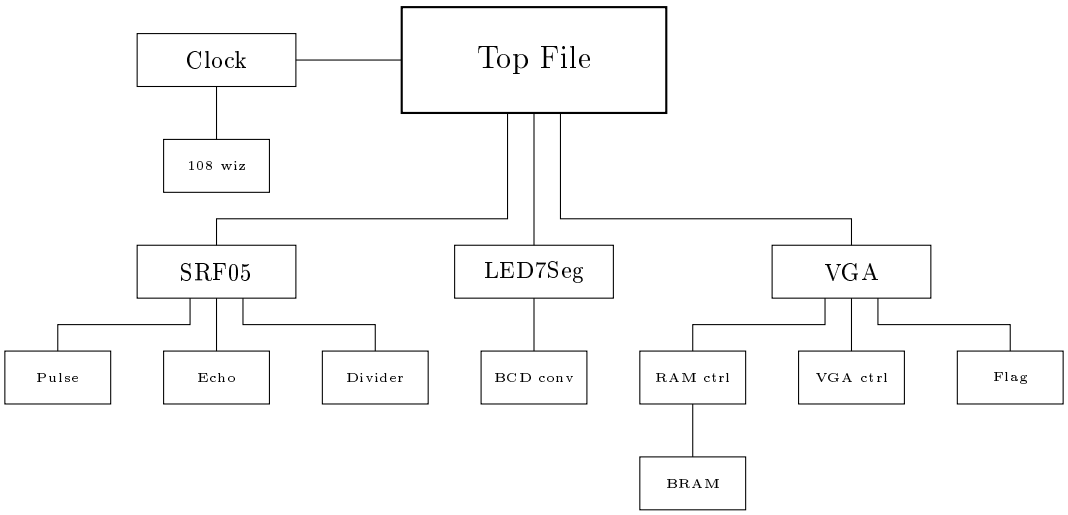


Figure 6: Design overview, block diagram

Figure 6 show how all of the VHDL components are structured and connected. One of the first experiences gained from this project was that many components can quickly become quite messy. Thus, one of the important aspects have been to have a structured and logical component hierarchy.

3.2 Code description

All of the code is developed in Vivado 2015.1 with emphasizes to create a modular system and using material learned during the course. The code is also written as an attempt to follow best practices and the Xilinx guidelines.

The system consist of a main top file that connects everything together. Subsystems, such as the VGA, SRF05 Sensor and so forth, are called cores and have the suffix `core` in the entity- and file name. A core can can be considered as a driver for that particular subsystem and can contain processes as well as other components to work. Independent components, usually used by cores, have the suffix `module` with exception of the Clock Generator.

Furthermore, all modules are written in lower case. General named signals have the suffix `in` or `out` to avoid confusion. For clarity, port and generic mapping is done using named association instead of positional association. Files that are dependent of other components are using entity instantiation [2] instead of component instantiation. This is to make a leaner and more readable code that is less error-prone and less time consuming to change during the development.

Individual testbenches have been created for the main top file and each of the cores. With exception of the RAM module which has a verification module instead of a dedicated testbench. Though it has been manually simulated. Additionally, an individual testbench component have been made to simulate a VGA display. This is a standalone testbench component that is used both with the top file and the VGA core.

All of the files in this project is written by the author of this report, with exception of the 108 MHz clock generator and the VGA controller.

3.3 Functionality

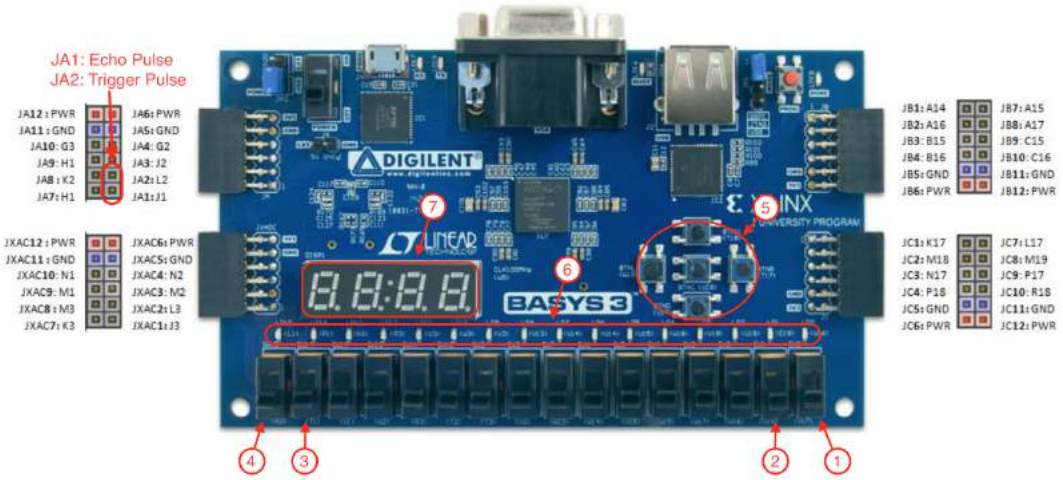


Figure 7: Board functionality map

1. Reset switch for the VGA (active low).
2. Enable switch for the range sensor (active high).
3. Switch for bisecting the sensor data displayed on the VGA. When on the full graph height is displayed, when off the height is divided by two.
4. Controls the data flow displayed on the VGA screen. When on the data is updated continuously, when off the data is only updated when new values occurs.
5. Control buttons, moves the graph to the desired location on the screen. The center button resets the position to default.
6. LED-meter that indicate the range from 0 to 450 cm, each LED represents 30 cm.
7. 7-segment display that shows the received sensor data in real time.

4 Modules

This section will give a brief description of each of the cores and modules used in this project. Modules marked with «extern» in the title is not created by the author of this project. It is recommended to read the source code for a more in-depth description of the functionality.

4.1 Clock Generator

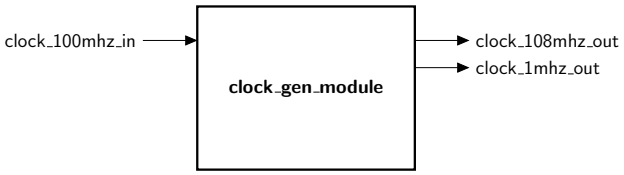


Figure 8: Entity of jrh_clock_gen_module.vhd

The clock generator was created to have one module that generate all of the necessary clock frequencies, 1 MHz and 108 MHz, for this project.

The 1 MHz clock, often referred to as the «slow clock» in this project, is made using a counter that downscale the 100 MHz input clock to 1 MHz. This clock drives all of the low speed components and functions such as the range sensor, RAM writing, LED-meter and the 7-segment display.

4.1.1 108 MHz clock (extern)

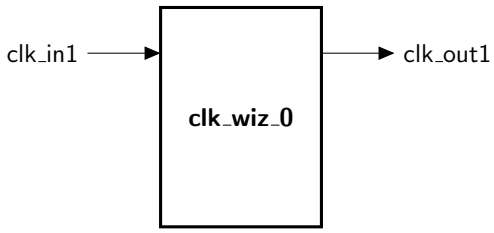


Figure 9: Entity of clk_wiz_0.vhd

The 108 MHz clock is only used for the VGA display and thus also RAM reading. This clock is driven by the Mixed-Mode Clock Manager (MMCM) that is provided by the Xilinx FPGA. The code for this module was given with the instructions for laboratory exercise 1 and appears to be generated from a clock wizard. It has not been modified and is used as is.

4.2 Range Sensor

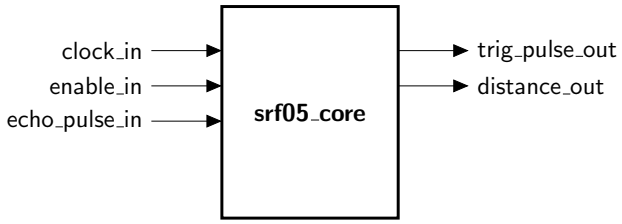


Figure 10: Entity of jrh_srf05_core.vhd

The SRF05 core is the driver for the SRF05 Range Sensor and is dependent of three modules: Pulse generator, Echo pulse analyzer and a divider.

4.2.1 Pulse Generator

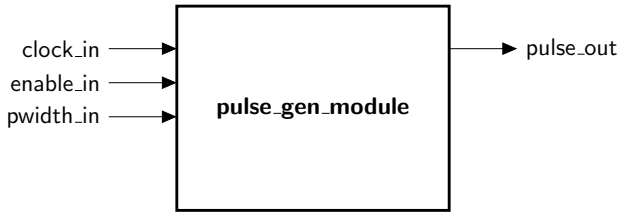


Figure 11: Entity of jrh_pulse_gen_module.vhd

The pulse generator module activates the range measurement by sending a trigger pulse to the sensor. The frequency of this pulse is 10 Hz and is sent continuously as long as the `enable_in` is set high. In other words, the `enable_in` signal controls whether or not the sensor is activated.

Due to the requirements from laboratory exercise 3 the code is structured so that the pulse width can be adjusted between 10 µs and 20 µs during runtime. This functionality have been partly disabled and the pulse width is set with a static width of 15 µs for this project.

4.2.2 Echo Pulse Analyzer

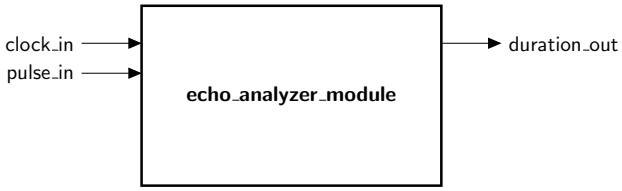


Figure 12: Entity of jrh_echo_analyzer_module.vhd

The purpose of the echo analyzer is to measure the duration of the response pulse from the SRF05 sensor.

The duration is measured using a counter, and according to the SRF05 datasheet [3] a timeout occurs if the duration exceeds 30 ms – this is also handled by this module. To avoid inconsistency in the readout the most recent duration value is set on the `duration_out` signal, this prevents flickering when new echo pulses arrives.

4.2.3 58 Divider

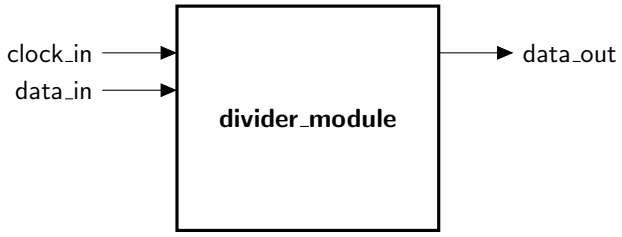


Figure 13: Entity of jrh_divider_module.vhd

The divider module have one task and that is to convert the duration received from the echo analyzer to centimeters. This is done by dividing the pulse duration with the number 58[3].

The distance is saved in a 16-bit wide logic vector, this size is used throughout the system with exception of the RAM. The reason for using 16-bit instead of the minimum 9-bit required is due to laboratory exercise 3 where pulse widths way above 400 µs was tested.

Since regular division requires a lot of resources and should be avoided on the FPGA, bitshifting have been used for division, as shown below.

Division (from jrh_divider_module.vhd)

```
37  if to_integer(unsigned(data_in)) < 58 then
38    data_out <= (others => '0');
39  else
40    -- since regular division should be avoided, (2^16)/58 = 1129,9 =>
          1130.
41    data_out <= std_logic_vector(resize(resize(unsigned(data_in) * 1130,
          32) srl 16, 16));
42  end if;
```

For all values below 58 the `data_out` signal is set to 0. Other values is multiplied with 1130 into a temporary storage of 32 bits to avoid overflow. It is then right shifted 16 positions and resized to a depth of 16 bits. The number 1130 comes from $2^{16}/58 = 1129.93 \approx 1130$ where 16 is the size of the logic vector.

This method was tested using Python[2]. The result of the test showed that the approximation of 1130 can give an error of 0.78 %, at least in Python. In practice, this means that distances from 282 cm is rounded one up, 282 cm becomes 283 cm and so forth. This error could easily be accounted for in the code but since the range sensor already have an uncertainty of measurement[4] this have not been done.

4.3 RAM

The RAM implementation consists of two modules: the main RAM module and a RAM controller. Both modules are written with flexibility and reuse in mind, and are therefor using generic variables to set the RAM length and depth, and address size.

[2]Python - https://www.python.org/

4.3.1 RAM Module

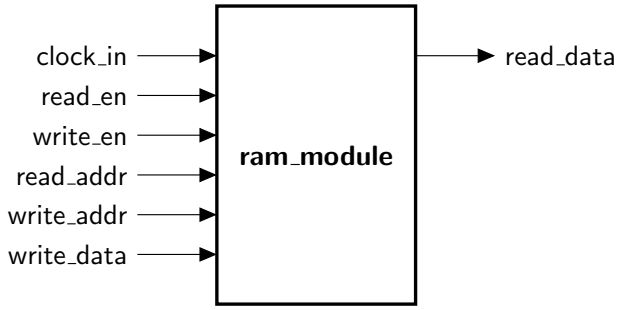


Figure 14: Entity of jrh_ram_module.vhd

The RAM module is implemented as a generic RAM module with enable signals for read and write, signals for data input and output, and a signal for setting the read address and write data.

One of the goals have been to keep the RAM module as simple as possible and keep the structure more or less identical to the one written for laboratory exercise 2. Even though the size of the sensor data has a depth of 16 bits throughout this project, the storage in the RAM module is limited to the minimum size necessary which is a depth of 9 bits. This gives the RAM a size that is just above 4k, $512 \times 9 = 4608$ bit

The RAM is implemented as Block RAM to follow best practices. This is explained more thoroughly in section 5.4.1.

4.3.2 RAM Controller

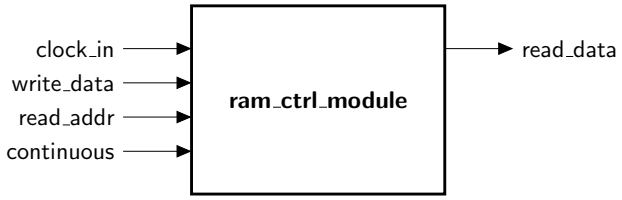


Figure 15: Entity of jrh_ram_ctrl_module.vhd

The RAM controller works as a circular buffer[3] (also known as ring buffer). One counter keeps track of the write index of the RAM memory. This index increases for each data that is written, and the read address is shifted accordingly in the controller as shown in Figure 16.

The controller also takes care of the read and write enable signals, and if the data should be written continuously or for new values only.

By implementing the RAM controller as a circular buffer the RAM module is used as a simplified FIFO without modifying the RAM module itself.

[3] http://www.embedded.com/electronics-blogs/embedded-round-table/4419407/The-ring-buffer

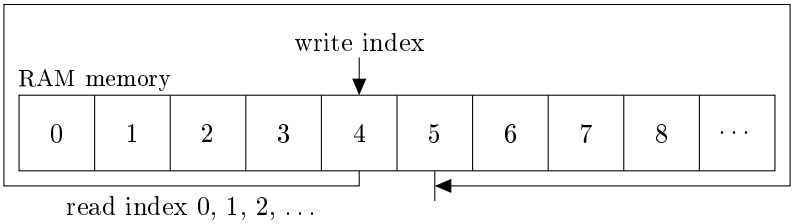


Figure 16: Principle of circular buffers, when increasing the write index the read index is shifted accordingly

4.4 VGA Display

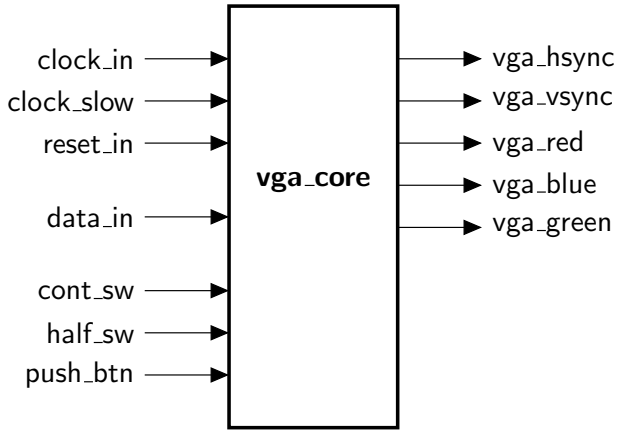


Figure 17: Entity of jrh_vga_core.vhd

The VGA core is one of the more comprehensive sections in this project. It consists of three processes: sending data to the RAM controller, present the data on screen, controlling the position and layout of the presented data. It is also dependent of the RAM controller module, Flag Pattern module and the VGA controller module.

Saving Data Process

This process is pretty straight forward. It is a synchronous process using the slow clock (1 MHz), constrains the sensor data within 0 to 400 and sends it to the RAM controller which takes care of when and how it is saved.

Graphics Process

This process takes care of all the graphical elements drawn on the VGA display, such as the flag pattern, graph frame, horizontal meter and vertical bar plot. The data is read from the RAM controller and is clocked with the VGA clock at 108 MHz.

The division of the vertical bar plot height is also calculated in this process. This is controlled by a switch and is done the same way as the divider in the SRF05 core, using bit shifting to avoid regular division in the code.

Division of bar plot (from jrh_vga_core.vhd)

```
152   if half_sw = '0' then
153     -- if half size, divide data by 2 using right shift
154     -- since regular division should be avoided, 2^9/2 = 256.
155     data := to_integer(resize(resize(unsigned(read_data) * 256, 18) srl
            9, 9));
```

```
156     graph_h := 200;
157   else
158     data := to_integer(unsigned(read_data));
159     graph_h := 400;
160   end if;
```

Animation Process

This process uses all of the push buttons on the Basys3 board to control the position of the graph on the screen. The movement is controlled with a simple state machine where all of the four directions has its own state.

Setting move state (from jrh_vga_core.vhd)

```
271   if move = IDLE then
272
273     if push_btn(0) = '1' then -- center
274       graph_x := 703;
275       graph_y := 100;
276     elsif push_btn(1) = '1' then -- up
277       y_end := graph_y - 10;
278       move := MOVE_UP;
279     elsif push_btn(2) = '1' then -- left
280       x_end := graph_x - 10;
281       move := MOVE_LEFT;
282     elsif push_btn(3) = '1' then -- right
283       x_end := graph_x + 10;
284       move := MOVE_RIGHT;
285     elsif push_btn(4) = '1' then -- down
286       y_end := graph_y + 10;
287       move := MOVE_DOWN;
288     end if;
289
290   end if;
```

When one of the four buttons are pressed, a shared variable used by the Graphic Process will either increase or decrease depending on which state that is activated. The position update frequency is 200 Hz to give a smooth but notable transmission.The fifth push button is used to reset the graph to the default position.

4.4.1 VGA Controller (extern)

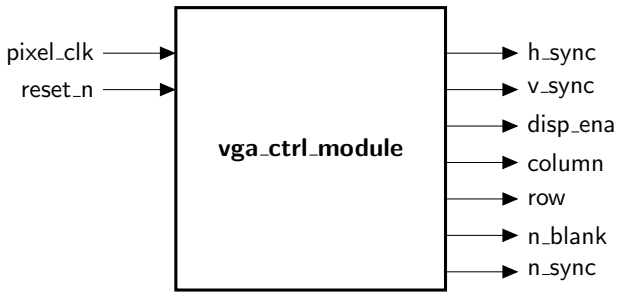


Figure 18: Entity of vga_ctrl_module.vhd

The VGA controller is the same as the one given for laboratory exercise 1. This code have not been modified and is used as is.

4.4.2 Flag Pattern

Figure 19: Entity of jrh_norwegian_flag_pattern.vhd

The flag pattern module is heavily based on the pattern module given with laboratory exercise 1. This module uses the row and column position given by the VGA controller to determine which RGB444 color that should be sent to the VGA display.

This module can be considered as a background pattern for the VGA core and should thus be easy to replace with another pattern module with the same signal structure.

4.4.3 VGA Simulator (testbench component)

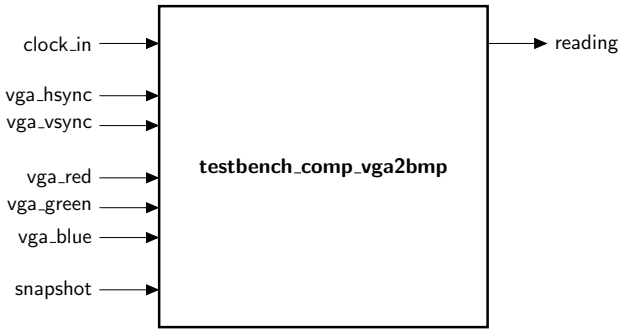


Figure 20: Entity of testbench_comp_vga2bmp.vhd

The VGA simulator have been one of the more challenging but fun components created for this project. It is an addition to the VGA testbench and was created because it seemed like the best possible way to test the VGA core.

The purpose of the VGA simulator is to act as a real VGA display, therefore, it requires only the standard VGA signals to save one VGA frame as BMP if the `snapshot` is set high. The output signal, `reading`, is set high while the simulator is capturing.

This testbench component was created with reuse in mind, meaning the only configuration needed is width and height of the display, and vertical and horizontal back porch settings of the resolution.

VGA simulator configuration

```
1  generic(
2      bmp_width  : integer := 1280;
3      bmp_height : integer := 1024;
4      h_bporch   : integer := 248;
```

```
5       v_bporch   : integer := 38;
6       file_name  : string  := "C:\VHDL\VGA_output.bmp"
7   );
```

The frame capturing works by waiting for the first vertical sync signal if the `snapshot` signal is set high. When the vertical sync is received it compensates for the vertical back porch, before it starts capturing the RGB data between each horizontal sync pulse and horizontal back porch compensation.

The RGB data is converted from RGB444 to RGB888 and saved to a 2-dimensional array during capturing. When the frame is full the BMP header is created and the RGB array is reversed and written to the output file.

4.5 LEDs and 7-Segment Display

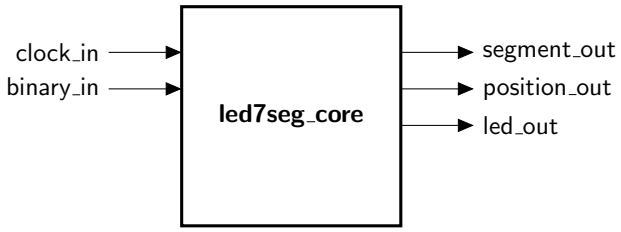


Figure 21: Entity of jrh_led7seg_core.vhd

This core serves as an direct indicator for the sensor value, meaning it does not store the data in RAM but output the data to the 7-segment display and 16 user LEDs directly. The advantages of this is that the «raw» distance can be read without any constrains, i.e. if the value exceeds the 400 cm range.

Two processes are used, one for controlling the LED-meter and one for the 7-segment display.

LED-meter Process

The LED-meter is a synchronous process using the slow clock. It is using Equation 1 to convert the distance into binary ones respectively.

$$led\_out(d) = 2^{\left\| 16 \times \frac{d}{450} \right\|} - 1 \tag{1}$$

To avoid dynamic (mathematical) operations within the code this formula have been translated into a static case structure where each case represent a step of 30 cm up to 450 cm. The upper limit of 450 cm is set intentionally since the sensor range in theory can exceed 400 cm.

7-Segment Display Process

The 7-segment display used is the one on the Basys3 board, this is a scanning display where the signals (LEDs) are active low[5]. The position of the digit is set low (active) and the 7 LEDs are turned on or off accordingly as shown in Figure 22.

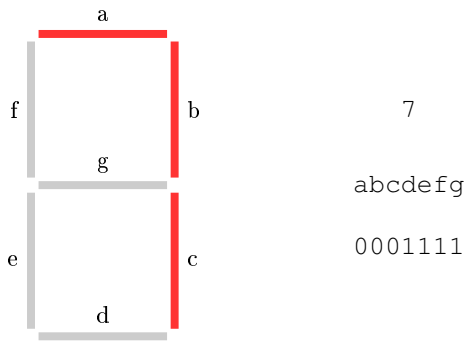


Figure 22: 7-Segment Display numbering

The 7-segment display is driven by the slow clock (1 MHz). Two case structures are used, one for translating the BCD into the 7 segments respectively and one for the position of the digit. The update frequency of the display is set to 1 kHz, this is the fastest possible refresh rate according to the documentation.

Digit	7-Segment	BCD
0	0000001	0000
1	1001111	0001
2	0010010	0010
3	0000110	0011
4	1001100	0100
5	0100100	0101
6	0100000	0110
7	0001111	0111
8	0000000	1000
9	0000100	1001

Table 2: 7-Segment Display conversion

The BCD conversion for the 7-segment display is dependent of the module described below.

4.5.1 Binary to BCD conversion

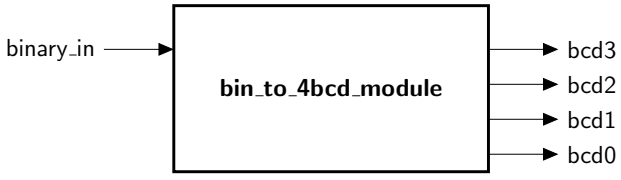


Figure 23: Entity of jrh_bin_to_4bcd_module.vhd

This module converts a 16 bit binary number into 4 binary-coded decimals (BCD). This is done by splitting the 16 bit number into 4 parts and then adjusting the 4 segments accordingly. The code is inspired from an online tutorial[4].

[4] http://www.deathbylogic.com/2013/12/binary-to-binary-coded-decimal-bcd-converter/

4.6 Main Top file

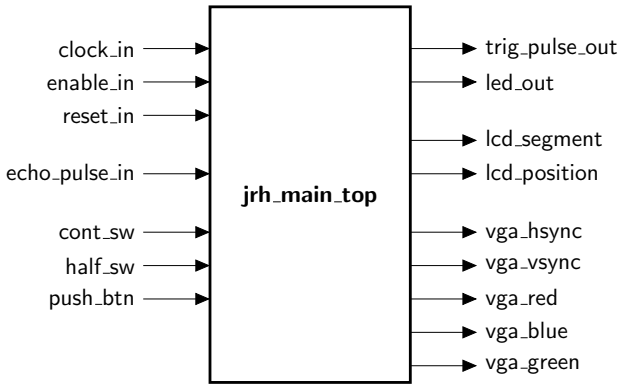


Figure 24: Entity of jrh_main_top.vhd

The top file connects all the cores and modules together and does not contain any processes. The internal signals are distributed to each of the cores respectively, and the input/output signals are connect to the board through the constraint file.

The content of the top file can be viewed in Appendix 10.B or be found in Appendix 10.A.

5 Simulation and verification

This section will describe the testbenches, simulation and verification of the system. Screenshots of simulations that are not annotated should be readable in regards of signal names and values.

5.1 Simulation and verification setup

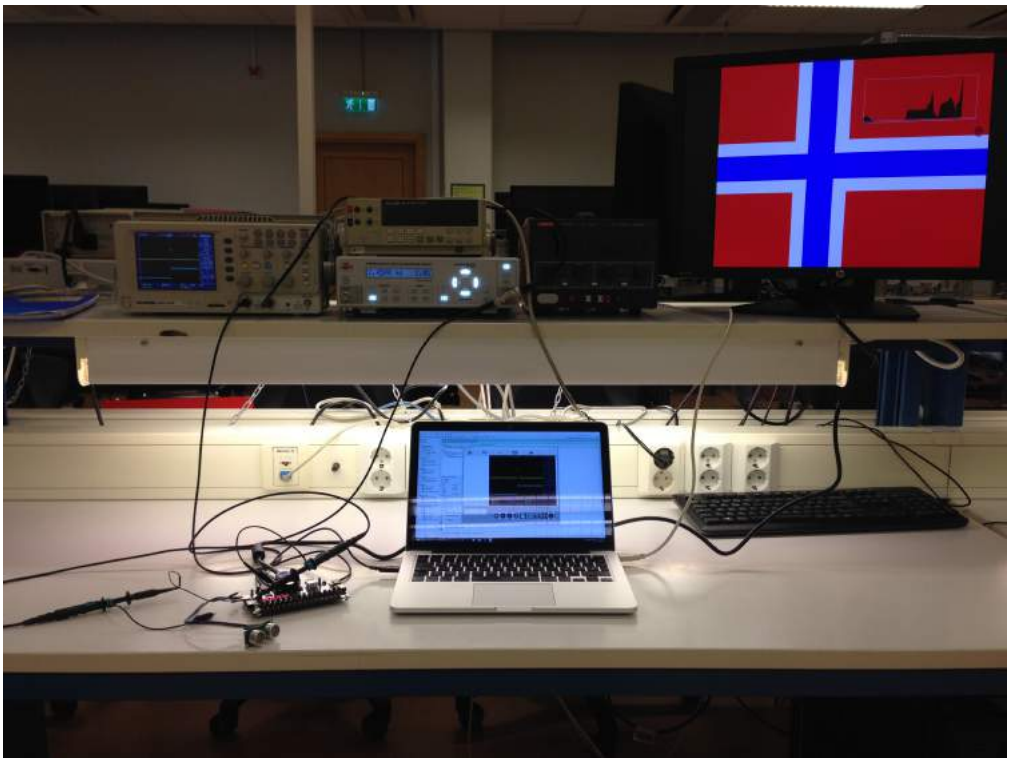


Figure 25: Testing and verification equipment

Simulation: Vivado 2015.1 (VM Windows 10 on MacBook Pro 2015)

Signal generator: HAMEG Instruments HM8150 - Programmable Function Generator

Oscilloscope: GwInstek GDS-1042 (40 MHz, 250 M Sa/s, 2 Ch, USB)

5.2 Clock Generator

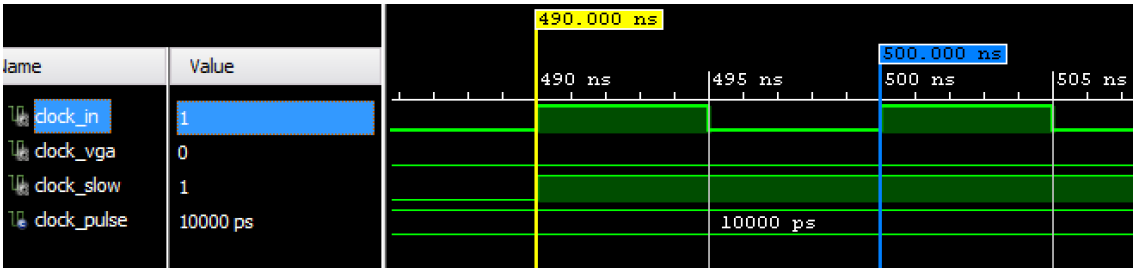


Figure 26: Clock simulation: 100 MHz

$$f = \frac{1}{500\,\text{ns} - 490\,\text{ns}} = 100\,\text{MHz} \tag{2}$$

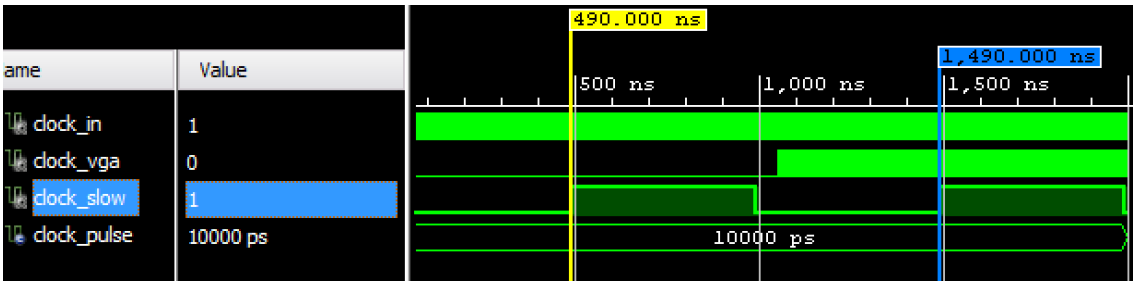


Figure 27: Clock simulation: 1 MHz

$$f = \frac{1}{1490\,\text{ns} - 490\,\text{ns}} = 1\,\text{MHz} \tag{3}$$

The 1 MHz clock was verified using an oscilloscope, unfortunately, this was not documented at the time being.

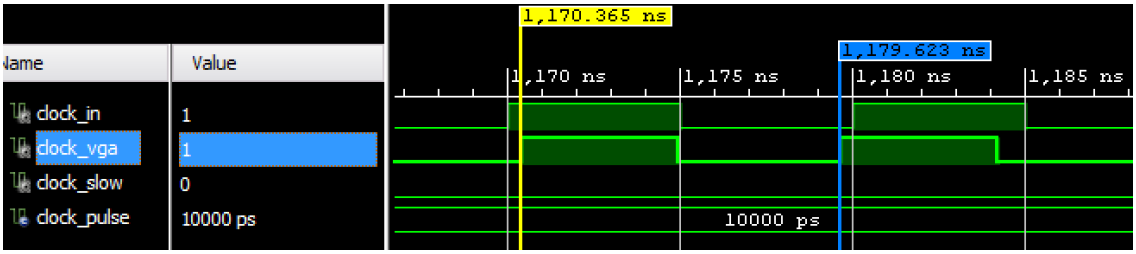


Figure 28: Clock simulation: 108 MHz

$$f = \frac{1}{1179.623\,\text{ns} - 1170.365\,\text{ns}} = 108\,014\,690\,\text{Hz} \approx 108\,\text{MHz} \tag{4}$$

5.3 Range Sensor

Manual simulation and testing have been done for each of the modules during development, the simulations described here will therefore only be based on the dedicated testbench since this summarizes the overall functionality.

5.3.1 Simulation

The testbench generates an echo pulse for each trigger pulse it receives. The echo pulse width is increased by 58 µs for every iteration.

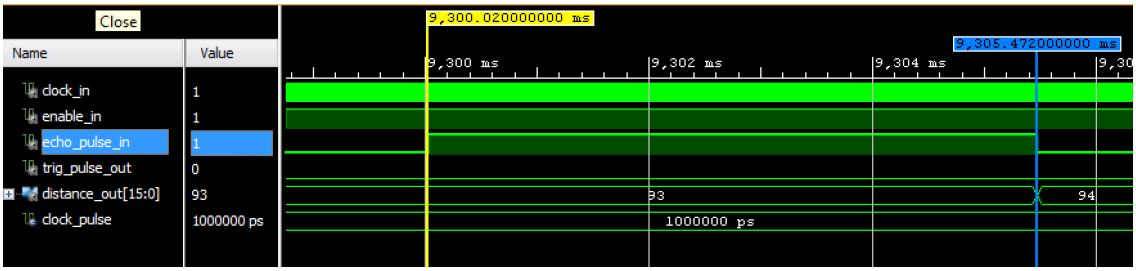


Figure 29: SRF05 simulation: echo pulse

Figure 29 shows a section from the simulation where the distance is 94 cm. By looking at the rising and falling edge of the echo pulse the distance is confirmed as shown in Equation 5.

$$d = \frac{9305472 - 9300020}{58} = 94 \tag{5}$$

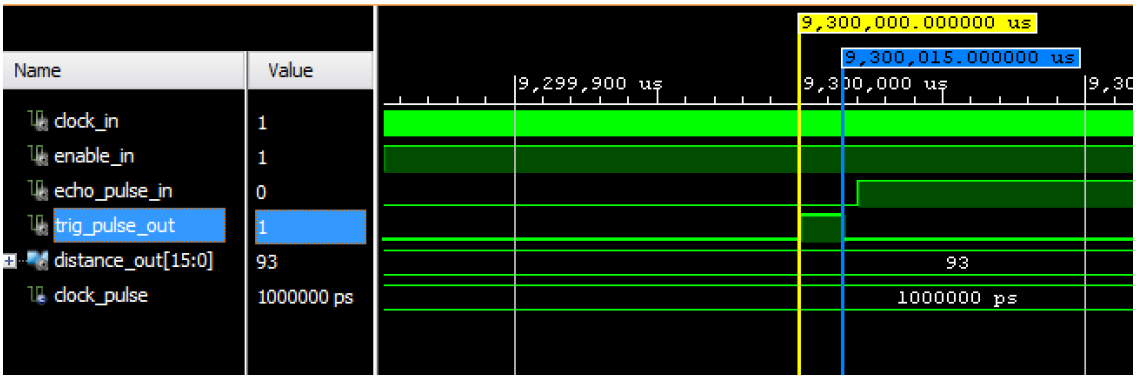


Figure 30: SRF05 simulation: trigger pulse width

A closer look at the trigger pulse shows that the width of the pulse is 15 µs.

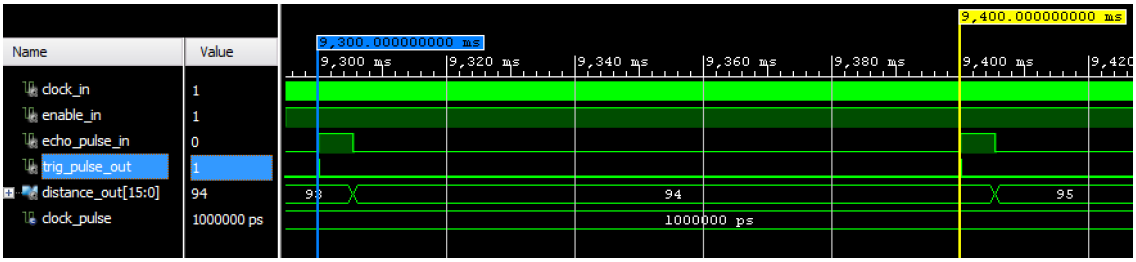
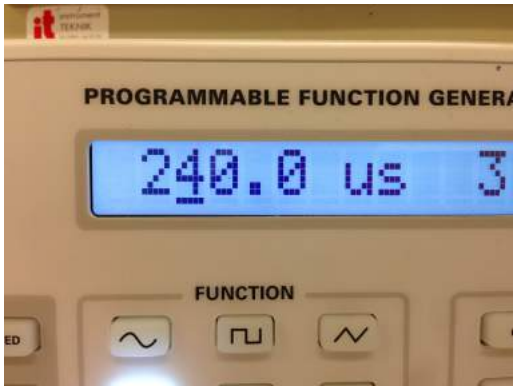


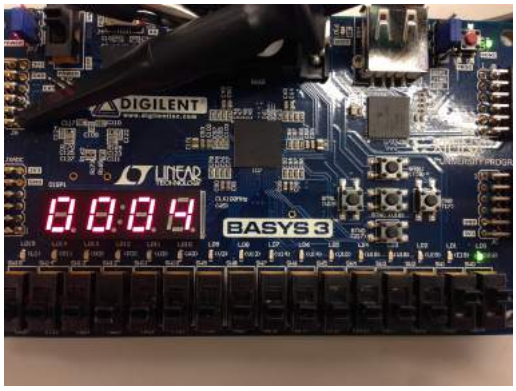
Figure 31: SRF05 simulation: trigger interval

The simulation also shows that the trigger pulse has an interval of 100 ms which gives a frequency of 10 Hz.

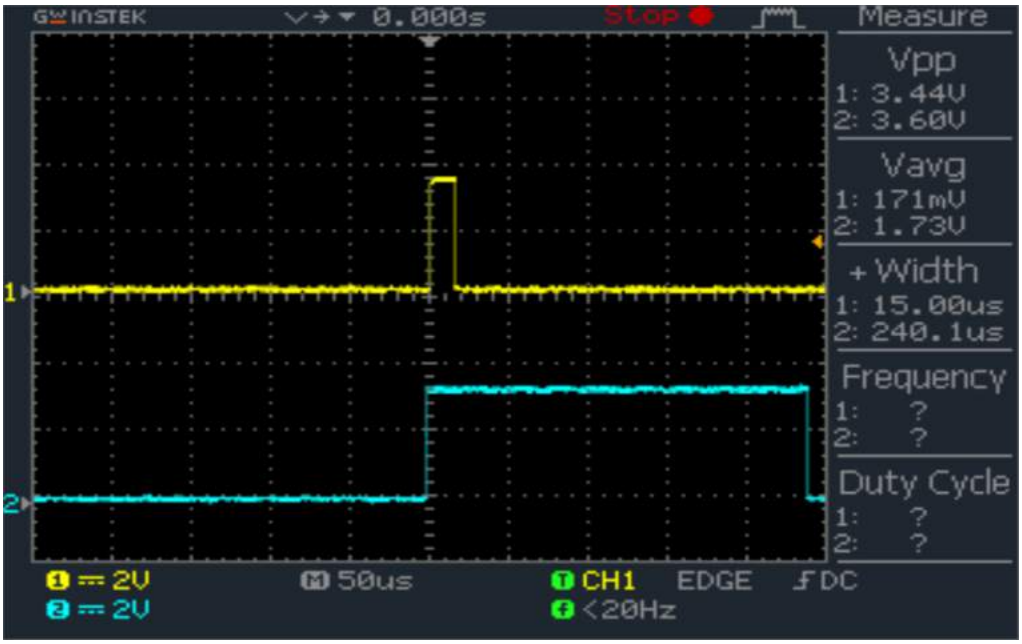
5.3.2 Verification



(a) Signal generator



(b) Board output



(c) Oscilloscope output of trigger pulse and echo pulse

Figure 32: Verifying SRF05 sensor using signal generator and scope

Figure 32 shows the verification of the echo pulse calculation. The signal generator simulates a 240 µs wide echo pulse that the FPGA translates to 4 cm. This is confirmed to be correct in Equation 6.

$$d = \frac{240}{58} = 4.1379 \approx 4 \tag{6}$$

Figure 32b also shows the width of the trigger pulse to be 15 µs as the simulation indicated. A lot of different sized echo pulses was generated without any incorrect results. By generating a narrow echo pulse, such as the one in Figure 32a, the SRF05 core show that it is able to handle short distances.

5.4 RAM

5.4.1 Implementation

There are three ways of implementing RAM on the Xilinx FPGA. Define it as RAM using LUTRAM (Distributed RAM) or BRAM (Block RAM), or using regular LUT/FF for implementation. Each of these methods have been tested and compared to find the best suited implementation for this project.

Resource	Utilization	Available	Utilization %
LUT	1354	20800	6.51
LUTRAM	96	9600	1.00
FF	580	41600	1.39
DSP	1	90	1.11
IO	53	106	50.00
BUFG	4	32	12.50
MMCM	1	5	20.00

Total On-Chip Power:	**0.228 W**
Junction Temperature:	**26,1 °C**
Thermal Margin:	73,9 °C (14,7 W)
Effective ϑJA:	5,0 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

(a) LUTRAM implementation

Resource	Utilization	Available	Utilization %
LUT	1232	20800	5.92
FF	571	41600	1.37
BRAM	0.50	50	1.00
DSP	1	90	1.11
IO	53	106	50.00
BUFG	4	32	12.50
MMCM	1	5	20.00

Total On-Chip Power:	**0.228 W**
Junction Temperature:	**26,1 °C**
Thermal Margin:	73,9 °C (14,7 W)
Effective ϑJA:	5,0 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

(b) BRAM implementation

Resource	Utilization	Available	Utilization %
LUT	3121	20800	15.00
FF	5284	41600	12.70
DSP	1	90	1.11
IO	53	106	50.00
BUFG	4	32	12.50
MMCM	1	5	20.00

Total On-Chip Power:	**0.282 W**
Junction Temperature:	**26,4 °C**
Thermal Margin:	73,6 °C (14,6 W)
Effective ϑJA:	5,0 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

(c) LUT/FF implementation

Figure 33: Different RAM implementations

According to the Xilinx documentation as a rule of thumb, memory sizes below 4 kbit should be implemented as LUTRAM and anything above as BRAM[6]. As mentioned earlier the memory size in this project is 4.608 kbit which is quite close to 4k. Comparing LUTRAM against BRAM, as expected, did not show any major differences. Except that BRAM gives slight reduction in the use of LUT and FF. The documentation shows that a more critical difference between LUTRAM and BRAM is to be expected if the memory size is lager.

A more notable difference arise when comparing LUTRAM and BRAM against LUT/FF, this concerns both power consumption and junction temperature. Using LUT/FF increases the power consumption around 23 %, from 0.228 W to 0.282 W, and the junction temperature by 0.3 °C.

As a result of this the RAM in this project has been implemented as BRAM. This also makes it easier to expand the memory size «for free» in future revisions, if needed, without compromising the power consumption. It should be noted that the BRAM has a delay of one clock cycle, but this is not an issue in this project.

5.4.2 Simulation

A verification module have been created for the RAM module instead of a dedicated testbench. This is to be able to test and verify the memory behavior using the switches on the Basys3 board. The verification module fills the RAM with data from 0 to 399, 9 of the switches on the board sets the read address and the read data is displayed as a binary value using the LEDs.

RAM verification (from verification_ram_module.vhd)

```
72  if ram_full = '1' then -- read address by using switches
73
74    read_en <= '1';
75    read_addr <= switch_in;
76    led_out <= read_data;
77
78  else -- fill ram once
79
80    if count <= 399 then
81      write_en <= '1';
82      write_addr <= std_logic_vector(to_unsigned(count, 9));
83      write_data <= std_logic_vector(to_unsigned(count, 9));
84      count := count + 1;
85    else
86      write_en <= '0';
87      ram_full <= '1';
88      count := 0;
89    end if;
90
91  end if;
```

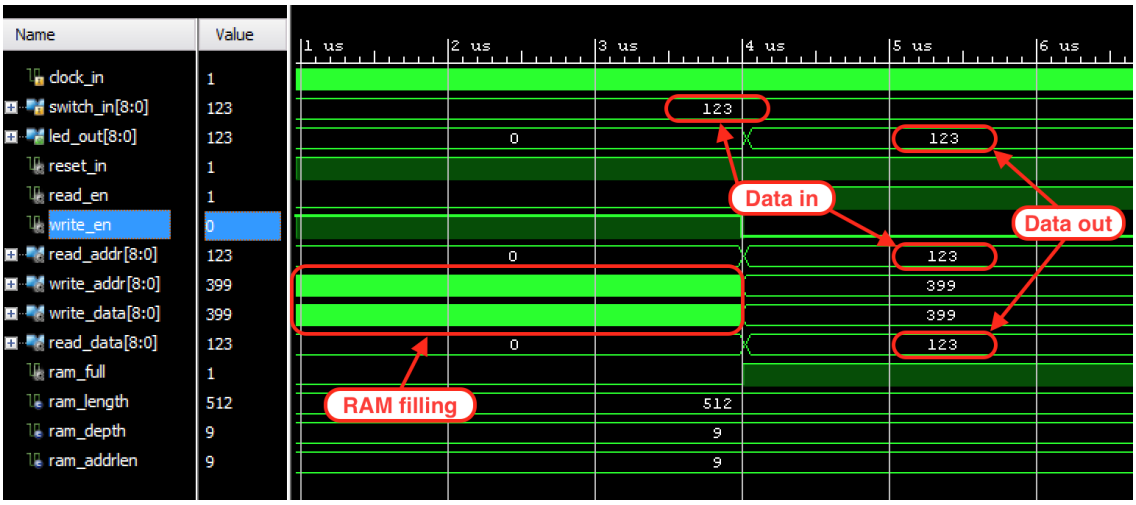
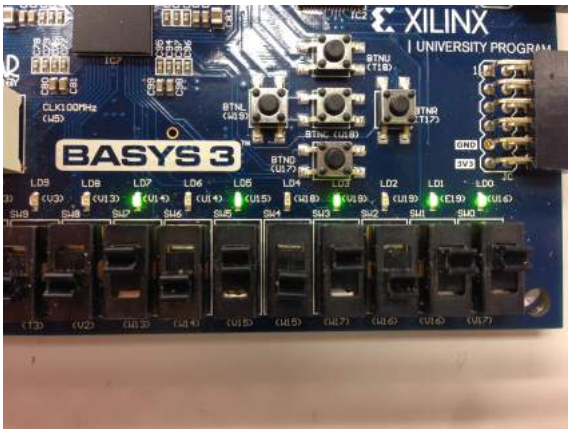


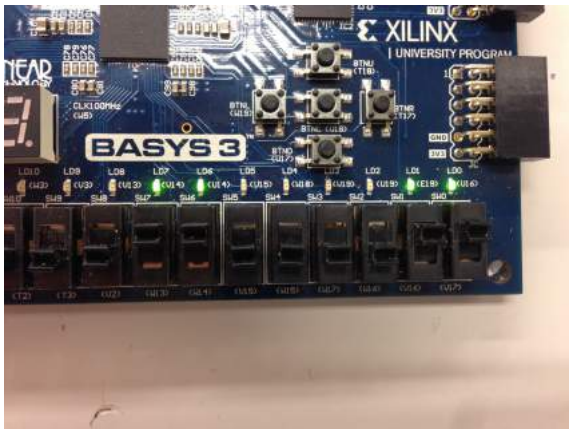
Figure 34: Simulation of RAM verification

Figure 34 shows that after the RAM is full and setting the switches in a position that represent memory address 123, the `read_data` shows the corresponding value from the RAM which is sent to the LED output.

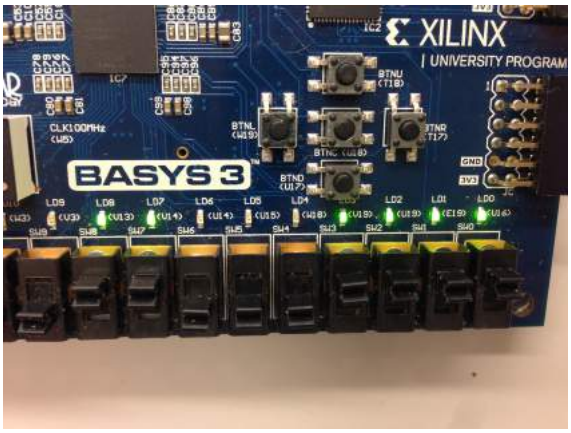
5.4.3 Verification



(a) Reading address 171



(b) Reading address 195



(c) Reading address 399

(d) Reading address 400

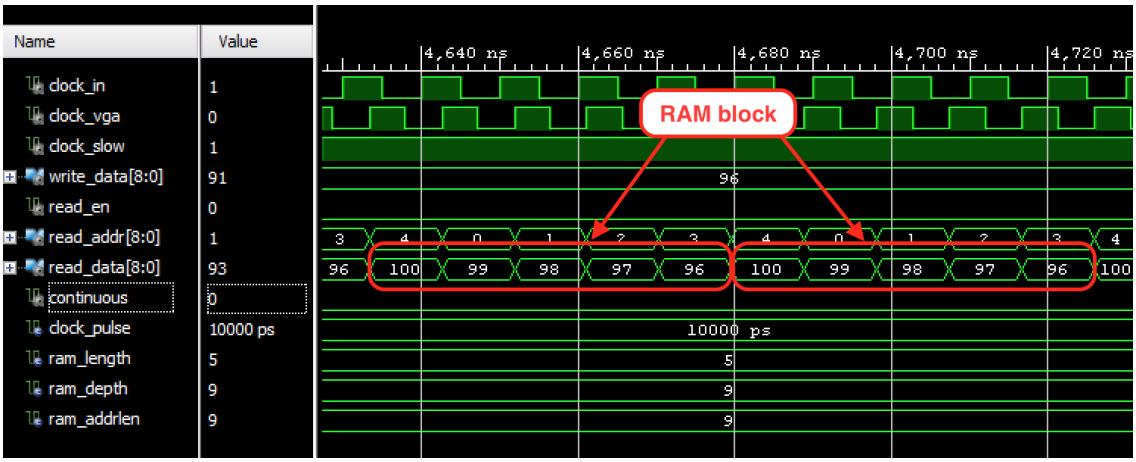
Figure 35: Verifying RAM memory by using LEDs

Testing the code on the FPGA shows that by setting the switches to address 399 the corresponding binary value is displayed on the LEDs. When increasing the address to 400, the read value is 0 as expected. Multiple addresses was tested and all seemed to work as intended.

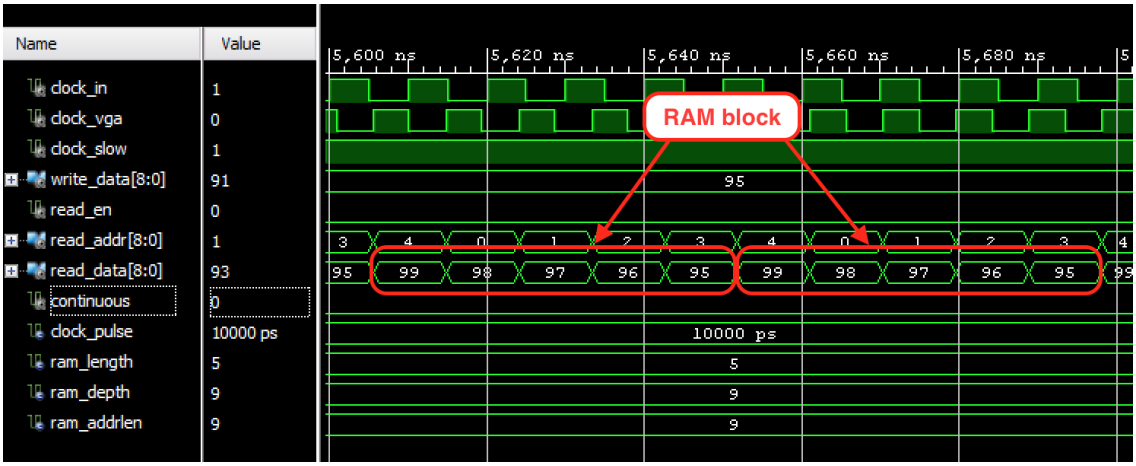
5.5 RAM Controller

This testbench includes the clock generator since the RAM controller is using the vga clock (108 MHz). To make examination of the circular buffer easier, the RAM block length has been set to 5 instead of 512. Furthermore, the continues mode on the controller is disabled, meaning only new values are written to the RAM.

The testbench writes a value starting from 100 at the speed of the slow clock (1 MHz), this value decreases by one for every iteration. While another process is reading the RAM at a frequency of 108 MHz.



(a) Plot at default position



(b) Plot moved with full height

Figure 36: RAM controller testbench

Comparing Figure 36a with Figure 36b shows that the RAM data is shifted one position and that the FIFO functionality is working as intended.

5.6 VGA

The testbench written for the VGA core consists of two files: the VGA core testbench and the VGA simulator, as mentioned earlier.

The testbench simulates sensor data from 0 to 400 twice and then captures the first complete VGA frame. Since the VGA controller does not send a vertical synchronization signal before the second frame, and the VGA frequency is 60 Hz, the simulation time required is around 34 ms.

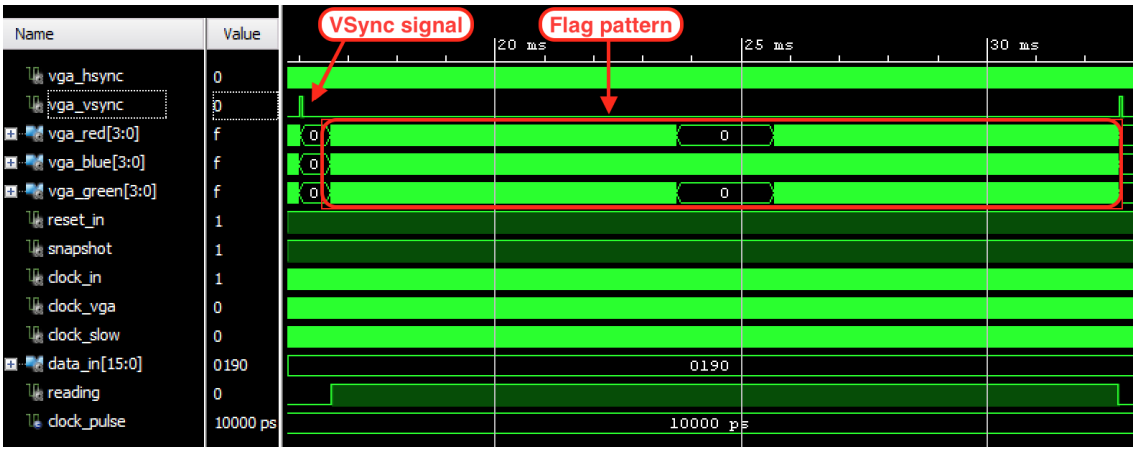
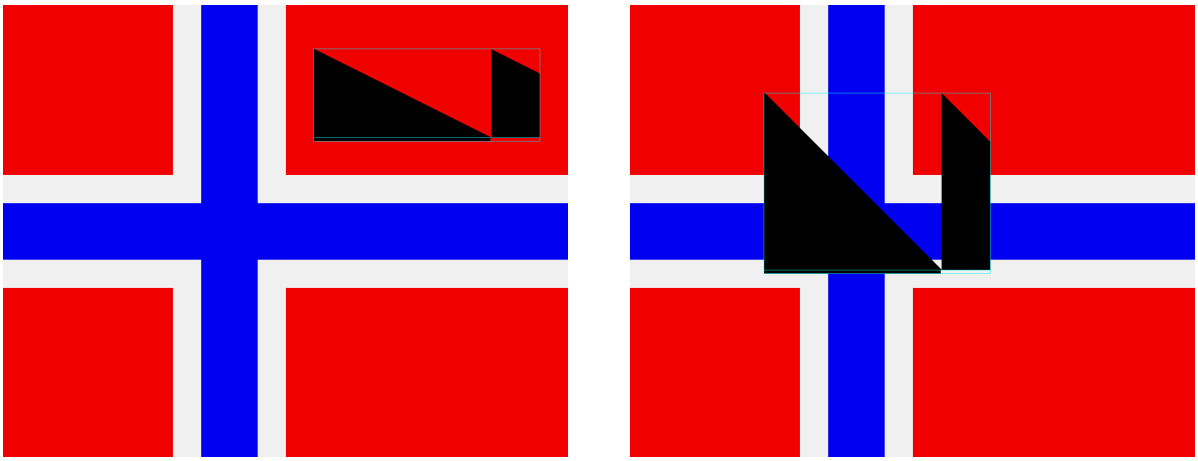


Figure 37: VGA testbench result



(a) Plot at default position

(b) Plot moved with full height

Figure 38: VGA to BMP simulation result

As shown in Figure 38, the flag pattern, the RAM controller and the presentation of the sensor data at different positions works as intended.

A larger image of the VGA output can be viewed in Appendix 10.D.

5.7 LED and 7-Segment Display

The LED and 7-Segment testbench was simulated using two different input numbers. One within the range of the sensor and one way above the range. This is to confirm the behavior of both the display and the LED-meter.

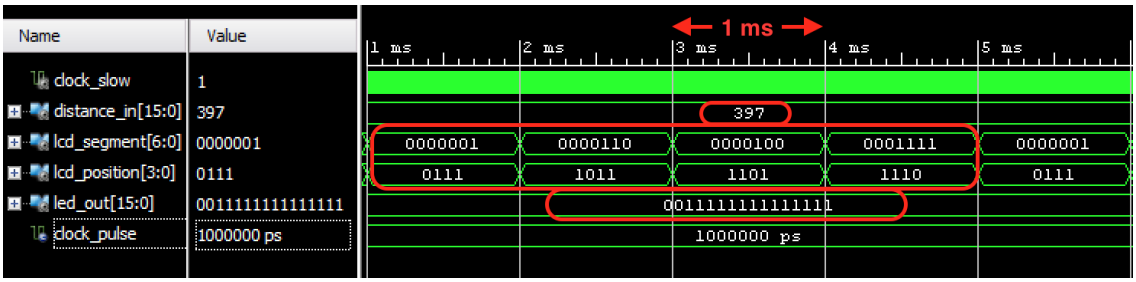


Figure 39: LED and 7-Segment testbench: 397 digit simulation

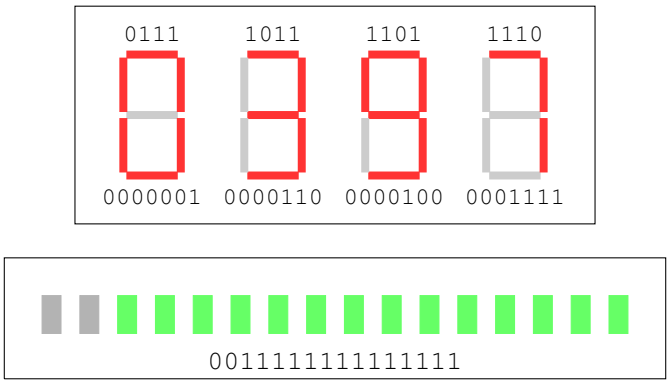


Figure 40: LED and 7-Segment testbench: 397 result

Figure 39 with the corresponding Figure 40 show that the indicators are working as expected. The 7-segment refresh rate of 1 kHz is also confirmed by the simulation.

Figure 41: LED and 7-Segment testbench: 9876 digit simulation

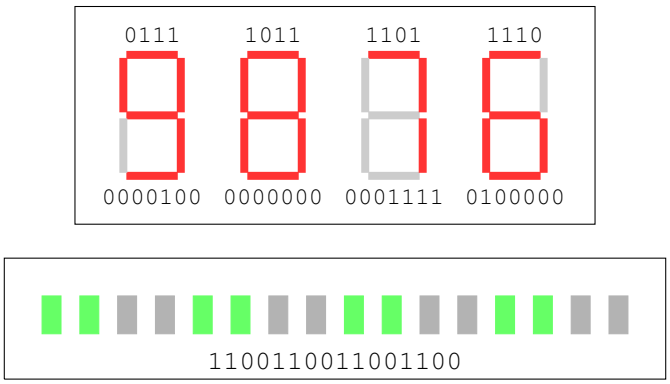


Figure 42: LED and 7-Segment testbench: 9876 result

Figure 41 and Figure 42 shows that all four digits of the 7-segment display is working. Since this value is outside the scope of the LED-meter, the «catch all» case from the case structure is active, hence the out-of-range pattern on the LED-meter is displayed.

5.8 Top File

Due to the long interval between the trigger pulse (100 ms) the testbench for the top file simulates an echo pulse directly instead of waiting for the trigger pulse. This echo pulse is 14.5 ms wide and should return a measured distance of 250 cm.

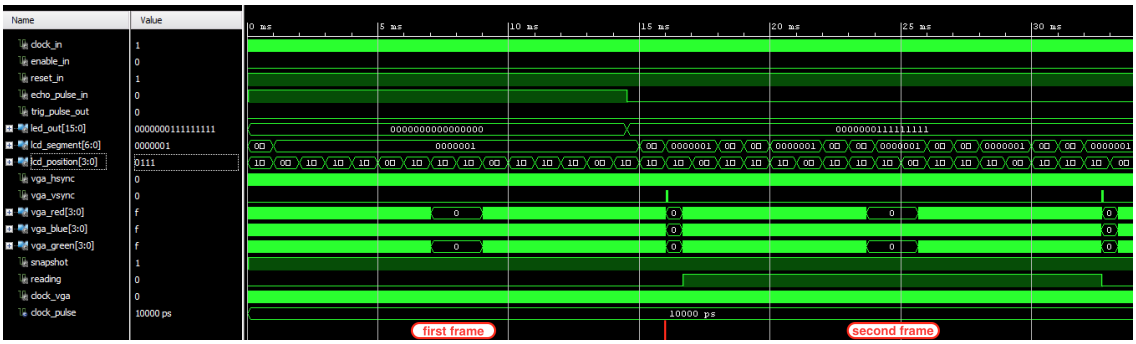


Figure 43: Top file testbench simulation results

Figure 43 shows the simulated echo pulse, LED-meter, 7-segment data and two VGA frames where the last one is captured by the simulator. It also shows that the echo pulse duration is converted into distance at the falling edge of the `echo_pulse_in` signal.

Figure 44: Top file testbench simulation results LED/7-Seg

Figure 44 shows that the LED-meter is set to `0000000111111111`. Since each LED represent 30 cm, this corresponds to a distance between 240 to 270 cm which is correct.

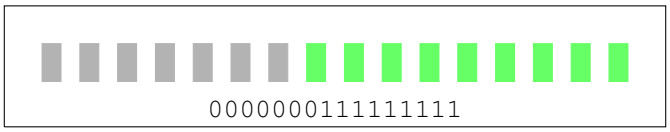


Figure 45: Top file testbench: LED-meter

Taking a closer look at the 7-segment display output and comparing it against Table 2 gives the number 250 as shown in Figure 46.

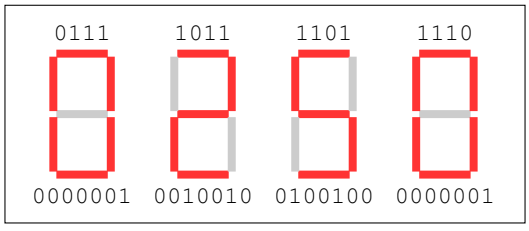
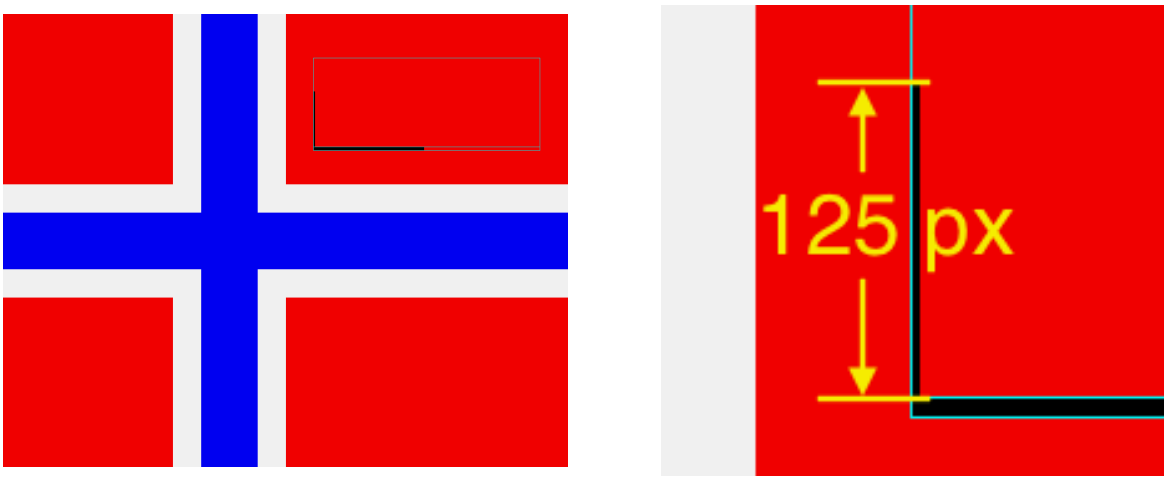


Figure 46: Top file testbench: 7-segment display



(a) VGA output

(b) Bar Plot

Figure 47: Top file testbench: BMP result

Furthermore, zooming in on the BMP output image shows that the bar height of the plot is 125 px high, this is correct since the bar plot by default is divided by 2.

Larger images of the testbench results can be viewed in Appendix 10.C.

6 Testing the system

Two types of measurements were done to test the behavior of the system in practice. One short range measurement and one long range measurement.

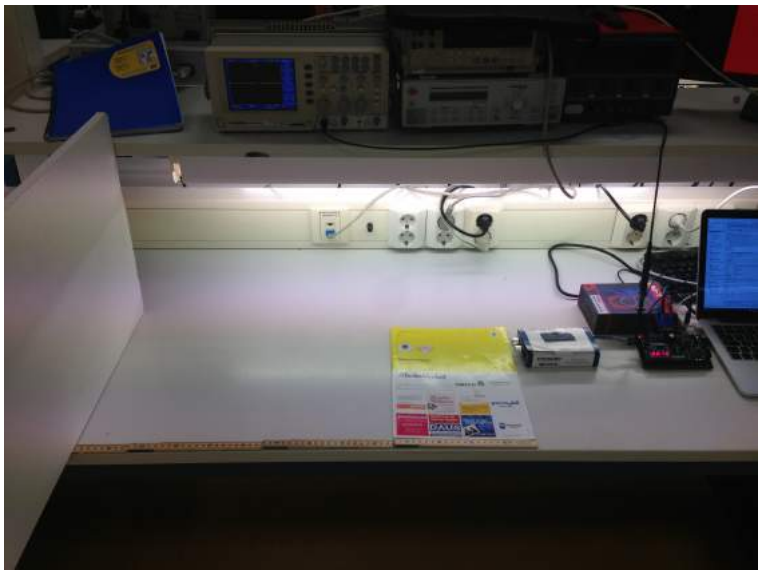


Figure 48: Testing environment: short range

The short range measurement was rigged to be as accurate as possible but seemed to be subject to surrounding noise. Distances measured was 5, 10, 15, 20, 25, 30, 50, 70 and 90 cm.

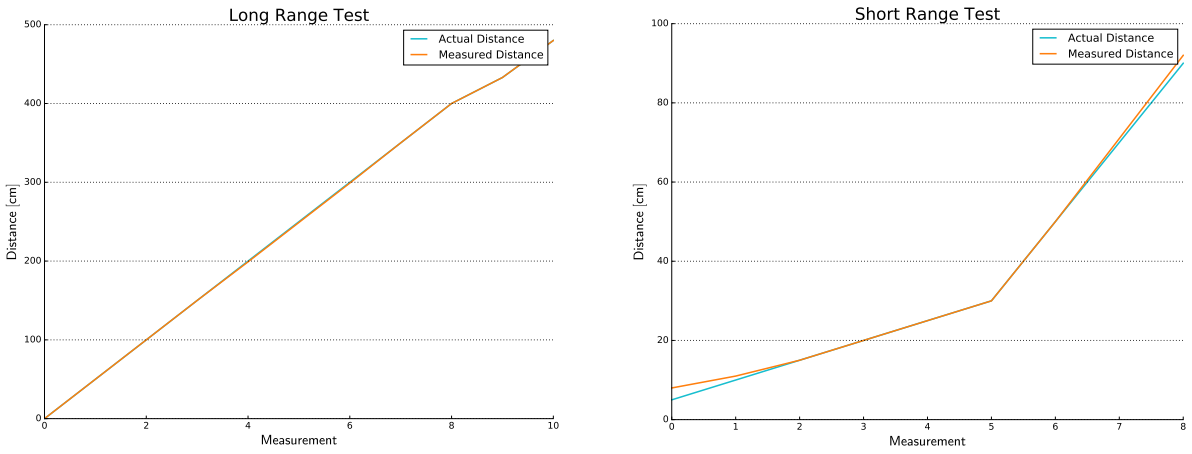
Figure 49: Testing environment: long range

The long range measurement was done in a «clean» and open environment. Distances of 50 cm up to 400 cm was marked using a folding ruler. The reflecting object was calibrated at 50 cm since this distance proved to be accurate during the short range test.

7 Results

Complete reports from Vivado can be found in Appendix 10.A.

7.1 Accuracy



(a) Long range plot

(b) Short range plot

Figure 50: Result: Long and short range plot

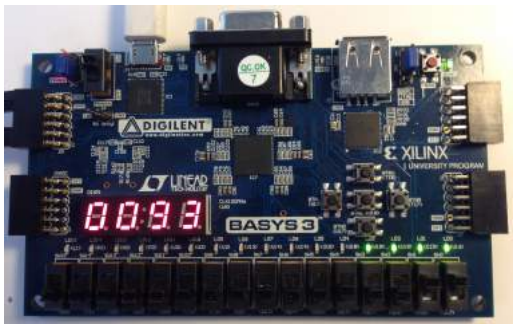
Range [cm]	Measured [cm]
50	50
100	100
150	150
200	199
250	249
300	299
350	350
400	400
433	433
480	480

Table 3: Long range test result

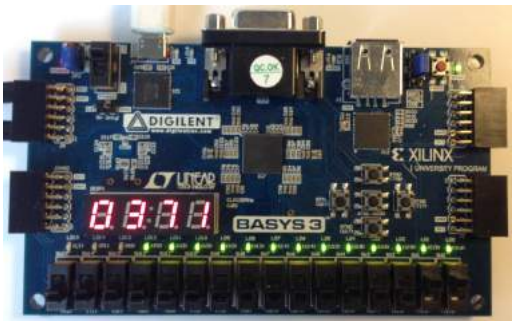
Range [cm]	Measured [cm]
5	7
10	11
15	15
20	20
25	25
30	30
50	50
70	71
90	91

Table 4: Short range test result

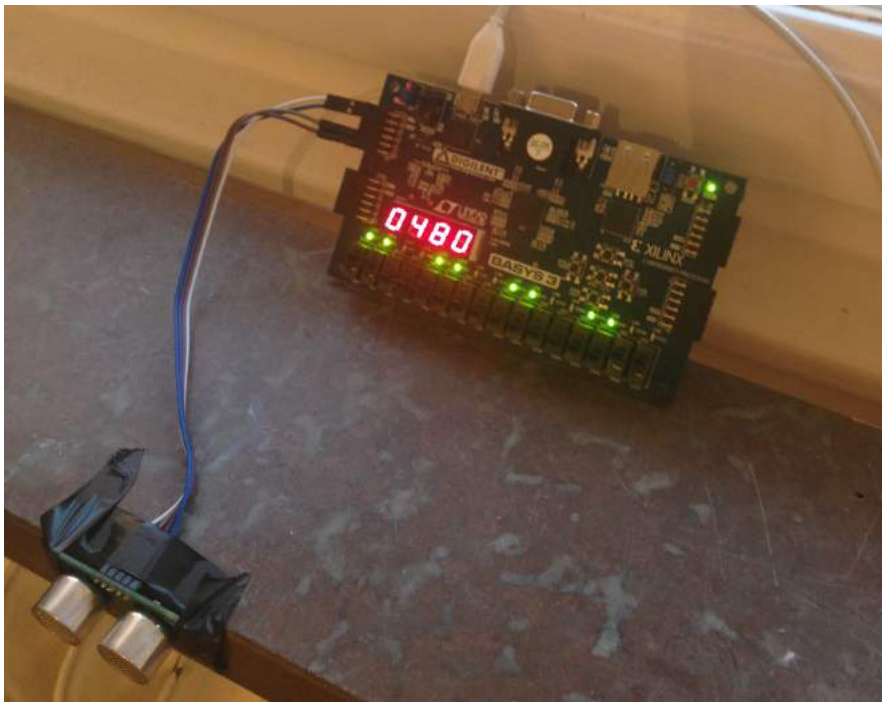
By comparing the mean values the long range test give an error of 0.11 %, while the short range test give an error of 1.59 %. This deviation of the measurement can also be seen in Figure 50. For the long range test the deviation was negative while in the short range the deviation was positive. With exception of the shortest range the deviation was 1 cm in all cases.



(a) 93 cm measured



(b) 371 cm measured



(c) 480 cm measured

Figure 51: Result of measurements

Figure 51a and 51b shows the measurement with the corresponding LED-meter active.

$$LED_{93cm} = \left\lceil \frac{93}{30} \right\rceil = 4 \tag{7}$$

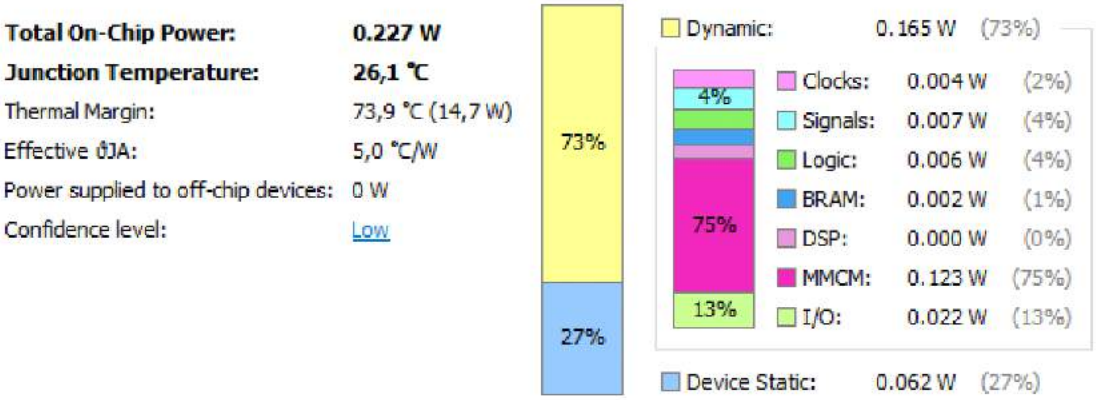
$$LED_{371cm} = \left\lceil \frac{371}{30} \right\rceil = 13 \tag{8}$$

Figure 51c shows that the system is capable of distances above 400 cm. This is the full length of the room. The out-of-range pattern on the LED-meter is also active since the distance exceeds the meter limit of 450 cm.

7.2 FPGA Resources

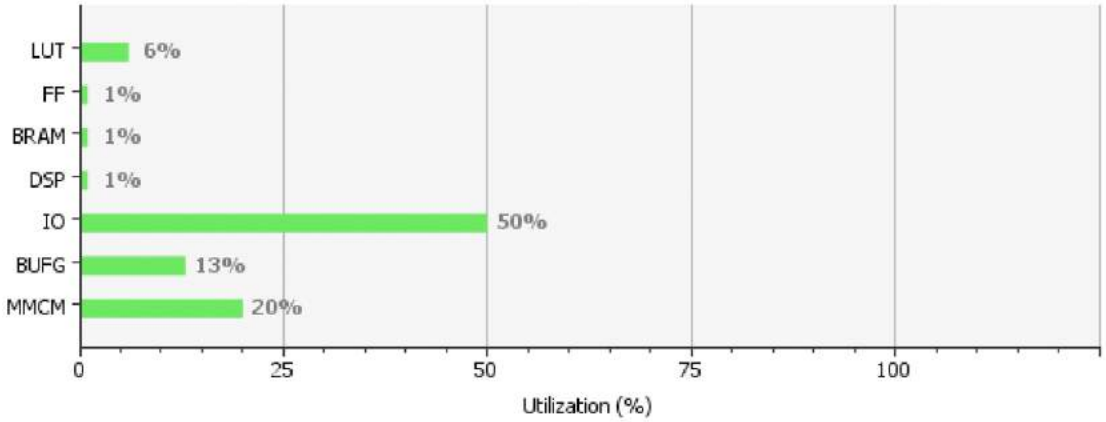
Worst Negative Slack (WNS):	0.213 ns	Worst Hold Slack (WHS):	0.044 ns	Worst Pulse Width Slack (WPWS):	3 ns
Total Negative Slack (TNS):	0 ns	Total Hold Slack (THS):	0 ns	Total Pulse Width Negative Slack (TPWS):	0 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	719	Total Number of Endpoints:	719	Total Number of Endpoints:	366
Implemented Timing Report		Implemented Timing Report		Implemented Timing Report	
Setup Hold Pulse Width		Setup **Hold** Pulse Width		Setup Hold **Pulse Width**	

(a) Timing summary



(b) Power usage summary

Resource	Utilization	Available	Utilization %
LUT	1218	20800	5.86
FF	571	41600	1.37
BRAM	0.50	50	1.00
DSP	1	90	1.11
IO	53	106	50.00
BUFG	4	32	12.50
MMCM	1	5	20.00



(c) Utilization summary

Figure 52: Result FPGA resources

8 Discussion

A lot of considerations have been done in this project. Such as proper RAM implementation, avoiding a secondary reset line[6], structure of the hierarchy, original functionality, general testing and development to name a few. All in all, the results seems to prove that the system is working.

Accuracy

Even though the tests was done as accurate and precise as possible with the equipment that was available: tape, a folding ruler and an near empty room, it can't be considered as a precision test.

For the short range test there was different factors that could affect the results, such as noise and the space available. A recurring problem was distances below 10 cm even though the SRF05 documentation claims to be able to handle distances down to 3 cm. The verification and simulation of the SRF05 core have also proven that the system is able to handle distances down to 1 cm.

The long range test on the other hand was surprisingly accurate. A much more inaccurate measurement result was expected due to the observations mentioned in section 4.2.3. The errors that occurred could just as well be human error, especially since it only affected 3 of 10 distances and the errors did not have any similarities with the possible division error. Table 3 and Figure 51c also shows that the system is capable of precise measurements above the maximum range claimed in the datasheets.

Resources

Figure 52a shows that the system does not violate any timing rules.

By looking at Figure 52b one can see that the MMCM, the 108 MHz clock generator, is causing the majority of the power consumption. In a future revision, for testing and if possible, this could be replaced with a PLL to compare the differences since the MMCM provides more features than the PLL.

Next is the use of I/Os. As mentioned earlier, one of the goals have been to utilize the potential of the Basys3 in regards of user peripherals such as switches, LEDs, etc. Much of this functionality could of course be reduced to save power.

The overall utilization usage, shown in Figure 52c, show that a lot of LUTs and FFs are available for future development. One of the reasons for this is that the RAM is implemented as BRAM.

9 Conclusion

Without any prior experience with FPGA or VHDL before this course, this project have been an interesting «journey» and the learning curve have been steep to say the least. This means that there are probably a lot of adjustments that could be done to improve both the power and resource consumption before the system is (hypothetically) put in production.

When that is said, a lot of time have been invested. Every component is carefully considered and planned as best as possible and have also been simulated and tested thoroughly. The system is working without any glitches or problems. Extra features have been added. The use of FPGA resources and power consumption have been reduced as best as possible.

All in all, the system does the job it is intended to do and should fulfill the requirements of this project.

List of Figures

List of Tables

References

[1] *VESA Signal 1280 x 1024 @ 60 Hz timing*, SECONS Ltd. [Online]. Available: http://tinyvga.com/vga-timing/1280x1024@60Hz

[2] *VHDL Entity Instantiation: Why and How*, FPGA-DEV.COM. [Online]. Available: http://www.fpga-dev.com/leaner-vhdl-with-entity-instantiation/

[3] *SRF05 Ultrasonic Range Sensor*, robot-electronics. [Online]. Available: http://www.robot-electronics.co.uk/htm/srf05tech.htm

[4] *Ultrasonic Rangers FAQ*, robot-electronics. [Online]. Available: http://www.robot-electronics.co.uk/htm/sonar_faq.htm

[5] *Basys 3^{TM} FPGA Board Reference Manual*, Digilent, 4 2016, rev. C. [Online]. Available: https://reference.digilentinc.com/_media/basys3:basys3_rm.pdf

[6] *What are FPGA Power Management HDL Coding Techniques*, Xilinx Training, 2012. [Online]. Available: http://www.xilinx.com/training/downloads/what-are-fpga-power-management-hdl-coding-techniques.pptx

10 Appendices

10.A Project files

All of the project files is supplied with this report in a separate compressed file.

```
jrh_vhdl_files_ET062G.zip
├── bitstream
│   └── <bitstream file>
├── images
│   └── <bmp-files from VGA simulation>
├── reports
│   └── <most relevant reports from Vivado>
├── vhdl_range_project
│   └── <all of the Vivado project files>
└── project_files
    └── <all of the VHDL code>
```

The VHDL code can also be found at:

https://github.com/jorgenrh/ET062G_VHDL_Range_Sensor

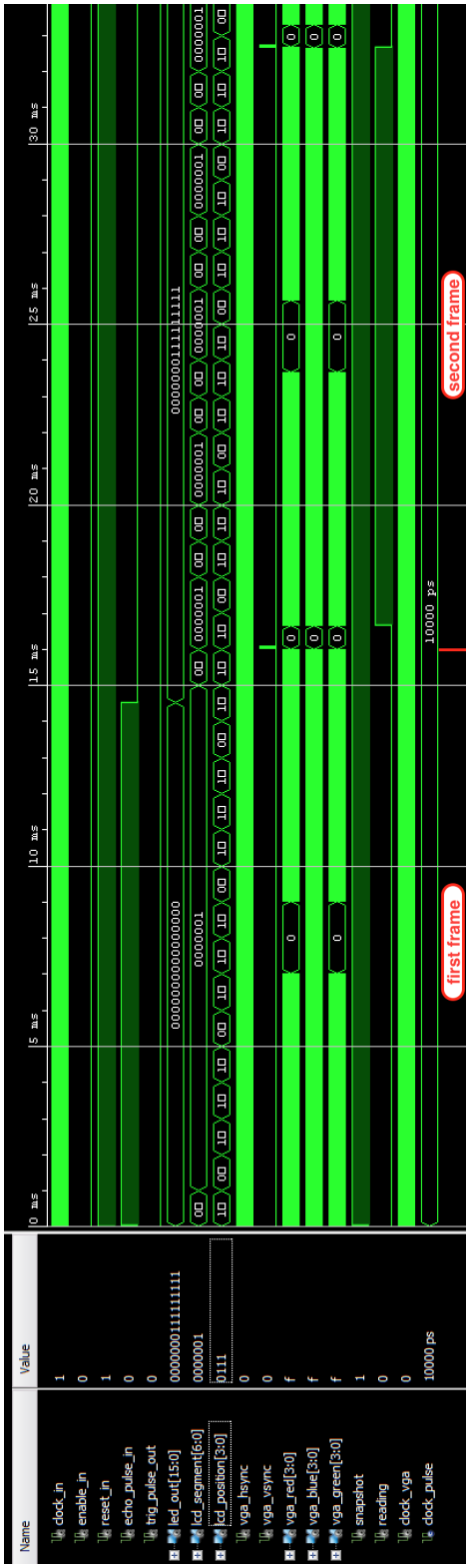
10.B Top File code

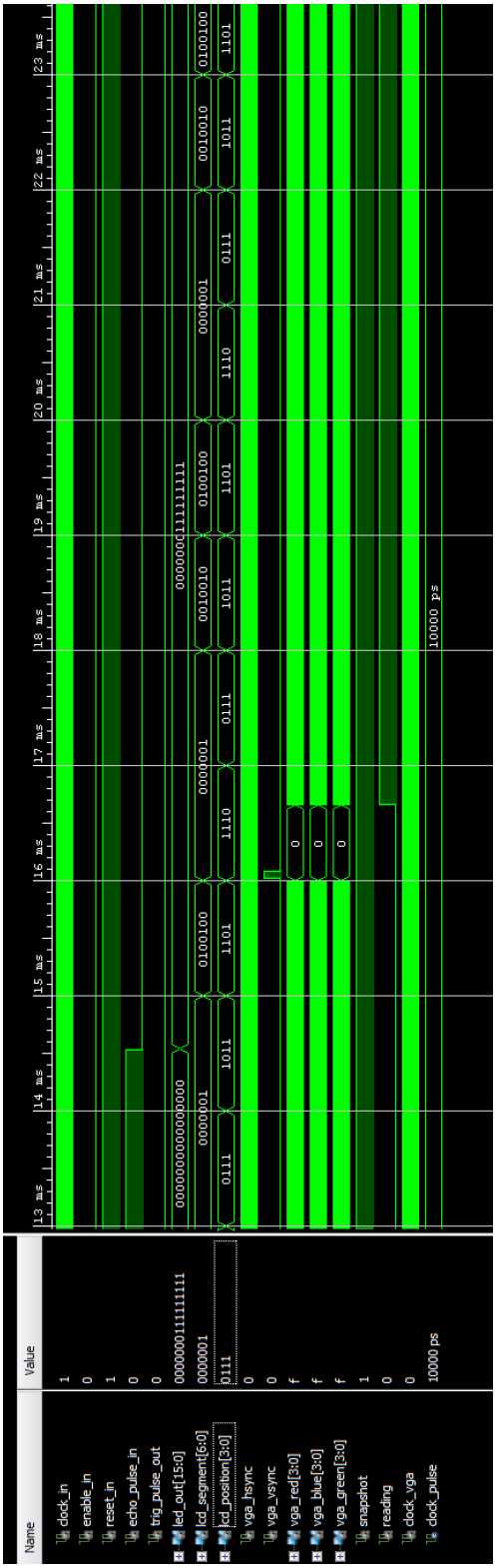
jrh_main_top.vhd

```
1  ----------------------------------------------------------------------------------
2  -- Author:        Jørgen Ryther Hoem
3  --
4  -- Course:        Digital System Design with VHDL (ET062G)
5  --                Mid Sweden University, 2016
6  --
7  -- Module Name:   jrh_main_top - Behavioral
8  -- Project Name:  Range Sensor System
9  -- Description:   Main top file for the project
10 ----------------------------------------------------------------------------------
11
12 library ieee;
13 use ieee.std_logic_1164.all;
14 use ieee.numeric_std.all;
15
16
17 entity jrh_main_top is
18
19     port(
20         -- General
21         clock_in       : in  std_logic;
22         enable_in      : in  std_logic;
23         reset_in       : in  std_logic;
24
25         -- SRF05
26         echo_pulse_in  : in  std_logic;
27         trig_pulse_out : out std_logic;
28         led_out        : out std_logic_vector(15 downto 0) := (others => '0');
29
30         -- LCD
31         lcd_segment    : out std_logic_vector(6 downto 0) := (others => '0');
32         lcd_position   : out std_logic_vector(3 downto 0) := (others => '1');
33
34         -- VGA
35         vga_hsync      : out std_logic := '0';
36         vga_vsync      : out std_logic := '0';
37         vga_red        : out std_logic_vector(3 downto 0) := (others => '0');
38         vga_blue       : out std_logic_vector(3 downto 0) := (others => '0');
39         vga_green      : out std_logic_vector(3 downto 0) := (others => '0');
40
41         -- Data display options
42         cont_sw        : in  std_logic := '0';
43         half_sw        : in  std_logic := '0';
44         push_btn       : in  std_logic_vector(4 downto 0) := (others => '0')
45     );
46
47 end jrh_main_top;
48
49 architecture Behavioral of jrh_main_top is
50
51     -- clock signals
52     signal clock_vga  : std_logic := '0';
53     signal clock_slow : std_logic := '0';
54
55     -- sensor signals
```

```
56      signal int_distance : std_logic_vector(15 downto 0) := (others => '0');
57
58  begin
59
60      CLOCK_GENERATOR:
61          entity work.clock_gen_module
62          port map(
63              clock_100mhz_in  => clock_in,
64              clock_108mhz_out => clock_vga,
65              clock_1mhz_out   => clock_slow
66          );
67
68      SRF05_SENSOR:
69          entity work.srf05_core
70          port map(
71              clock_in       => clock_slow,
72              enable_in      => enable_in,
73              echo_pulse_in  => echo_pulse_in,
74              trig_pulse_out => trig_pulse_out,
75              distance_out   => int_distance
76          );
77
78      LED_7SEGMENT:
79          entity work.led7seg_core
80          port map(
81              clock_in     => clock_slow,
82              binary_in    => int_distance,
83              segment_out  => lcd_segment,
84              position_out => lcd_position,
85              led_out      => led_out
86          );
87
88      VGA_GRAPHICS:
89          entity work.vga_core
90          port map(
91              clock_in   => clock_vga,
92              clock_slow => clock_slow,
93              reset_in   => reset_in,
94              data_in    => int_distance,
95              vga_hsync  => vga_hsync,
96              vga_vsync  => vga_vsync,
97              vga_red    => vga_red,
98              vga_blue   => vga_blue,
99              vga_green  => vga_green,
100             cont_sw    => cont_sw,
101             half_sw    => half_sw,
102             push_btn   => push_btn
103         );
104
105
106
107 end Behavioral;
```

10.C Top File testbench simulation





10.D Large VGA output