# Exercise 3

## 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.

## 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.

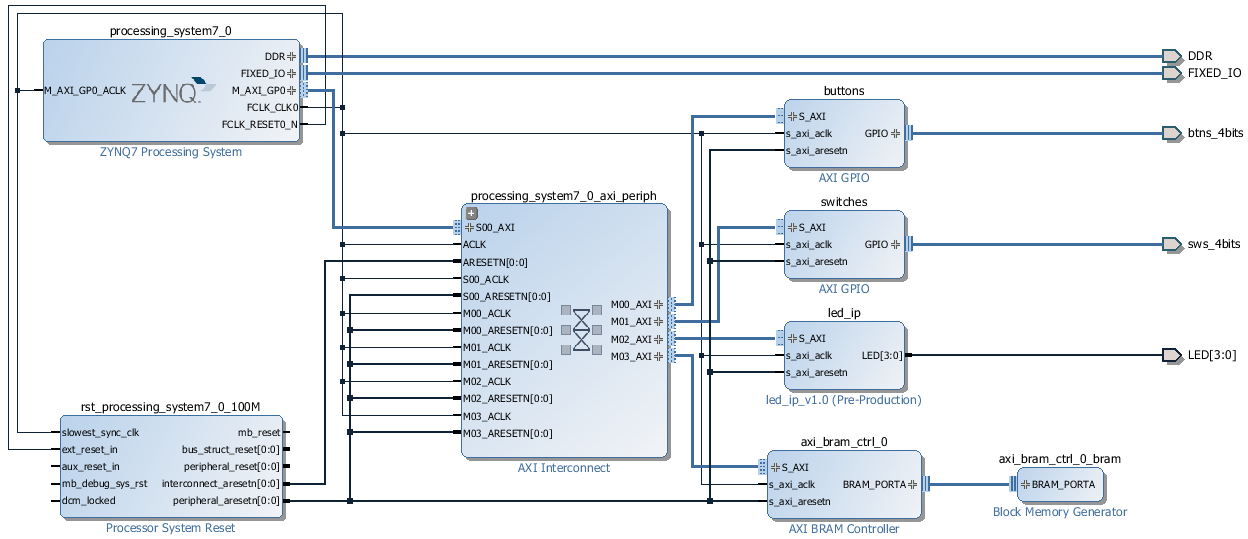


Figure 1.1 - Block Design for Exercise 3

The console input was implemented in a simple fashion using the *inbyte()* function to read characters from the console (See Snippet 1.1) . 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 1.3.

The program then runs a 1 second timer in a loop and updates the LEDs, when the second functionality is enabled (See Snippet 1.4). 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).

Snippet 1.3 - Invalid charater handling

**default** :

xil\_printf("Invalid input: %c\n\r",value);

// 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

**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

## 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’ ↵



Figure 1.2 - Results of running console application

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

## 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.

# Exercise 4

## 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.

## 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.

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 Snippet 2.3 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.

## 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.

# Exercise 5

## 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.

## 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()

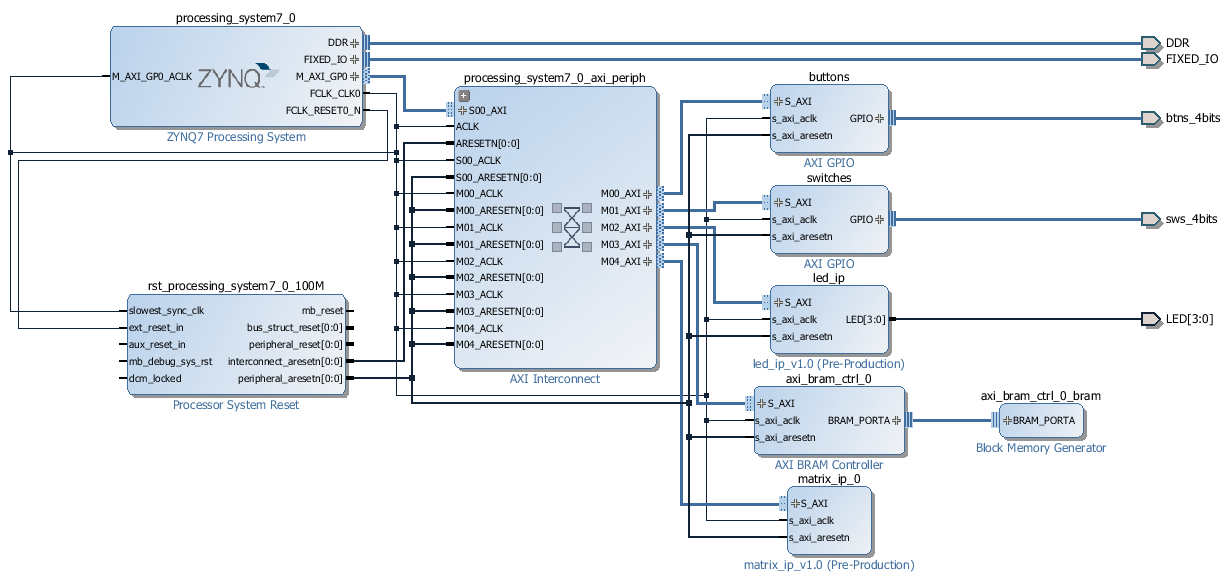
The HW Block Design used in this exercise look as shown in Figure 3.1. Notice the *matrix\_ip\_0* block at the bottom right; this is where HW acceleration is performed.

Figure 3.1 - Exercise 5 HW Block Design