Aarhus Universitet

ERTS - Group 2

Rapport

Assignment 2

Gruppemedlemmer:	Studienumre:
Daniel Tøttrup	201509520
Mathias Lønborg Friis	201505665
Stinus Lykke Skovgaard	201401682

6. oktober 2019

Kapitel 1

EXERCISE 3

Write a C-application that implements a command language interpreter, controlled via the USB-UART interface. The following commands must be implemented:

- **1 - Sets the binary value from 0-15 on the red led's by reading switch input (SW0-SW3)**
- **2 - Counts binary the red led's using a timer of 1 sec.**

1.1 Command 1

The essence of the first command is to set a binary value from 0-15 by using the four switches, which will be represented as a binary value by the four LED's on the board.

The code snippet below shows how we read input from the terminal, and execute the a command based on that input.

```
//read input from the terminal in byte
int userInput = inbyte();
xil_printf("Received: %d\r\n", userInput);

switch(userInput)
{
case (int)'1':
xil_printf("Received command 1\r\n");
execute_command_1();
break;
case (int)'2':
xil_printf("Received command 2\r\n");
execute_command_2();
break;
case (int)'3':
xil_printf("Received command 3\r\n");
execute_command_3();
break;
case (int)'4':
```

```
xil_printf("Received command 4\r\n");
execute_command_4();
break;
default:
xil_printf("Received unknown command.");
break;
}
```

Below is a code snippet of the implemented first command.
When the function **execute_command_1()** is called the switches on the board is read with the function **XGpio_DiscreteRead(&dip, 1)**. Where &dip is the InstancePtr that points to the XGpio instance that is being worked on, and the '1' is the channel.

The **writeValueToLEDs(int val)** takes the value returned from the **XGpio_DiscreteRead()** and writes the value to the LED registers.

```
void execute_command_1()
{
xil_printf("Executing command 1\r\n");
while(1)
{
dip_check = XGpio_DiscreteRead(&dip, 1);

writeValueToLEDs(dip_check);

}
}

void writeValueToLEDs(int val)
{
LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, val);
}
```

Command 1 work as expected.

1.2 Command 2

Command 2 should, as states in the problem description, count the binary red LED's using the timer of 1 sec.

First the timer needs to be initialized. Below is a code snippet of how the timer is implemented. Comments in the code explains the implementation.

```
#define ONE_SECOND 325000000 // half of the CPU clock speed

// PS Timer related definitions
XScuTimer_Config *ConfigPtr;
XScuTimer *TimerInstancePtr = &Timer;

int main (void)
{
//initialize timer
ConfigPtr = XScuTimer_LookupConfig (XPAR_PS7_SCUTIMER_0_DEVICE_ID);
s32 Status = XScuTimer_CfgInitialize (TimerInstancePtr, ConfigPtr, ConfigPtr->BaseAddr);
if (Status != XST_SUCCESS){
xil_printf("Timer init() failed\r\n");
return XST_FAILURE;
}

// Load timer with delay in multiple of ONE_SECOND
XScuTimer_LoadTimer(TimerInstancePtr, ONE_SECOND);
// Set AutoLoad mode
XScuTimer_EnableAutoReload(TimerInstancePtr);
}
```

The function **execute_command_2()** sets op a counter and writes the value of counter to the LED's with the function **writeValueToLEDs(counter)** which was explained above. Then it starts the timer, and waits for the timer to expire, which will take one second. When the timer has expired the timer is cleared and the counter is incremented, before the loop starts again. Below is a code snippet of command 2.

```
void execute_command_2()
{
xil_printf("Executing command 2\r\n");
int counter = 0;
while(1)
{
xil_printf("Counter: %d\r\n", counter);
writeValueToLEDs(counter);

// Start the timer
XScuTimer_Start(TimerInstancePtr);

// Wait until timer expires
while(!XScuTimer_IsExpired(TimerInstancePtr));

// clear status bit
XScuTimer_ClearInterruptStatus(TimerInstancePtr);

counter++;
```

```

// Timer auto-enables

}
```

The function **execute_command_2()** works as expected.

Kapitel 2

EXERCISE 4

This exercise will deal with the steps to implement a matrix multiplier.

At first three global variables of the data structure **vectorArray** called **pInst**, **aInst** and **bTInst** were created.

Next a function called **setInputMatrices()** was implemented. This function will initialize two matrices. Below is a code snippet of the implementation. The comments in the code explains how the each element in the matrices are initialized.

```
void setInputMatrices()
{
//Set matrix a
// A, row 0
aInst[0].comp[0] = 1;
aInst[0].comp[1] = 2;
aInst[0].comp[2] = 3;
aInst[0].comp[3] = 4;

// A, row 1
aInst[1].comp[0] = 5;
aInst[1].comp[1] = 6;
aInst[1].comp[2] = 7;
aInst[1].comp[3] = 8;

// A, row 2
aInst[2].comp[0] = 9;
aInst[2].comp[1] = 10;
aInst[2].comp[2] = 11;
aInst[2].comp[3] = 12;

// A, row 3
aInst[3].comp[0] = 13;
aInst[3].comp[1] = 14;
aInst[3].comp[2] = 15;
aInst[3].comp[3] = 16;

//Set matrix bT
// A, row 0
bTInst[0].comp[0] = 1;
```

```
bTInst[0].comp[1] = 2;
bTInst[0].comp[2] = 3;
bTInst[0].comp[3] = 4;

//Set matrix bT
// A, row 1
bTInst[1].comp[0] = 1;
bTInst[1].comp[1] = 2;
bTInst[1].comp[2] = 3;
bTInst[1].comp[3] = 4;

//Set matrix bT
// A, row 2
bTInst[2].comp[0] = 1;
bTInst[2].comp[1] = 2;
bTInst[2].comp[2] = 3;
bTInst[2].comp[3] = 4;

//Set matrix bT
// A, row 3
bTInst[3].comp[0] = 1;
bTInst[3].comp[1] = 2;
bTInst[3].comp[2] = 3;
bTInst[3].comp[3] = 4;
}
```

Next a function called **displayMatrix(vectorArray matrix)** was implemented. As the name of the function strongly implies, this function will display a given matrix in the terminal. To implement **displayMatrix(vectorArray matrix)** a function called **displayMatrixRow(vectorArray matrix, int row)** is implemented in addition. That's because we have to display the matrix row for row. Below is a code snippet of the implemented function.

```
void displayMatrix(vectorArray matrix)
{
xil_printf(”[\r\n”);
displayMatrixRow(matrix,0);
displayMatrixRow(matrix,1);
displayMatrixRow(matrix,2);
displayMatrixRow(matrix,3);
xil_printf(”]\r\n”);
}

void displayMatrixRow(vectorArray matrix, int row)
{
xil_printf(”%d %d %d %d \r\n”, matrix[row].comp[0], matrix[row].comp[1],
    matrix[row].comp[2], matrix[row].comp[3]);
}
```

Next a function called **multiMatrixSoft(vectorArray in1, vectorArray in2, vectorArray out)** was implemented, which is responsible for computing 4x4 matrix product of the expression

$$P = AxB^T$$

This function takes three input because its not possible to return a matrix. So the result matrix is given as the third input parameter, that way we can set the new value of the result matrix in the function. To iterate all combinations of the result matrix P to calculate the sum of products of each element in P, three nested for-loops were used. Below is a code snippet of the **multiMatrixSoft(vectorArray in1, vectorArray in2, vectorArray out)**

```
void multiMatrixSoft(vectorArray in1, vectorArray in2, vectorArray out)
{
  for (int row = 0 ; row < MSIZE; row++)
  {
    for (int col = 0 ; col < MSIZE; col++)
    {
      xil_printf("[%d,%d]\r\n", row, col);
      for (int i = 0 ; i < MSIZE ; i++)
      {
        out[row].comp[col] += in1[row].comp[i] * in2[row].comp[i];
      }
    }
  }
};
```

Next, we wanted to test the multiMatrixSoft function, and test the execution time in clock ticks. For this we implemented a timer, and two integers: **time_SW_pre** and **time_SW_post**. We use the **XScuTimer_GetCounterValue(TimerInstancePtr)** to get the elapsed time before and after the execution and store it in the variables **time_SW_pre** and **time_SW_post**. We then subtract **time_SW_post** form **time_SW_pre** to get the excution time in clock ticks. Below is the a code snippet of this with comments.

```
 XScuTimer Timer; /* Cortex A9 SCU Private Timer Instance */

 // PS Timer related definitions
 XScuTimer_Config *ConfigPtr;
 XScuTimer *TimerInstancePtr = &Timer;

// Multiply matrices in SW.
xil_printf("Calculating P in SW\r\n");
int time_SW_pre, time_SW_post;
XScuTimer_Start(TimerInstancePtr);  // Start the timer
time_SW_pre = XScuTimer_GetCounterValue(TimerInstancePtr); // Get elapsed time
multiMatrixSoft(aInst, bTInst,pInst);
```

```
time_SW_post = XScuTimer_GetCounterValue(TimerInstancePtr); // Get elapsed time
int time_elapsed_SW = time_SW_pre-time_SW_post; // Calculate elapsed time
xil_printf(”P = \r\n”);
displayMatrix(pInst);

xil_printf(”SW: %d\r\n”,timeElapsed_SW); //print execution time
```

On figur 2.1 a screenshot of the terminal is displayed. It shows the two matrices that is being multiplied, their product and at the bottom the time it took to execute in clock cycles.

```
Commands:
1 - Set binary value of LED by reading switch input.
2 - Timer counts using red LEDs.
3 - Matrix multiplication3.
Received: 51
Received command 3
Executing command 3
Matrix multiplication
A =
[
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
]
bT =
[
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
]
Calculating P in SW
[0,0]
[0,1]
[0,2]
[0,3]
[1,0]
[1,1]
[1,2]
[1,3]
[2,0]
[2,1]
[2,2]
[2,3]
[3,0]
[3,1]
[3,2]
[3,3]
P =
[
30 30 30 30
70 70 70 70
110 110 110 110
150 150 150 150
]

Multiplication times (clock cycles):
SW: 3159612
```

Figur 2.1: multiMatrixSoft result in terminal

Kapitel 3

EXERCISE 5

The goal og this exercise is to integrate a IP core into the project. This IP core is capable of multiplying two matrices and output the result. First we have to add the core to the project. This is done by following the lab3b_ EmbeddedSystem.pdf guide.

When the IP core has been added a method for it also needs to be implemented. This can be seen in the code below.

```
1  void multiMatrixHard(vectorArray in1, vectorArray in2, vectorArray out)
2  {
3    for (int row = 0 ; row < MSIZE; row++)
4    {
5      for (int col = 0 ; col < MSIZE; col++)
6      {
7        xil_printf("[%d,%d]\r\n", row, col);
8
9        Xil_Out32(XPAR_MATRIX_IP_0_S00_AXI_BASEADDR +
             MATRIX_IP_S00_AXI_SLV_REG0_OFFSET, in1[row].vect);
10       Xil_Out32(XPAR_MATRIX_IP_0_S00_AXI_BASEADDR +
             MATRIX_IP_S00_AXI_SLV_REG1_OFFSET, in2[col].vect);
11       out[row].comp[col] =
             Xil_In32(XPAR_MATRIX_IP_0_S00_AXI_BASEADDR +
             MATRIX_IP_S00_AXI_SLV_REG2_OFFSET);
12
13     }
14   }
15
16 }
```

It can be seen that the code iterates the rows and columns of the matrices. This means that each element of the matrices are being multiplied and added. At line 11 it can be seen that the output of this is put into an output array, which is the result of the multiplication.

Lastly we will look into the execution time between the hardware and the software implementation. We do this by using the XScuTimer from before. It is initiated just before the multiMatrixSoft() method and when the program has finished the matrix calculations a new timestamp is extracted from the timer. These two numbers are subtracted from each other and this gives us the execution time for the method. The same approach has been made for the multiMatrixHard() method.

```
xil_printf(”Executing command 3\r\n”);
xil_printf(”Matrix multiplication\r\n”);
setInputMatrices();

xil_printf(”A = \r\n”);
displayMatrix(aInst);

xil_printf(”bT = \r\n”);
displayMatrix(bTInst);

// Multiply matrices in SW.
xil_printf(”Calculating P in SW\r\n”);

XScuTimer_Start(TimerInstancePtr); //Start timer
multiMatrixSoft(aInst, bTInst,pInst);

int time_SW_post = XScuTimer_GetCounterValue(TimerInstancePtr); // Get elapsed time in clock cycles
int time_elapsed_SW = ONE_SECOND - time_SW_post; // Calculate elapsed time

xil_printf(”P = \r\n”);
displayMatrix(pInst);



// Multiply matrices in HW.
int time_HW_post;
setInputMatrices(); // Make sure that matrices are ”reset”.
xil_printf(”Calculating P in HW\r\n”);

XScuTimer_RestartTimer(TimerInstancePtr);  // Start the timer
multiMatrixHard(aInst, bTInst, pInst);
time_HW_post = XScuTimer_GetCounterValue(TimerInstancePtr); // Get elapsed time in clock cycles
int time_elapsed_HW = ONE_SECOND - time_HW_post; // Calculate elapsed time

xil_printf(”P = \r\n”);
displayMatrix(pInst);


//Display elapsed time of matrix multiplications
xil_printf(”\r\n”);
xil_printf(”Multiplication times (clock cycles):\r\n”);
xil_printf(”SW: %d\r\n”,time_elapsed_SW);
xil_printf(”HW: %d\r\n”,time_elapsed_HW);
```

The two variables **time_elapsed_SW** and **time_elapsed_HW** hold the amount of clock cycles used for the two different calculation methods. This can be recalculated into seconds by using the below equation:

$$TimeElapsedInSecs = \frac{ClockCyclesUsed}{FreqOfProcessor}$$

The result of this gives us

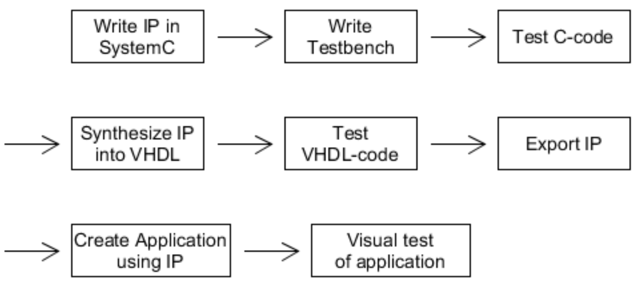
- Hardware = 9.7224ms
- Software = 9.7219ms

The hardware solution should in theory be faster, but because of the frequency of the CPU is faster than the FPGA the software solution is still the fastest.

Kapitel 4

EXERCISE 7

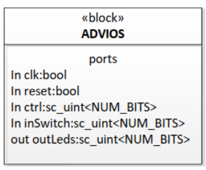
In this exercise, a custom IP called ”advios”is created using SystemC. Then a testbench is written that verifies the functionality of the IP. Using the tool ”vivado HLS”, the C-code is synthesized into VHDL-code. Using the aforementioned testbench, the VHDL-code is verified to have the same funcitonlaity as the C-code. After synthesizing the IP, a project is created, in which the IP is used in an application, and the functionality is tested through visual inspection.



Figur 4.1: Procedure of exercise 7.

4.1 Implementation of IP using SystemC

First, the IP is implemented using SystemC. The interfaces and the functionality is given in the assignment, as seen in figures **??** and **??**.



Figur 4.2: Procedure of exercise 7.

Ctrl value	Behavior
0x0	The outLeds are incremented by a second counter and cleared by inSwitch. *outLeds = increments every 1 seconds* *if inSwitch = 0x8 then clear outLeds*
0x1 – 0f	The value of outLeds are masked by ctrl register and inSwitch. *outLeds = ctrl AND inSwitch*

Figur 4.3: Procedure of exercise 7.

The IP is implemented in the following .h and .cpp-files.

```
#pragma once
#include <systemc.h>

#define NUM_BITS 4
#define CLKFREQ = 100000000

#ifndef _ADVIOS_
#define _ADVIOS_

#include <systemc.h>

SC_MODULE(advios) {

//Ports
sc_in <bool> clk;
sc_in <bool> reset;
sc_in<sc_uint<NUM_BITS> > ctrl;
sc_in<sc_uint<NUM_BITS> > inSwitch;
sc_out<sc_uint<NUM_BITS> > outLeds;

// Signal to communicate between the two threads.
sc_signal<bool> oneSecPulse;

//Variables
sc_uint<8> switchs;

int clkCount; // Used in clock-divider-thread

//Process Declaration
void adviosThread();
void clkDivide();
void writeToLeds();

//Constructor
SC_CTOR(advios) {
clkCount = 0;
//Process Registration
// Clock-divider-thread, which outputs a high signal to oneSecPulse once every second.
SC_CTHREAD(clkDivide, clk.pos());
reset_signal_is(reset, true);

// "Main"-thread for the module.
SC_CTHREAD(adviosThread, clk.pos());
}
};

#endif
```

```
#include "Advios.h"

void advios::clkDivide()
{
while (1)
{
// Clock = 100 MHZ.
// Every 100*1000*1000 cycles, make a "true" pulse, and reset the counter.
clkCount++;
if (clkCount >= 100000000)
{
oneSecPulse.write(true);
clkCount = 0;
}
else
{
oneSecPulse.write(false);
}
// Wait to be triggered by clock again.
wait();
}
}

void advios::adviosThread()
{

// Used in connecting the ctrl-port to the AXI4Lite-interface.
#pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0"
    variable=ctrl

// Init counter
sc_uint<4> cnt = 0;

while (1)
{
// if ctrl == 0, write value of counter to LEDs. Counter increments 1 every
    second.
// if ctrl != 0, write the value of the switches masked by the ctrl-signal.
int val_ctrl = ctrl.read().to_int();
int val_switches = inSwitch.read().to_int();
if (val_ctrl == 0)
{
outLeds.write(cnt);
// if all switches are engaged, clear LEDs and reset counter.
if (val_switches == 0x8)
{
outLeds.write(0);
```

```
cnt = 0;
}
else
{
// If the one-sec-pulse is high (which it is for only 1 cycle pr. second), increment
    the counter.
if (oneSecPulse == true)
{
//Increment counter, write to LEDs and wait for new clock.
cnt++;
outLeds.write(cnt);
}
}
}
else
{
// sc_uint<NUM_BITS> outLedVal = val_switches && val_ctrl;
outLeds.write((val_switches & val_ctrl));
}

// Wait to be triggered by clock again.
wait();
}
}
```

4.2 Test of C-code

In order to test the functionality of the C-code, a test-bench and a driver is made. The driver defines the interfaces to the ”Device Under Test”, along with a series of input stimuli and expected outputs. The test-bench sets up the test environment and executes the test-simulation.

The drivers and the test-bench are implemented in the following .h and .cpp-files

```
#ifndef _ADVIOS_DRIVER
#define _ADVIOS_DRIVER

#include <systemc.h>
//#include "Advios.h"

#define NUM_BITS 4

SC_MODULE(advios_driver) {

//Ports
sc_in <bool> clk;
```

```
sc_out <bool> reset;

sc_out<sc_uint<NUM_BITS> > ctrl;
sc_out<sc_uint<NUM_BITS> > outSwitch;
sc_in<sc_uint<NUM_BITS> > inLeds;

int retval;

//Process Declaration
void test();

//Constructor
SC_CTOR(advios_driver) : retval(-1) {

//Process Registration
SC_THREAD(test);
sensitive << clk.pos();
}
};

#endif
```

```
#include "advios_driver.h"

void advios_driver::test() {


//Variables
sc_uint<NUM_BITS> sw_test;
sc_uint<NUM_BITS> ctrl_test;
sc_uint<NUM_BITS> led_result;

//Initialization
sw_test = 0b1111;
ctrl_test = 0b0111;

// Reset at start, then make sure reset is false.
reset.write(true);
wait();
reset.write(false);
wait();

// Write stimuli to DUT
ctrl.write(ctrl_test);
outSwitch.write(sw_test);

// Wait for DUT to react to stimuli.
wait();
wait();
```

```

// Record output
led_result = inLeds.read();
wait();

// Compare output to expected value.
if (ctrl_test == led_result)
retval = 0;
else
retval = 1;
}
```

```
#include <systemc.h>
#include <stdio.h>

// if running RTL-simulation use generated RTL-wrapper
#ifdef __RTL_SIMULATION__
#include ”advios_rtl_wrapper.h”
#define advios advios_rtl_wrapper
// if not, the c-simulation is being run, and the c-version is used instead.
#else
#include ”advios.h”
#endif

#include ”advios_driver.h”

#define TRACE_FILE_NAME ”advios_trace”

int sc_main (int argc , char *argv[])
{
sc_report_handler::set_actions(”/IEEE_Std_1666/deprecated”,
    SC_DO_NOTHING);
sc_report_handler::set_actions( SC_ID_LOGIC_X_TO_BOOL_, SC_LOG);
sc_report_handler::set_actions(
    SC_ID_VECTOR_CONTAINS_LOGIC_VALUE_, SC_LOG);
sc_report_handler::set_actions( SC_ID_OBJECT_EXISTS_, SC_LOG);

sc_trace_file *tracefile;

// Test signals
sc_signal<bool> s_reset;
sc_signal<sc_uint<NUM_BITS> > s_switch;
sc_signal<sc_uint<NUM_BITS> > s_ctrl;
sc_signal<sc_uint<NUM_BITS> > s_leds;

// Create a 10ns period clock signal
sc_clock s_clk(”s_clk”, 10, SC_NS);
advios U_Advios(”advios”);
advios_driver U_advios_driver(”advios_driver”);
```

```

// Create tacefile
tracefile = sc_create_vcd_trace_file(TRACE_FILE_NAME);
if (!tracefile) cout << "Could not create trace file." << endl;

// Set resolution of trace file to be in 10 US
tracefile->set_time_unit(1, SC_NS);

// Trace signals
sc_trace(tracefile, s_clk,  "clock");
sc_trace(tracefile, s_reset, "reset");
sc_trace(tracefile, s_ctrl,  "ctrl");
sc_trace(tracefile, s_leds,  "leds");
sc_trace(tracefile, s_switch, "switch");

// Connect the DUT to the signals
U_Advios.clk(s_clk);
U_Advios.reset(s_reset);
U_Advios.ctrl(s_ctrl);
U_Advios.outLeds(s_leds);
U_Advios.inSwitch(s_switch);

// Connect the driver to the signals
U_advios_driver.clk(s_clk);
U_advios_driver.reset(s_reset);
U_advios_driver.inLeds(s_leds);
U_advios_driver.outSwitch(s_switch);
U_advios_driver.ctrl(s_ctrl);

// Simulate for 200
int end_time = 200;
std::cout << "INFO: Simulating" << std::endl;
// start simulation
sc_start(end_time, SC_NS);

// Check whether test passed or not
if (U_advios_driver.retval == 0) {
printf("Test passed !\n");
} else {
printf("Test failed !!!\n");
}

// Close trace file.
sc_close_vcd_trace_file(tracefile);
std::cout << TRACE_FILE_NAME << ".vcd" << std::endl;

return U_advios_driver.retval;
};
```

4.2.1 Result of C-simulation

Figure **??** show the result of running the C-simulation of the code. As seen, the test is passed, which indicates that the received result from the simulated matched the expected result.

4.3 Synthesis of C-code into VHDL

Using Vivado HLS, the C-code is synthesized into VHDL-code. From the figures **??** and **??** the performance and resource utilization of the synthesized code can be seen. It can be seen that there is a latency of a maximum of 5 clock cycles throughout the module. Furthermore, it can be seen that there is only used 170 Flip Flops and 187 Lookup Tables, which is virtually none of the existing resources of the board.

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../../Assignment_lab7/Assignment_lab7/tb_advios.cpp in debug mode
   Compiling ../../../../../Assignment_lab7/Assignment_lab7/Advios.cpp in debug mode
   Compiling ../../../../../Assignment_lab7/Assignment_lab7/advios_driver.cpp in debug
   Generating csim.exe
Note: VCD trace timescale unit is set by user to 1.000000e-009 sec.
INFO: Simulating
Test passed !
advios_trace.vcd
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Figur 4.4: Result of C-simulation

Performance Estimates

⊟ **Timing (ns)**

⊟ **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	6.16	1.25

⊟ **Latency (clock cycles)**

⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
0	5	1	6	none

⊟ **Detail**

⊞ **Instance**

⊞ **Loop**

Figur 4.5: Performance of synthesized VHDL code.

Utilization Estimates

⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	-	-	133	178
Memory	-	-	-	-
Multiplexer	-	-	-	9
Register	-	-	37	-
Total	0	0	170	187
Available	120	80	35200	17600
Utilization (%)	0	0	~0	1

Figur 4.6: Resource utilization of synthesized VHDL code.

4.4 RTL/C co-simulation

In order to verify that the synthesized VHDL-code functions as expected, an RTL/C co-simulation is carried out. Basically, a the test defined using the test bench and the driver is carried out, first on C-level as before, and then again using the newly synthesized VHDL-code. It then compares the result of the two tests, in order to verify that the functionality is the same.

As seen in figure **??**, it is seen that the simulation passes, indicating that the functionality of the synthesized VHDL-code matches that of the C-code.

4.5 Connect ctrl-port to AXI4Lite-interface

The ctrl-port is connected to the AXI4Lite-interface, simply by adding the following line of code

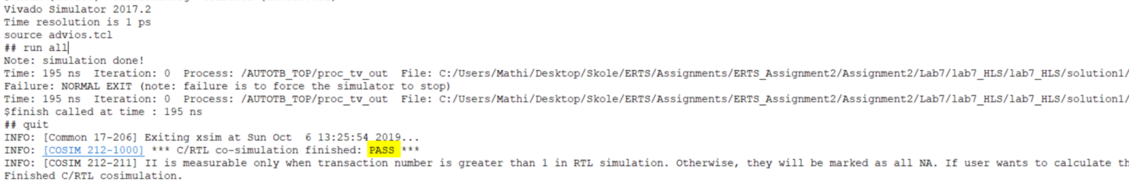
```
#pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=ctrl
```

at the top of the "main-thread of the advios-module, as seen in the file Advios.cpp. This makes it possible to connect an external signal to the block in order to send messages to the ctrl-port of the module.

4.6 Using the IP in an application

4.6.1 Generating the hardware

After having exported the RTL core from Vivado HLS, by simply pressing "Export RTL", the IP core can be used in a Vivado block design. A design was made using the IP, the ZYNQ processing system and an instance of the AXI4Lite-interface, as


source xsim.dir/advios/xsim_script.tcl
xsim {advios} -autoloadwcfg -tclbatch {advios.tcl}
Vivado Simulator 2017.2
Time resolution is 1 ps
source advios.tcl
run all
Note: simulation done!
Time: 195 ns Iteration: 0 Process: /AUTOTB_TOP/proc_tv_out File: C:/Users/Mathi/Desktop/Skole/ERTS/Assignments/ERTS_Assignment2/Assignment2/Lab7/lab7_HLS/lab7_HLS/solution1/
Failure: NORMAL EXIT (note: failure is to force the simulator to stop)
Time: 195 ns Iteration: 0 Process: /AUTOTB_TOP/proc_tv_out File: C:/Users/Mathi/Desktop/Skole/ERTS/Assignments/ERTS_Assignment2/Assignment2/Lab7/lab7_HLS/lab7_HLS/solution1/
$finish called at time : 195 ns
quit
INFO: [Common 17-206] Exiting xsim at Sun Oct 6 13:25:54 2019...
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
INFO: [COSIM 212-211] II is measurable only when transaction number is greater than 1 in RTL simulation. Otherwise, they will be marked as all NA. If user wants to calculate th
Finished C/RTL cosimulation.

Figur 4.7: Result of RTL/C co-simulation seen in figure **??**.

Next, the bitstream is generated, and the hardware is exported to Xilinx SDK, where an application for the hardware is written.

4.6.2 Writing the application

A simple application is written, which uses UART-communication to promt the user for a value to be used as the ctrl-value. Upon receiving a new value, the value is written to the ctrl-signal. As the rest of the functionality is handled by hardware, the program will function as expected, even though the "main-program is waiting for a new input from the user.

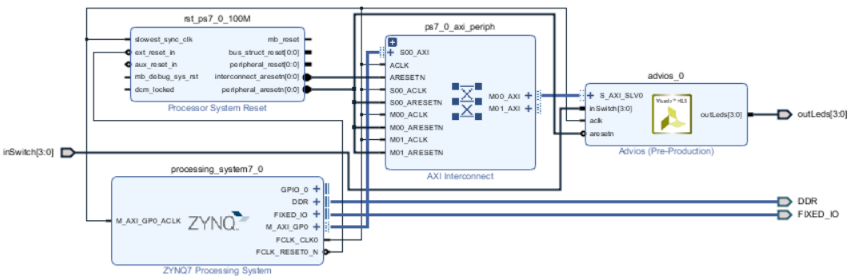
```
#include "xparameters.h"
#include "xadvios.h" // Include HAL for iosc driver
//=======================================================
void writeToCtrl(int);

// driver created as global, so it can be used in "helper"-funciton writeToCtrl.
XAdvios adviosHLS; // Create an instance of the advios driver

int main (void)
{
// Initialize the advios driver
if (XAdvios_Initialize(&adviosHLS, XPAR_ADVIOS_0_DEVICE_ID) !=
      XST_SUCCESS) return XST_FAILURE;

xil_printf("-- Start of the Program --\r\n");

// Loop: Prompt user for ctrl-value. If valid value, write to ctrl, else tell user.
while(1)
{
xil_printf("\r\n");
xil_printf("------------------------------\r\n");
```



Figur 4.8: Block design using the exported IP.

```
xil_printf(”\r\n”);
xil_printf(”Enter ctrl-value\r\n”);

int userInput = inbyte()-48;
if ((userInput <0 )||(userInput > 15))
{
xil_printf(”Value must be between 0 and 15\r\n”);
}
else
{
writeToCtrl(userInput);
}
}
}


void writeToCtrl(int val)
{
// Writing 0xff to the ctrl register of the iosc IP core
XAdvios_SetCtrl(&adviosHLS, val);
}
```

4.6.3 Results

Upon visual inspection, the following functionality is confirmed:

- **ctrl = 0:** The LEDs are used to count, incrementing with the value of 1 every second.
- **ctrl = 1-15:** The LEDs show the value of the number corresponding to the value of ctrl, provided that all switches are in the ”on-position. If a switch is in the ”off-position, the corresponding LED is turned off.

Thus, the functionality of the system is as stated in the requirements.

4.7 Summary

In this exercise, an IP has been written in C using SystemC, synthesized using High Level Synthesis, and integrated in a design. It has taught us the general procedure for creating custom IPs at a relatively high level of abstraction, in order to utilize them at a much lower level of abstraction.