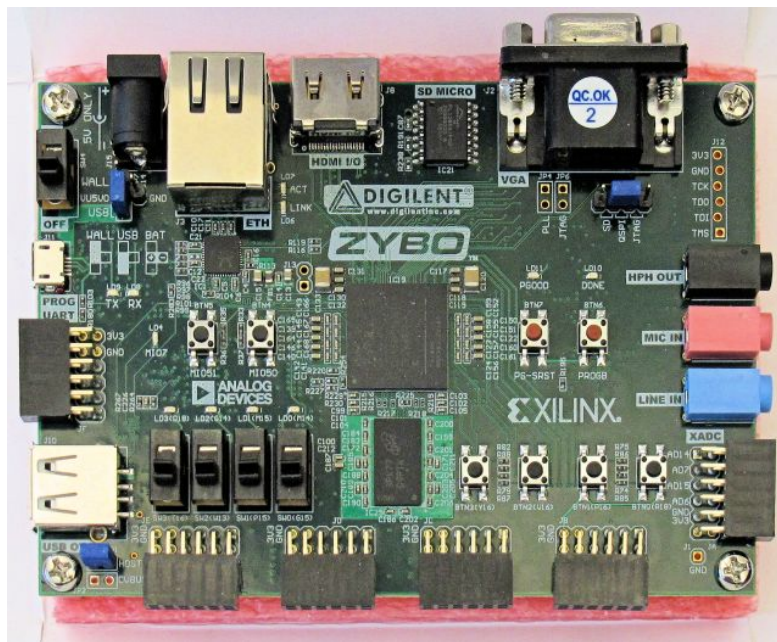


Embedded Real Time Systems



Assignment 2

**Building SoPC designs on a FPGA platform
(Xilinx Zynq - ZYBO)**

Building SoPC designs on a FPGA platform

Description:

In this exercise, you will learn how to develop embedded designs with the ZYBO platform building customized hardware and software.

You must install Xilinx Vivado version 2017.2 in advance by following the guidelines in “**ERTS_Vivado_Installation_2017_2.pdf**”.

After installation, you must follow the below instructions:

Setup hardware

Connect ZYBO

- a. Set the power supply jumper to USB so the board can be powered up and laboratory assignments can be carried out using single micro-USB cable.
- b. Set jumper JP5 to position JTAG.
- c. Connect micro USB cable between PROG UART port of ZYBO and PC.

IT IS VERY IMPORTANT TO INSTALL SUPPORT FOR THE ZYBO PLATFORM SEE INSTRUCTION BELOW: (finstallation of Vivado on Windows or Linux)

Copy the **zybo.zip** file and extract it in the folder:

<Vivado_2017_2_install_dir>\Vivado\2017.2\data\boards\board_files\

This directory is the board files directory and having it in the specified directory will allow you to select Zybo board during the design creation (refer to labs of EmbeddedSystem_labs).

Mandatory exercise for hand-in of assignment: 3, 4, 5 and 7 in journal form. The journal should have focus on discussing the results and obtained learning's.

Goals:

- Learn about Xilinx Design Tools including Vivado, SDK and HLS.
- Learn the design flow for system on a programmable chip.
- Learn about configuring the Processing System (PS).
- Learn about the Programmable Logic (PL) and connecting to the PS.
- Learn about interface driver for USB-UART, GPIO and TIMER.
- Construct a simple command line parser to control hardware.
- Learn how to add Custom IPs to your design.
- Learn how to use High Level Synthesis to simulate C/SystemC and generate HDL code .

1.) Building a Complete Embedded SoPC System

Follow the lab guides in directory **EmbeddedSystem_labs**:

“**lab1_2_EmbeddedSystem.pdf**”, here you will learn to build your first complete embedded SoPC system with the Zync processing system.

“**lab3b_EmbeddedSystem.pdf**”, here you will learn how to add custom IP and BRAM to the system.

2.) Writing Basic Software Application

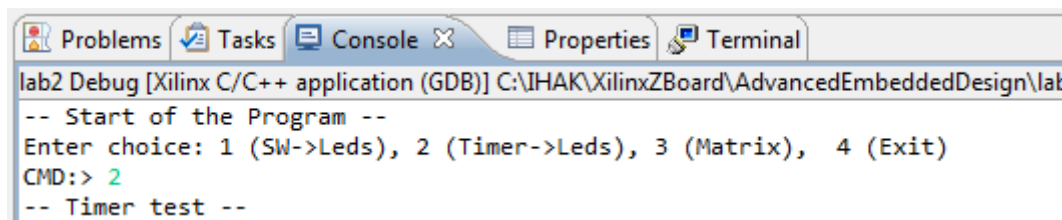
Follow the lab guide in directory **EmbeddedSystem_labs**:

“**lab4_EmbeddedSystem.pdf**”, here you will learn to write a basic software application and use a custom IP.

3.) Create a Console Application

This exercise builds on the hardware and software platform created in **exercise 2**. Change the Linker Script back to use the **ps7_ram_0_S_AIX_BASEADDR** for the Heap and Stack section.

1. 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.



```

lab2 Debug [Xilinx C/C++ application (GDB)] C:\IHAK\XilinxZBoard\AdvancedEmbeddedDesign\lab
-- Start of the Program --
Enter choice: 1 (SW->Leds), 2 (Timer->Leds), 3 (Matrix), 4 (Exit)
CMD:> 2
-- Timer test --
  
```

HINTS:

Input interpreter:

Use the **inbyte()** function to get input character from the USB-UART.

Example:

```

xil_printf("CMD:> ");
/* Read an input value from the console. */
value = inbyte();
skip = inbyte(); //CR
skip = inbyte(); //LF (Skip this line using PuTTY terminal)
switch (value)
{
    case '1':
  
```

To use timer include “XScuTimer.h”:

Select the **system.mss** tab in the BSP. Open the Documentation for **scutimer** on the **Files** link related API functions and data structures are documented for the timer. Example code is listed below for how to initialize the timer and use it.

```

XScuTimer Timer; /* Cortex A9 SCU Private Timer Instance */
#define ONE_SECOND 325000000 // half of the CPU clock speed

// PS Timer related definitions
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);

// Check timer expired
if(XScuTimer_IsExpired(TimerInstancePtr)) {
    // clear status bit
    XScuTimer_ClearInterruptStatus(TimerInstancePtr);
    ....
}

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

4.) Create a Matrix Multiplication

```

// Matrix size
#define MSIZE 4

typedef union {
    unsigned char comp[MSIZE];
    unsigned int vect;
} vectorType;

typedef vectorType VectorArray[MSIZE];

void setInputMatrices(VectorArray A, VectorArray B);
void displayMatrix(VectorArray input);
void multiMatrixSoft(VectorArray A, VectorArray B, VectorArray P);
void multiMatrixHard(VectorArray A, VectorArray B, VectorArray P);
  
```

Use these interfaces for the functions described below

1. Create three global variables of the data structure *vectorArray* called *pInst*, *aInst* and *bTInst*.
2. Implement a function called *setInputMatrices* that fills out the data structures with the input matrix values below:

$$aInst = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}; bTInst = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 \end{bmatrix}^T$$

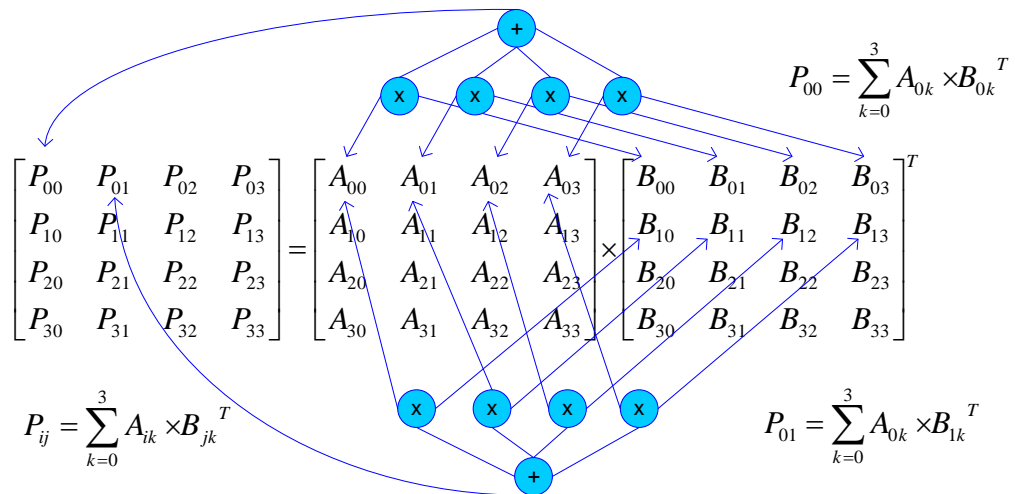
HINT:

$A[0].comp[2] = 3$; sets element on row 0 and column 2 to the value 2 for the matrix A.

3. Implement a function called *displayMatrix* that displays a 4x4 matrix via the USB-UART interface. Test it on matrix *AInst* or *BTinst*.
4. Implement a function called *multiMatrixSoft* that computes the 4x4 matrix product of the expression: $P = A \times B^T$.

HINT:

Use two nested for loops to iterate through all combinations of P_{ij} to calculate the sum of products of each element in P.



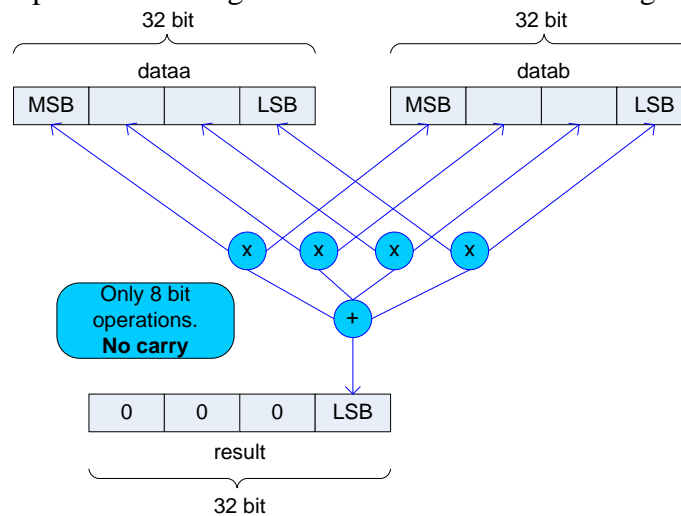
5. Add a command called **mult** that multiplies **AInst** and **BTinst** matrices with the values above. Display the result P and the execution time (in clock ticks) of the function **multiMatrixSoft** in the USB UART window.

5.) Create a Hardware IP Acceleration.

1. Study the implemented VHDL **matrix_ip** component in *matrix_ip_v1_0_S_AXI.vhd* (included in exercise zip file).

All components in VHDL have a number of input and output ports described in the **entity** part of the code. The **architecture** part of the VHDL code describes the implementation the component functionality.

The *matrix_ip* component is adding numbers as illustrated in the figure below.



Remark that all code between **begin** and **end** in the **architecture** part is executed in parallel. VHDL is a strong type language and it is only possible to multiply and add on **unsigned**, **signed** or **integer** types. That's why the `slv_reg0` and `slv_reg1` is casted to **unsigned** and back again to **std_logic_vector** as illustrated in the code snippet below.

```

-- Add user logic here
result1 <= std_logic_vector(unsigned(slv_reg0(31 downto 24)) *
                               unsigned(slv_reg1(31 downto 24)));
result2 <= std_logic_vector(unsigned(slv_reg0(23 downto 16)) *
                               unsigned(slv_reg1(23 downto 16)));
result3 <= std_logic_vector(unsigned(slv_reg0(15 downto 8)) *
                               unsigned(slv_reg1(15 downto 8)));
result4 <= std_logic_vector(unsigned(slv_reg0(7 downto 0)) *
                               unsigned(slv_reg1(7 downto 0)));

i_sum <= std_logic_vector(unsigned(result1(7 downto 0)) +
                           unsigned(result2(7 downto 0)) +
                           unsigned(result3(7 downto 0)) +
                           unsigned(result4(7 downto 0)));

slv_reg2 <= X"000000" & i_sum;
  
```


2. Use the guide “**lab3b_ EmbeddedSystem.pdf**” for how to add a customized IP to the hardware design. Unzip and use the **matrix_ip_1.0.zip** that contains the matrix IP core with user logic to perform multiplication and additions as described above.
3. Create a new function called **multiMatrixHard** that computes the matrix product of the expression: $P = A \times B^T$ but use the new IP core **matrix_ip** to perform part of the matrix multiplication. The C syntax for using the **matrix_ip** is:

```
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);
```

Here parameters A and B are written to the **matrix_ip** in register REG0 and REG1. The result is returned in register REG2.

HINT:

The multiply and add custom instruction can replace the calculations within the inner loop of the matrix product calculation function. Use the $A[i].vect$ notation of the union structure to read the complete row (four bytes) of the matrix instead of $A[i].comp[j]$ that only reads every matrix element (one byte).

4. Display result P and the execution time (in clock ticks) of the function **multiMatrixHard** in the USB UART window and compare it to **multiMatrixSoft**. The ARM processor (PS) runs at 650 Mhz and the hardware matrix multiplication (PL) is clocking at 100 Mhz. The timer is running at half of the PS clock (325 Mhz). Evaluate the computation speed of software vs. hardware implementation based on measured execution time and clock speeds.

6.) HLS exercise

Follow the lab guides in directory **HLS_labs**:

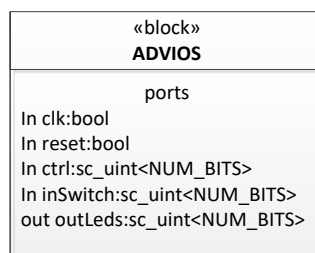
“**lab1_HLS.pdf**”, here you will learn to simulate and synthesis C-code for HDL code generation. Step 7 of the lab exercise is optional. (Matrix multiplication)

“**lab2_HLS.pdf**”, here you will learn to optimize a design using compiler directives. (YUV image filter)

“**lab4_HLS.pdf**”, here you will learn how to create an IP core that will be inserted into a SoPC design. The design is processing audio stereo music and the HLS IP core performs filtering.

7.) HLS exercise with SystemC

In this exercise you will learn how to use SystemC in writing IP cores controlled by software using the AXI4 Lite interface. The figure below shows an advanced input/output (ADVIOS) controller. It should be able to control the 4 leds on the ZYBO board based on the 4 switch input and a memory mapped register (ctrl) that can be controlled from software. The interface to the ADVIOS IP core is shown below as a SysML block.



By writing a bit pattern the ctrl register the function of the ADVIOS IP core should behave as described in the table below. The clk input should be connected to a 100 Mhz clock and reset is active high. (NUM_BITS = 4)

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

It is possible to simulate and synthesis SystemC code with Vivado HLS, see the Vivado Design Suite User guide¹ page 386, 394 – 395 for how to use SC_CTHREAD with clock and reset. Study

¹ Search for UG902 May 30, 2014 Vivado Design Suite User Guide High-Level Synthesis (Copy on BB)

the example of writing a test bench as be found on page 390-392. Read about how to interface the AXI4 bus with SystemC on page 157, 160-161.

1. Write the ADVIOS IP core and a testbench in SystemC.

Hint: Use a CTHREAD to generate a 1 sec. pulse based on the clk input and another CTHREAD to control the port outLeds based on ctrl, inSwitch and the generated 1 sec. pulse signal. Define a **sc_signal** to generate the 1 sec. pulse between the two threads inside the AVIOS module.

2. Document and verify the result of simulation and synthesis using Vivado HLS.
3. Connect the ctrl port to the AXI4Lite interface by using pragma as described in UG902, p. 161 by inserting:

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

Export the RTL core as a VHDL code.

4. Create a Vivado project (Use guide from **lab1_2_EmbeddedSystem.pdf**) and add the ADVIOS IP core and connect it to the LEDS and SWITCHES on the ZYBO board.

In Vivado -> Project Settings -> IP -> Add Repository and browse for the directory: **ADVIO\solution1\impl\ip** and it will be possible to select and add the ADVIOS IP core from **Add IP**.

Add the contrains file **constraints.xdc** that specifies connection to the ZYBO board.

5. Write a program with the Xilinx SDK that verifies the functionality of the IP core and document the results.

