ERTS - Assignment 2

|  |  |
| --- | --- |
| Name | Study number |
| Rasmus B. Langhoff | 201506966 |
| Mads M. Beck | 201508319 |
| Martin W. Kjær | 201509268 |

# Assignment 3

In the first assignment, the goal is to create a C-application command language interpreter, which is controlled via the USB-UART interface. At this stage, the interpreter should only have the following to commands:

1. Writing ‘1’ to the UART results in the onboard switches being able to display the binary value their position corresponds to on the four green LED’s above, that is, values from 0-15.
2. Writing ‘2’ to the UART results in the board using the ScuTimer to display a binary value on green LED’s, which increments each second.

Both of these uses the LED IP core. The implementation of the above-mentioned functionality can be seen below in figure 1 and 2:

#define ONE\_SECOND 325000000

//====================================================

int main (void)

{

XGpio dip, push;

int i, psb\_check, dip\_check, flag;

char value, skip;

int32\_t Status, timeCount = 0;

// PS Timer related definitions

XScuTimer Timer;

XScuTimer\_Config \*ConfigPtr;

XScuTimer \*TimerInstancePtr = &Timer;

// Initialize the timer

ConfigPtr = XScuTimer\_LookupConfig (XPAR\_PS7\_SCUTIMER\_0\_DEVICE\_ID);

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);

// Start the timer

XScuTimer\_Start(TimerInstancePtr);

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

xil\_printf("Enter choice: 1 (SW->Leds), 2 (Timer->Leds), 3 (Matrix), 4 (Exit)\r\n");

XGpio\_Initialize(&dip, XPAR\_SWITCHES\_DEVICE\_ID);

XGpio\_SetDataDirection(&dip, 1, 0xffffffff);

XGpio\_Initialize(&push, XPAR\_BUTTONS\_DEVICE\_ID);

XGpio\_SetDataDirection(&push, 1, 0xffffffff);

xil\_printf("CMD:> ");

/\* Read an input value from the console. \*/

value = inbyte();

skip = inbyte(); //CR

skip = inbyte(); //LF

Figure 1 Code snippet of initialization in assignment 3

while (flag)

{

switch (value)

{

case '1':

dip\_check = XGpio\_DiscreteRead(&dip, 1);

// output dip switches value on LED\_ip device

LED\_IP\_mWriteReg(XPAR\_LED\_IP\_S\_AXI\_BASEADDR, 0, dip\_check);

break;

case '2':

if(XScuTimer\_IsExpired(TimerInstancePtr)) {

XScuTimer\_ClearInterruptStatus(TimerInstancePtr);

timeCount = timeCount + 1;

}

// output dip switches value on LED\_ip device

LED\_IP\_mWriteReg(XPAR\_LED\_IP\_S\_AXI\_BASEADDR, 0, timeCount);

default:

flag = 0;

break;

}

for (i=0; i<9999999; i++);

}  
xil\_printf("Program will now terminate \r\n");  
}

Figure 2 Code snippet of program loop in assignment 3

In figure 1, the initialization of the program is shown, where both variables are initialized but also the timer is initialized and setup. The main program loop is shown in figure 2, where the two cases are defined according to the numeric list mentioned before the two figures. The first case simply uses the method XGpio\_DiscreteRead() to read from the switches and then uses the method LED\_IP\_mWriteReg() to output the binary values to the LED’s. The second case instead uses the ScuTimer to display the binary value on the LED’s. Thus, we need to make sure check whether it has expired and clear the interrupt status bit, while also incrementing the current time count which is displayed to the LED’s. The default case serves as the exit case, and ceases the program execution.

# Assignment 4

In this assignment, the goal is to extend the functionality of the previous program. This entails adding a new case for the interpreter which is performing matrix multiplication in software.

In the exercise description, a proposed interface is described along with the matrices which are to be used for the program. As these are constant, the implementation is simplified as the size and setup is thus trivial. The full implementation is shown below in figure 3:

typedef union{

unsigned char comp[4];

unsigned int vect;

} vectorType;

typedef vectorType VectorArray[4];

void multiMatrixSoft(VectorArray A, VectorArray B, VectorArray P)

{

int i, j;

for (i = 0; i < 4; i++)

{

for (j = 0; j < 4; j++)

{

P[i].comp[j] = A[i].comp[0]\*B[j].comp[0] + A[i].comp[1] \* B[j].comp[1] + A[i].comp[2] \* B[j].comp[2] + A[i].comp[3] \* B[j].comp[3];

}

}

}

void setInputMatrices(VectorArray A, VectorArray B)

{

//Set input matrix A:

for(int i = 0; i<4;i++)

{

for (int j = 0; j<4;j++)

{

A[i].comp[j] = (j+1)+(i\*4);

}

}

//Set input for matrix b:

for(int i = 0; i<4;i++)

{

for (int j = 0; j<4;j++)

{

B[i].comp[j] = i+1;

}

}

}

void displayMatrix(VectorArray input)

{

for(int i = 0; i<4;i++)

{

for (int j = 0; j<4;j++)

{

xil\_printf("%d ",input[i].comp[j]);

}

xil\_printf("\n");

}

}

Figure 3 Code snippet of implementation of assignment 4

As can be seen, the setup of the matrices is constant along with the size used to iterate through the for-loops in all the functions, as the matrices are constant. The matrix multiplication implemented in the method multiMatrixSoft() is implemented using two nested for-loops which go through all combinations of the result to calculate the sum of products of each element.

In the program loop, the implementation is done as shown in the following figure, where a new case has been added to the switch statements in the interpreter:

case '3':

setInputMatrices(aInst,bTInst);

displayMatrix(aInst);

displayMatrix(bTInst);

XScuTimer\_RestartTimer(TimerInstancePtr);

xil\_printf("-- Clock count before matrix multiplication %i \r\n", ONE\_SECOND- XScuTimer\_GetCounterValue(TimerInstancePtr));

multiMatrixSoft(aInst,bTInst,pInst);

xil\_printf("-- Clock count after matrix multiplication %i \r\n", ONE\_SECOND-XScuTimer\_GetCounterValue(TimerInstancePtr));

displayMatrix(pInst);

flag = 0;

break;

Figure 4 Code snippet of matrix multiplication case in assignment 4

In figure 4, it is seen that first the matrices are setup, then displayed, and finally calculated. However, before the calculation starts, the timer is reset before it is read after the multiplication has been performed, as the program should be able to count the clock cycles of the matrix multiplication. A terminal output of this case’s execution is shown below in figure 5:

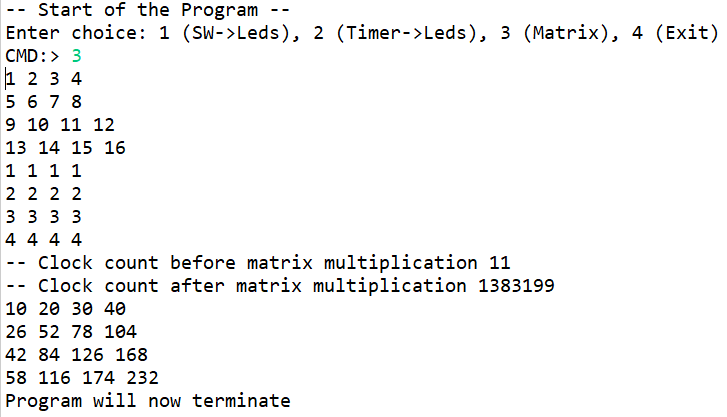


Figure 5 Terminal output from assignment 4

In figure 5 it is shown, that the two matrices are first displayed, then the clock count before the matrix multiplication is displayed, afterwards the clock count after the matrix multiplication, and lastly the result is shown. It should be noted, that the print method has not been timed, so the actual clock count is not known precisely, but this serves as a good estimation.

# Assignment 5

In this third exercise, the goal is to implement a matrix multiplication IP core given by the exercise. This IP core uses hardware to perform the matrix multiplication in contrast to before, where the matrix multiplication is performed purely in software.

First the IP core is added to the block design in Vivado. The final block diagram can be seen below in figure 6:

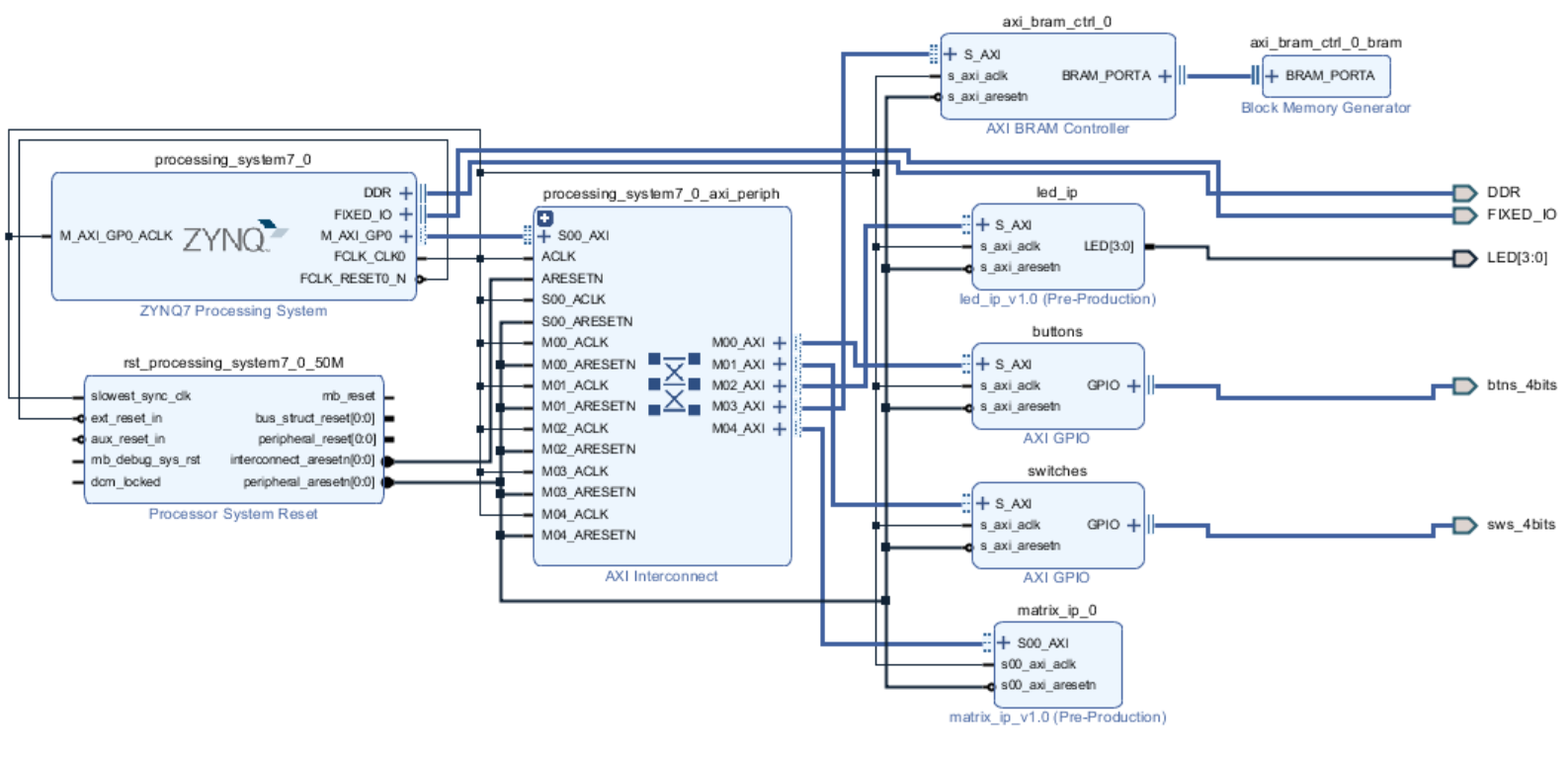


Figure 6 Block diagram of system in assignment 5

Afterwards, the matrix implementation shown in figure 3 is extended with a new method, which uses the new IP. This implementation is shown below in figure 7:

void multiMatrixHard(VectorArray A, VectorArray B, VectorArray P)

{

int i, j;

for (i = 0; i < 4; i++)

{

for (j = 0; j < 4; j++)

{

Xil\_Out32(XPAR\_MATRIX\_IP\_0\_S00\_AXI\_BASEADDR + MATRIX\_IP\_S00\_AXI\_SLV\_REG0\_OFFSET, A[i].vect);

Xil\_Out32(XPAR\_MATRIX\_IP\_0\_S00\_AXI\_BASEADDR + MATRIX\_IP\_S00\_AXI\_SLV\_REG1\_OFFSET, B[j].vect);

P[i].comp[j] = Xil\_In32(XPAR\_MATRIX\_IP\_0\_S00\_AXI\_BASEADDR + MATRIX\_IP\_S00\_AXI\_SLV\_REG2\_OFFSET);

}

}

}

Figure 7 Code snippet of method using the matrix IP core for matrix multiplication

Finally, case 3 in the interpreter has been extended as is shown below in figure 8:

case '3':

setInputMatrices(aInst,bTInst);

displayMatrix(aInst);

displayMatrix(bTInst);

XScuTimer\_RestartTimer(TimerInstancePtr);

xil\_printf("-- Clock count before hard matrix multiplication %i \r\n", ONE\_SECOND-XScuTimer\_GetCounterValue(TimerInstancePtr));

multiMatrixHard(aInst,bTInst,pInst);

xil\_printf("-- Clock count after hard matrix multiplication %i \r\n", ONE\_SECOND-XScuTimer\_GetCounterValue(TimerInstancePtr));

XScuTimer\_RestartTimer(TimerInstancePtr);

xil\_printf("-- Clock count before matrix multiplication %i \r\n", ONE\_SECOND- XScuTimer\_GetCounterValue(TimerInstancePtr));

multiMatrixSoft(aInst,bTInst,pInst);

xil\_printf("-- Clock count after matrix multiplication %i \r\n", ONE\_SECOND-XScuTimer\_GetCounterValue(TimerInstancePtr));

displayMatrix(pInst);

flag = 0;

break;

Figure 8 Code snippet of updated case 3 in assignment 5

A terminal output of figure 9 is shown below in figure 9:

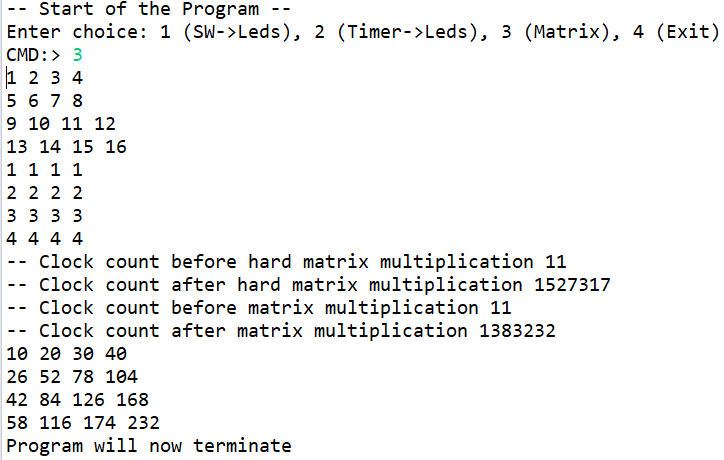


Figure 9 Terminal output from assignment 5

As can be seen in figure 9, the clock count lower for the hard matrix multiplication than the soft matrix multiplication (as they count down from the define: #define ONE\_SECOND 325000000, as would be expected.

# Assignment 7

In the last assignment, the goal is to write a IP core using SystemC, synthesize it using Vivado HLS, and then program it to the Zybo board with a program that verifies the functionality of the IP core.

Specifically, the IP core to the implemented is an advanced input/output (ADVIOS) controller. Depending on a given control value, the interface will either increment the LED’s every second (ctrl value = 0x0), or it will mask the value of the control value and the switches and output this to the LED’s (ctrl value = 0x1 -0x0f). Also, if the ctrl value is 0x8, then the output LED’s are cleared. The SystemC implementation is based upon two threads, one handling the functionality and the other handling the implementation. These are shown below in figure 10 and 11:

void adviosc::adviosThread() {

//Group ports into AXI4 slave slv0

#pragma HLS resource core=AXI4LiteS metadata="-bus\_bundle slv0" variable=ctrl

//Initialization

wait();

unsigned char count = 0;

// Process the data

while (true) {

// Wait for start

wait();

if (inSwitch.read() == 0x8)

{

count = 0;

}

if (ctrl.read() == 0)

{

if (timer.read() == 1)

{

outLeds.write(++count);

}

else

{

outLeds.write(count);

}

}

else

{

switchs = inSwitch.read() & ctrl.read();

outLeds.write(switchs);

}

}

}

Figure 10 Code snippet of adviosThread in assignment 7

void adviosc::timeThread()

{

wait();

int count = 0;

timer.write(0);

while (true)

{

wait();

//if (count < 5000)

if(count < 1000000)

{

count++;

timer.write(0);

}

else

{

count = 0;

timer.write(1);

}

}

}

Figure 11 Code snippet of timeThread in assignment 7

The SystemC implementation has been run and a wavefile has been generated to verify the functionality before it was imported to Vivado HLS. The waveform can be seen below, where the timescale and timing has been adjusted, so the simulation would not take as long to perform. Attempts have been made to generate a wavefile using the correct timing, however, this generated a wavefile with a size of 8 Gb, and GTKWave was not able to open it. The wave simulation is shown below in figure 12:

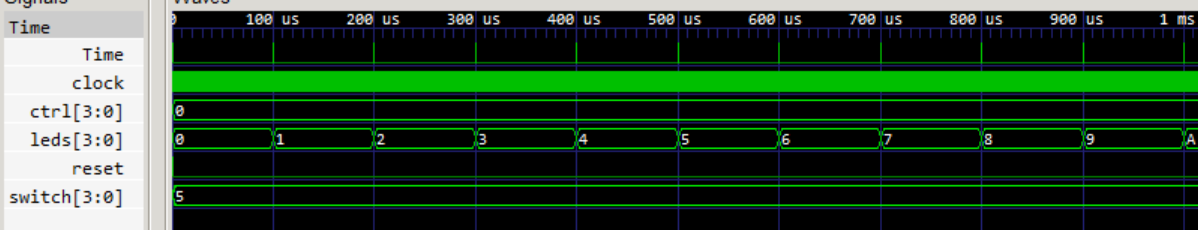


Figure 12 Waveform of the SystemC implementation with ctrl value = 0x0.

Afterwards, the SystemC implementation was imported in Vivado HLS, where it was simulated and synthesized. Below part of the synthesis report for the IP has been shown in figure 13:

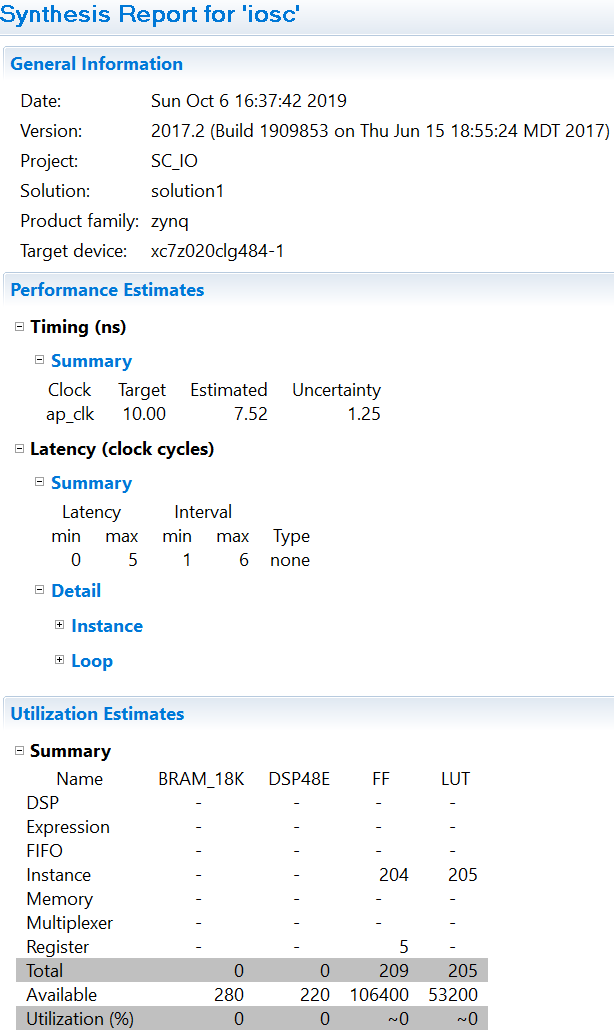


Figure 13 Synthesis report for advios core

Afterwards, the RTL core is exported and can now be added in a Vivado project by adding it to the IP catalog. The IP core was then added as has been done previously, and the input switches and output LED’s was made external. The final block diagram is shown below in figure 14:

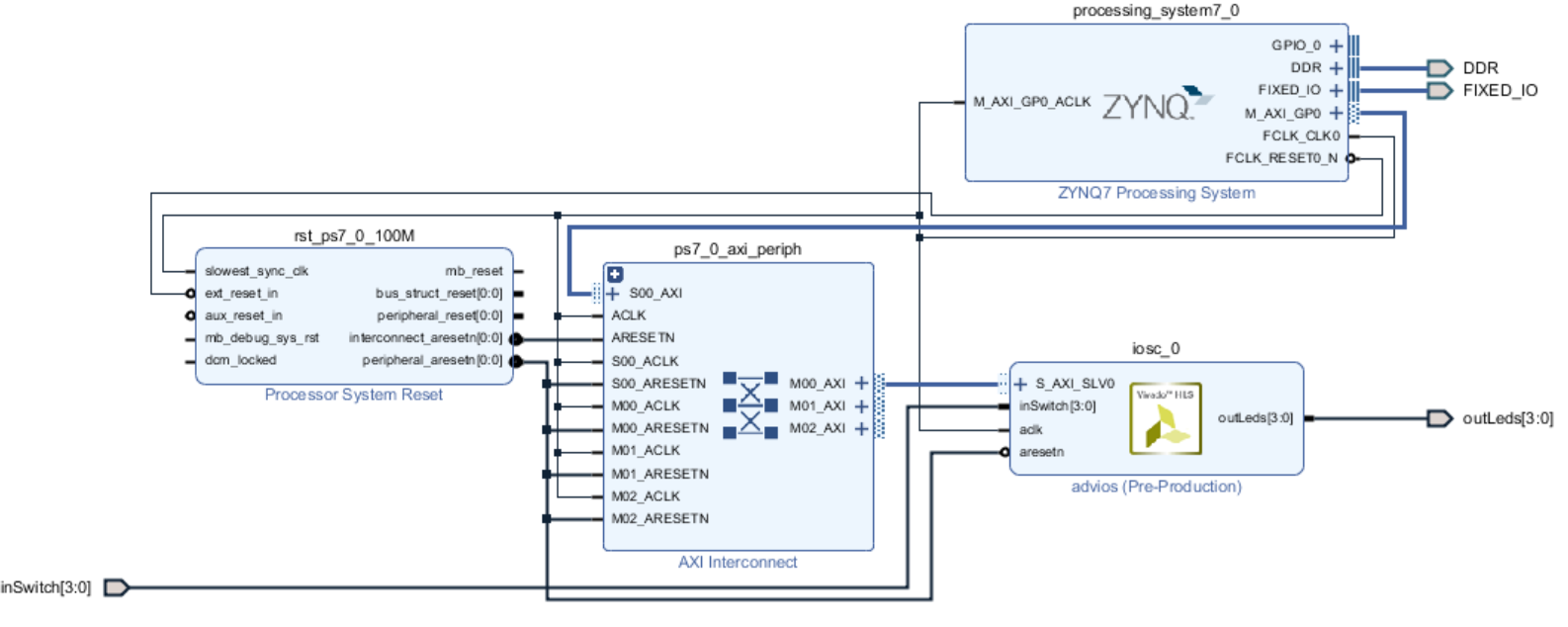


Figure 14 System block diagram from assignment 7

Finally, a program has been written to verify the IP core functionality. This is shown below in figure 15:

int main(void)

{

XIosc ioscHLS; // Create an instance of the iosc driver

u32 result;

// Initialize the iosc driver

if (XIosc\_Initialize(&ioscHLS, XPAR\_IOSC\_0\_DEVICE\_ID) != XST\_SUCCESS) return XST\_FAILURE;

// Writing 0xf to the ctrl register of the iosc IP core

XIosc\_SetCtrl(&ioscHLS, 0xF);

// Reading from the ctrl register of the iosc IP core

result = XIosc\_GetCtrl(&ioscHLS);

// Verifying ctrl value

if (result == 0xf)

xil\_printf("Successful \r\n");

else

xil\_printf("Unsuccessful \r\n");

// Uncomment for timing functionality

//XIosc\_SetCtrl(&ioscHLS, 0x0);

for (;;);

}

Figure 15 Program to verify IP core from assignment 7

When the above from figure 15 is written to the Zybo board, it results in figure 16 shown below, which behaves as is expected:



Figure 16 Zybo running case 1 (switch->LED's)