TIHSC Hand-In 2

Journal over exercises 3,4,5&7

Name:

Kasper Nielsen | 10731

Martin Dawood |10786

Date:

19/5 - 2015

Contents

1 Exercise 3

1.1 Introduction

In this exercise the objective is to create a console application to run on the Hard-Core ARM processor of the Xilinx Zynq Programmable System (PS) on the ZYBO board.

The application has two core functionalities.

If it gets a '1' as an input it is to read the value of the four DIP switches on the ZYBO board, and set the on-board LEDs accordingly.
If it gets a '2' as an input it is supposed to count from 0 to 15 in binary on the on-board LEDs with an interval of 1 second.

This exercise will teach us how to setup a simple processing system on the PS, how to export the hardware system for use with the SDK and how write both hardware and software to the PS. Further this exercise will give insight into the use of the on-board USB-UART and the build-in *XscuTimer*. It will also show the learnings of earlier exercises where HW GPIO blocks where created in order to use the LEDs and switches as Memory-Mapped (MM) devices.

1.2 Implementation

The HW system used here looks as shown in Figure 1.1. The BRAM Memory and controller that is shown at the bottom is left over from a previous exercise and is not used.

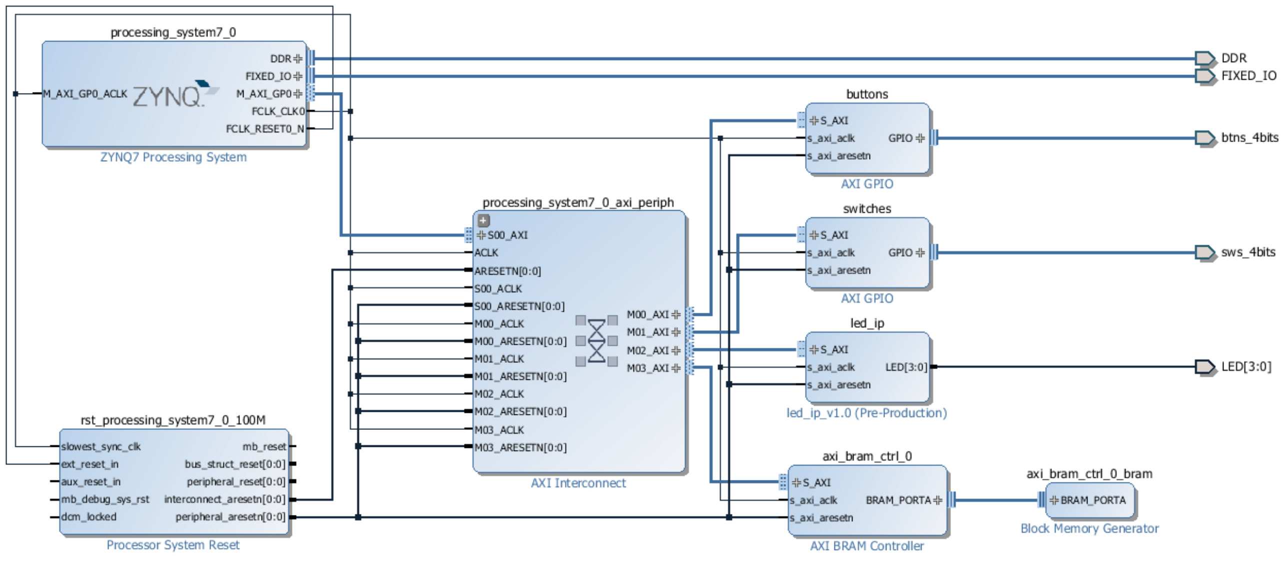


Figure 1.1 - Block Design for Exercise 3

The console input was implemented in a simple fashion using the *inbyte()* function to read characters from the console (See **Fejl! Henvisningskilde ikke fundet.**) . The function is run once to read the desired input, and twice more to discard of the following <CR> and <LF> characters. This crude implementation means that the system will behave unpredictably if presented with multi character inputs.

```
while (1) {
   // Start console interpreter
   xil_printf("CMD:> ");
   value = inbyte(); // Read char from console
   inbyte(); // skip CR
   inbyte(); //skip LF
```

Snippet 1.1 - Console input

The first functionality is implemented by simply reading the MM switches as a register and writing to the MM LEDs as a register. A console output showing the read switch value is also produced (See Snippet 1.2).

```
switch (value) {
   case '1':
      // Read DIP's and display
      dip_check = XGpio_DiscreteRead(&dip, 1);
      LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, dip_check);
      xil_printf("LEDs set to: %d\r\n", dip_check);
      break
```

Snippet 1.2 - DIP switches to LEDs handling

The second functionality is a bit more involved. First the timer must be created and initialized as shown in Snippet 1.4.

The program then runs a 1 second timer in a loop and updates the LEDs, when the second functionality is enabled (See Snippet 1.5). Although not expressly required in the exercise description, the program is set to *break* the loop if a push button is activated. This is implemented to stop the program from becoming stuck in the timer loop, requiring a HW reset to escape.

Finally a case for handling invalid single character inputs is implemented (See Snippet 1.3).

```
default :
   xil_printf("Invalid input: %c\n\r",value);
```

Snippet 1.3 - Invalid charater handling

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

// PS Timer related definitions
XScuTimer_Config *ConfigPtr;
XScuTimer *TimerInstancePtr = &Timer;
ConfigPtr = XScuTimer_LookupConfig(XPAR_PS7_SCUTIMER_0_DEVICE_ID);
int 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)
```

Snippet 1.4 - Timer creation and initialization

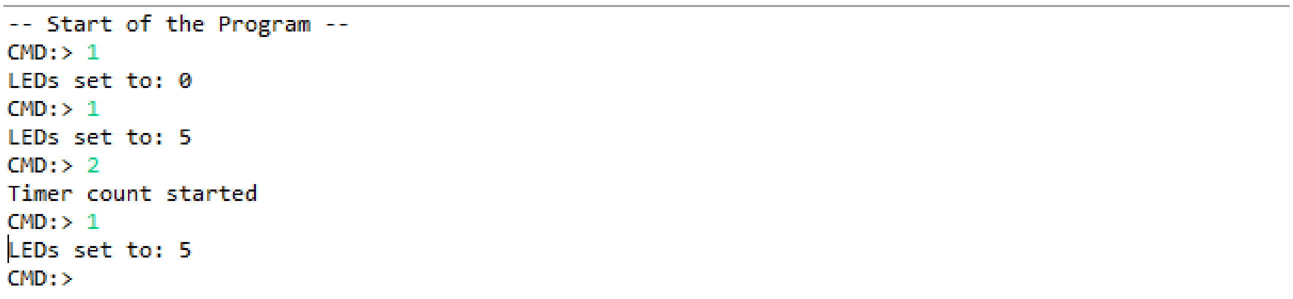
```c
case '2':
   // Start the timer
   xil_printf("Timer count started\r\n");
   XScuTimer_Start(TimerInstancePtr);

   count = 0;

   while (1) {
      // Check timer expired
      if (XScuTimer_IsExpired(TimerInstancePtr)) {
         // clear status bit
         XScuTimer_ClearInterruptStatus(TimerInstancePtr);
         // Code here
         count = (count + 1) % 0x10;
         LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, count);
      }

      // If a Button is pushed go back to interpreter
      if(XGpio_DiscreteRead(&push, 1)){
         break;
      }

   }
   // Stop the timer
   XScuTimer_Stop(TimerInstancePtr);
   Break;
```

Snippet 1.5 - Timer counter handling

1.3 Results

Figure 1.2 shows the results of running the console application on the ZYBO Board. It is the result of the following sequence of inputs:

1. Set switches to *0b0000*
2. Input ‘1’ ↵
3. Set switches to *0b0101*
4. Input ‘1’ ↵
5. Input ‘2’ ↵
6. Waiting and observing the LEDs count up at regular intervals
7. Pressing *BTN0*
8. Input ‘1’ ↵



```
-- Start of the Program --
CMD:> 1
LEDs set to: 0
CMD:> 1
LEDs set to: 5
CMD:> 2
Timer count started
CMD:> 1
LEDs set to: 5
CMD:>
```

Figure 1.2 - Results of running console application

As is visible from Figure 1.2, the application behaves as intended.

1.4 Discussion

As was intended this exercise was a good stepping stone into the development of SW applications on custom HW. The use of MM HW devices and build-in timers and UART made the exercise relative easy to handle.

2 Exercise 4

2.1 Introduction

In this exercise the objective is to create a console application to run on the Hard-Core ARM processor of the Xilinx Zynq Programmable System (PS) on the ZYBO board.

This console application is supposed to carry out a multiplication of two 4x4 matrices in software, time the calculation and display the results.

There is not much new in this exercise, than the need to think algorithmically about linear algebra, nut it serves as a baseline for comparing with HW calculation in Exercise 5.

2.2 Implementation

The HW system used in this exercise is the same as the one used in Exercise 3.

In this exercise two matrices are created using the data structure *vectorArray*. Creation of the matrices is done in the function *setInputMatrices()* (See Snippet 2.1).

```
void setInputMatrices(VectorArray A, VectorArray B) {

   int row, col, k;
   k = 1;
   for (row = 0; row < MSIZE; ++row) {
      for (col = 0; col < MSIZE; ++col) {
         A[row].comp[col] = k++;
      }
   }

   k = 1;
   for (col = 0; col < MSIZE; ++col) {
      for (row = 0; row < MSIZE; ++row) {
         B[row].comp[col] = k;
      }
      k++;
   }
}
```

Snippet 2.1 - Implementation of setInputMatrices()

This creates to matrices A and B and assigns them to them to the *vectorArrays* pointed to by the input parameters.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix}$$

Another function, d*isplayMatrix()*, were implemented which simply displays a 4x4 Matrix out to the terminal using the USB-UART interface (See Snippet 2.2).

```
void displayMatrix(VectorArray input) {
   int row, col;

   for (row = 0; row < MSIZE; ++row) {
      xil_printf("[ ");
      for (col = 0; col < MSIZE; ++col) {
         xil_printf("%d ", input[row].comp[col]);
      }
      xil_printf("] \r\n");
   }
   xil_printf("\r\n");
};
```

Snippet 2.2 - Implementation of displayMatrix()

The matrix multiplication is done in the function *multiMatrixSoft ().* The computation is implemented as three nested for loops; two to iterate over each combination of rows and columns in the resulting matrix, and one to do the Multiplication and Accumulation (MAC) operations of the internal vector multiplication.

The implementation is shown in **Fejl! Henvisningskilde ikke fundet.** below.

```
void multiMatrixSoft(VectorArray A, VectorArray B, VectorArray P){
   int row, col, k;

   for (row = 0; row < MSIZE; ++row) {
      for (col = 0; col < MSIZE; ++col) {
         P[row].comp[col] = 0;
         for(k = 0; k<MSIZE; ++k) {
            P[row].comp[col] =
                  P[row].comp[col] +
                  (A[row].comp[k] * B[k].comp[col]);
         }
      }
   }
}
```

Snippet 2.3 - Implementationen of matrixMultSoft()

The result of the computation is then printed out to the terminal using DisplayMatrix.

2.3 Discussion and Results

The presentation and discussion of results is differed to Section 3 because the comparison of this and the next exercise will be more meaningful than looking at this in isolation.

3 Exercise 5

3.1 Introduction

The objective of this exercise is to attempt to improve the performance of the matrix multiplication of Exercise 4 by way of hardware acceleration. This is done by performing the innermost loop, essentially an inner product of two vectors, in hardware logic in the Programmable Logic (PL) section of the Zynq processor.

3.2 Implementation

The HW inner product implementation is done by wrapping a piece of supplied VHDL code in an IP-core. The VHDL code, aside from performing calculation, implements an AXI-Memory mapped interface. This allows a SW application running on the hard-core processor to invoke the operation by simply writing the input parameters to MM registers in the IP-core and read the result in another MM register.

The values in the matrices are 8-bit unsigned chars, so a rows/columns are 4x8 bit = 32 bit words. These can be written to the MM registers in a single *Xil_Out32()* command.

```
void multiMatrixHard(VectorArray A, VectorArray B, VectorArray P) {
   int row, col;

   for (row = 0; row < MSIZE; ++row) {
      for (col = 0; col < MSIZE; ++col) {
         P[row].comp[col] = 0;
         Xil_Out32(
               XPAR_MATRIX_IP_0_S_AXI_BASEADDR + MATRIX_IP_S_AXI_SLV_REG0_OFFSET,
               A[row].vect
               );
         Xil_Out32(
               XPAR_MATRIX_IP_0_S_AXI_BASEADDR + MATRIX_IP_S_AXI_SLV_REG1_OFFSET,
               B[col].vect
               );
         P[row].comp[col] = Xil_In32(
               XPAR_MATRIX_IP_0_S_AXI_BASEADDR + MATRIX_IP_S_AXI_SLV_REG2_OFFSET
               );
      }
   }
}
```

Snippet 3.1 - Implementation of multiMatrixHard()

The HW Block Design used in this exercise look as shown in Figure 3.1. Notice the *matrix_ip_0* block at the bottom right; this is where HW acceleration is performed.

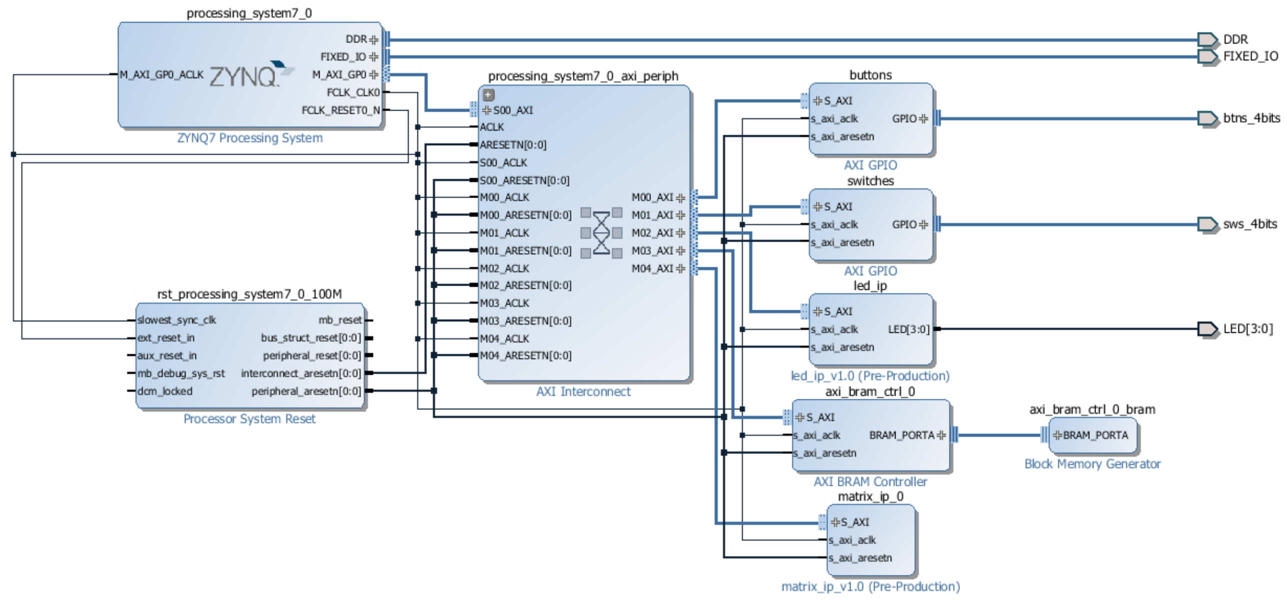
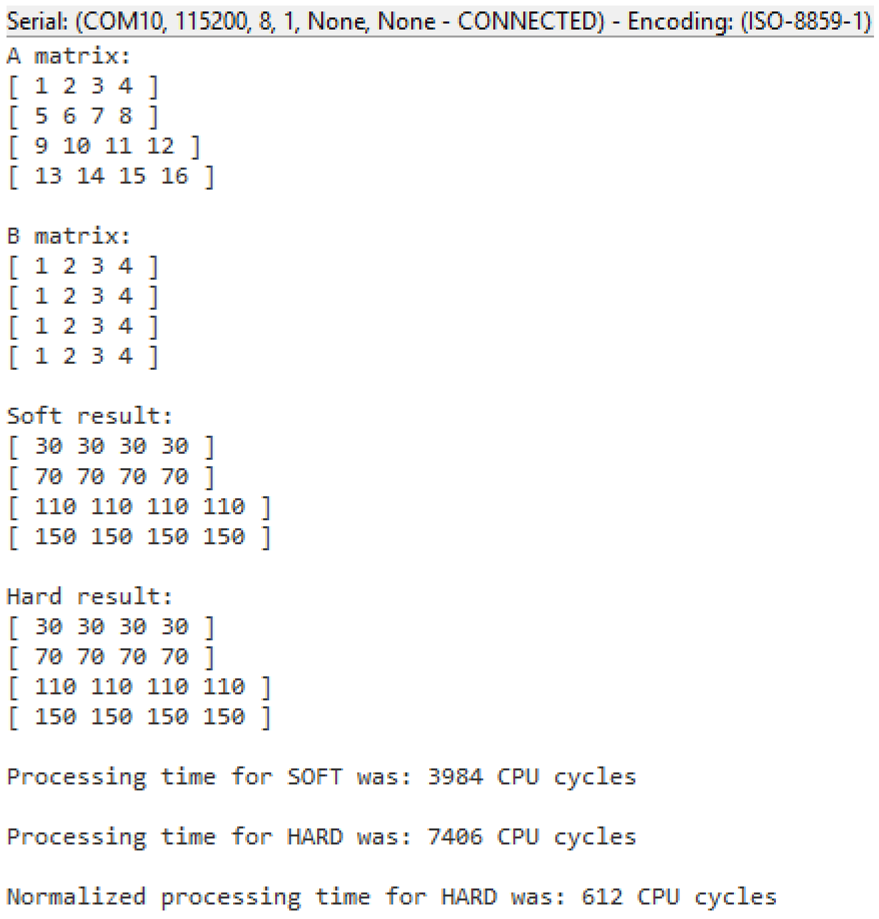


Figure 3.1 - Exercise 5 HW Block Design

3.3 Results

Figure 3.2 below shows the results of exercises 4 and 5. In this we see that the two implementations agree on the result. Further we see that the HW accelerated implementation is slower (7406 CPU cycles) than the SW implementation (3984 CPU cycles).



```
Serial: (COM10, 115200, 8, 1, None, None - CONNECTED) - Encoding: (ISO-8859-1)
A matrix:
[ 1 2 3 4 ]
[ 5 6 7 8 ]
[ 9 10 11 12 ]
[ 13 14 15 16 ]

B matrix:
[ 1 2 3 4 ]
[ 1 2 3 4 ]
[ 1 2 3 4 ]
[ 1 2 3 4 ]

Soft result:
[ 30 30 30 30 ]
[ 70 70 70 70 ]
[ 110 110 110 110 ]
[ 150 150 150 150 ]

Hard result:
[ 30 30 30 30 ]
[ 70 70 70 70 ]
[ 110 110 110 110 ]
[ 150 150 150 150 ]

Processing time for SOFT was: 3984 CPU cycles

Processing time for HARD was: 7406 CPU cycles

Normalized processing time for HARD was: 612 CPU cycles
```

Figure 3.2 – Console output showing the results of Exercises 4-5

For the sake of comparison a crude estimate of a normalized processing time has been made according to the formula:

$$\tau_{normalized} = \frac{f_{PL}}{f_{CPU}} * \tau_{HARD} = \frac{100\ MHz}{650\ MHz} * 7406\ cycles = 612\ cycles$$

3.4 Discussion

Somewhat counterintuitively the HW accelerated implementation is the slower of the two. This is initially surprising, as one expects the HW to be much faster. The cause for this slowdown is however apparent when one considers the different clock speeds of hard-processor and PL. Even though the HW computation is efficient, the SW runs at a rate 6.5 times faster and does not have any outside communication overhead.

For comparison a crude normalization of the two results have been made, showing that at equal clock rates the HW accelerated implementation may very well be faster. This is however speculation, as this does not take into account wait-times and cross clock problems.

Also, in both implementations there is great room for improvement. For the HW there is great potential in pipelining the entire calculation instead of only the inner loop. This would however limit flexibility in input/output matrix size.

As for the SW, the hard-core ARM processor of the Zynq PS already possesses an ARM NEON SIMD (Single Instruction Multiple Data) vector processer that could help to accelerate processing.

4 Exercise 7

4.1 Introduction

The objective of this exercise is to learn how to use systemC in writing IP cores controlled by software using the AXI4 Lite interface.

The IP core called ADVIOS (Advanced I/O System) will handle switch input and LEDs independently from the SW. A MM control register will allow the SW to control how the ADVIOS behaves. If the control register is cleared the ADVIOS will count binary on the LEDs, incrementing every 1 second. Otherwise the ADVIOS will mask the switch input with the control value and display the result on the LEDs. Finally if the control value is cleared and the switches are set to 0x08 the LEDs are cleared.

4.2 Implementation

The first step in this exercise was to implement the ADVIOS controller in systemC. One module was implemented called "ios", which reflects the functionality of the ADVIOS controller. The ios module contains of two threads, `iosThread` and `timerThread`. The `timerThread` increments a counter every clock cycle and when one second is reached, a flag *second* is set (See Snippet 4.1). Simultaneous the `iosThread` reads the switches input and the *ctrl* input (See Snippet 4.2). If the *ctrl* input is zero and the switches are set to 0x08, the LEDs are simply cleared and if second flag is set the output LEDs sets to a counter value, which increments every time the second flag is set. Otherwise the `iosThread` mask the input values of the switches and ctrl and sets the LEDs corresponding to the result of the mask.

After passing the test bench the code is synthesized to RTL and packaged is as a IP core.

```
void iosc::timerThread() {
    while (1) {
        wait();

        if (timerCount++ == ONE_SECOND) {
            second = true;
            timerCount = 0;
        }
        else {
            second = false;
        }
    }
}
```

Snippet 4.1 - Implementaion of timerThread()

```
void iosc::iosThread() {
    //Group ports into AXI4 slave slv0
    #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=ctrl

    //Initialization
    wait();

    // Process the data
    while (true) {
        // Wait for start
        wait();

        switchs_in = inSwitch.read();
        ctrl_in = ctrl.read();

        if (ctrl_in == 0) {
            if (switchs_in == 0x08) {
                outLeds.write(0); // Clear LEDs
            }

            if (second == true) {
                outLeds.write(second_count++);
            }
        }

        else {
            switchs = switchs_in & ctrl_in; // Mask
            outLeds.write(switchs);         // Write
        }
    }
}
```

Snippet 4.2 - Implementation of iosThread()

The next step was to create a new Vivado project, which connect the implemented ADVIOS IP core in to LEDs and switches on the ZYBO board (See Figure 4.1).

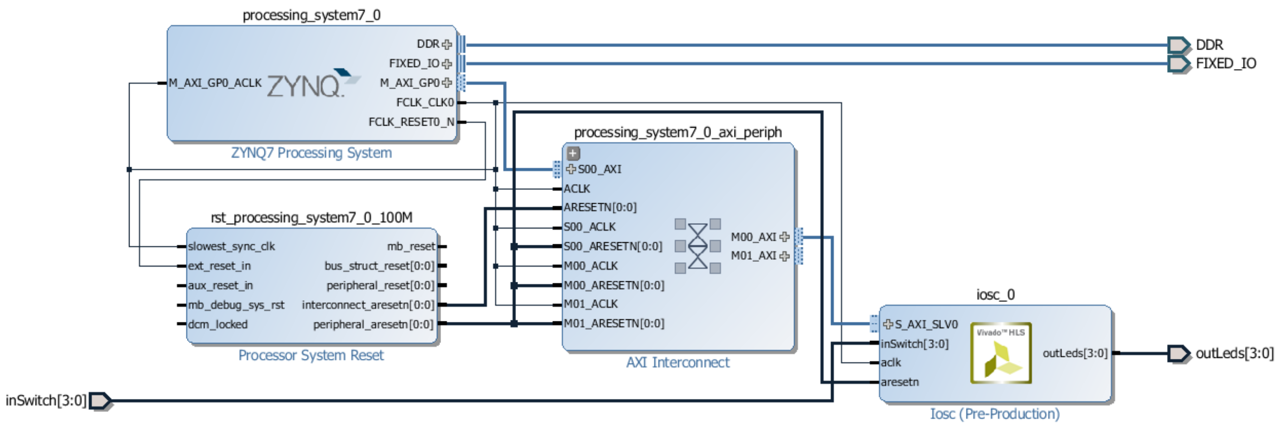


Figure 4.1 - Exercise 7 Block Design

Then the bitstream and generated and the HW is exported so a test application can be made using the SDK. This application, implemented as a simple console application, takes an input (a ctrl value) from the user and writes to the ADVIOS *ctrl* register (See Snippet 4.3). This way the behavior can be verified.

```c
#include "xparameters.h"
#include "xiosc.h"

int main(void) {
    char ctrlVal, value;
    XIosc IOsys;

    // Initialize the iosc driver
    if (XIosc_Initialize(&IOsys, XPAR_IOSC_0_DEVICE_ID) != XST_SUCCESS) {
        return XST_FAILURE;
    }

    while (1) {
        xil_printf("CMD:> ");
        value = inbyte(); // Read char from console
        inbyte(); // skip CR
        inbyte(); //skip LF

        if (value >= 'A' && value <= 'F') {
            ctrlVal = value - 'A' + 10;
        }

        else if (value >= 'a' && value <= 'f') {
            ctrlVal = value - 'a' + 10;
        }

        else if (value >= '0' && value <= '9'){
            ctrlVal = value - '0';
        }

        else {
            xil_printf("Invalid character \n\r");
        }

        XIosc_SetCtrl(&IOsys,ctrlVal);
        xil_printf("Value: %X written to ctrl \r\n", ctrlVal);
    }
    return 1;
}
```

Snippet 4.3 - Test program for Exercise 7

4.3 Results

The test application behaved showed that the IP-core behaved as intended.

4.4 Discussion

This exercise was a very instructional in forcing you to figure out all the details involved in the use of custom IP-cores that seem trivial when you follow a lab-guide. Especially, remembering to remove unused ports before generating a bitstream and using the automatically generated header files for getter/setter functions for MM registers in custom IP-cores.