ECE 441/543 Lab 4 Report

Soft Core Processors and IP Blocks

1st Khephren Gould
V01012827
Dept. of Electrical and Computer Engineering
University of Victoria
khephrengould@uvic.ca

2nd Zarina Guzman
V01075558
Dept. of Electrical and Computer Engineering
University of Victoria
zguzman@uvic.ca

***Abstract*—This lab examined the use of the Vivado IP Library to synthesize soft core microcontrollers and microprocessors on an FPGA. Both the MicroBlaze processor and microcontroller were synthesised. The Vivado SDK was used to develop compile and load C programs onto the synthesized Microblaze systems. An ILA was used to probe signals internal to an 8-bit counter synthesized on the FPGA.**

***Index Terms*—Microprocessor, Microcontroller, Vivado, FPGA, ILA, UART, GPIO, C, AXI, MicroBlaze**

I. Introduction

Soft Core Processors have many applications including SoC based CPU-FPGA systems, ASIC prototyping, and embedded applications which require custom computing hardware (intensive DSP computations etc.). The Xilinx MicroBlaze Soft Core processor is available within the Vivado IP library and was used for the experiments in this lab. The MicroBlaze is a Harvard architecture CPU that implements the RISC ISA. In this lab, both the stand-alone MicroBlaze Processor as well as the complete Microblaze Microprocessor were synthesized on an Artix-7 FPGA. Additional features of the Vivado IP library were used including peripheral IP blocks for UART and GPIO. Furthermore, an industry standard debugging tool called an ILA (integrated logic analyzer) was synthesized and used to analyze a simple 8-bit counter circuit.

II. In Lab Steps

A. Stand alone MicroBlaze Microcontroller Instantiation - hello world

In this step, a complete microcontroller was synthesized and programmed on the lab's Artix 7 FPGA. The MicroBlaze microcontroller contains a RISC soft core processor that interfaces with many built in peripherals (UART, GPIO interfaces, Block RAM memory etc.).

First, the built-in IP Library from Vivado was used to configure and instantiate an entity for the microcontroller. Eight GPI and GPO pins were each enabled, the UART interface was enabled at 9600 baud and a clock frequency of 100MHz was selected.

After instantiating the microcontroller IP block, the Vivado SDK was used to write a small C Program. The C code was compiled and included in the bitstream used for programming the FPGA. The C-code used for this step is shown in Figure 1. As shown in the figure, the code prints the text "Hello" to the screen 5 times using a loop.

Next, the bitstream was generated and used to program the FPGA. The results from connecting to the microcontroller's UART interface were viewed using the PuTTY serial console. This is shown in Figure 2.

Finally, a C program for interacting with the GPIO interface of the microcontroller was compiled and the FPGA was programmed. The program simply reads the state of the GPI inputs (connected to the switches) and writes this state to the GPO outputs (connected to the LEDs) in a loop. The results are shown in Figure 3.

B. MicroBlaze system using IP blocks - AXI UART helloworld

In this step, a MicroBlaze Processor was instantiated and programmed onto the FPGA. As opposed to a Microcontroller, a Processor does not contain any built in communication peripherals.

This step used the Vivado IP Block Design tool. This tool allows the designer to configure their system graphically. It also contains automations for routing the interconnections between the IP blocks.

First, a block for the system clock was added to the design. Next, a Microblaze processor block and a UART interface block were added. In this case, the UART interface communicates with the processor using the AXI protocol.

As in step 1, the Vivado SDK was used to write a simple C program. For this step, the program consists of a loop that prints the loop index at each iteration. The code along with the results from programming the FPGA and monitoring the serial console are shown in Figures 5 and 6 respectively.

C. Extending the MicroBlaze system to include AXI GPIO

In step 3, a GPIO peripheral was added to the processor. As with the UART peripheral in step 2, the GPIO interface communicates with the processor using the AXI protocol. The

```c
/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"


int main()
{
    init_platform();
    for(int i = 0; i < 5; ++i){
        print("Hello");
    }

    cleanup_platform();
    return 0;
}
```

Fig. 1: ”Hello World” C Program

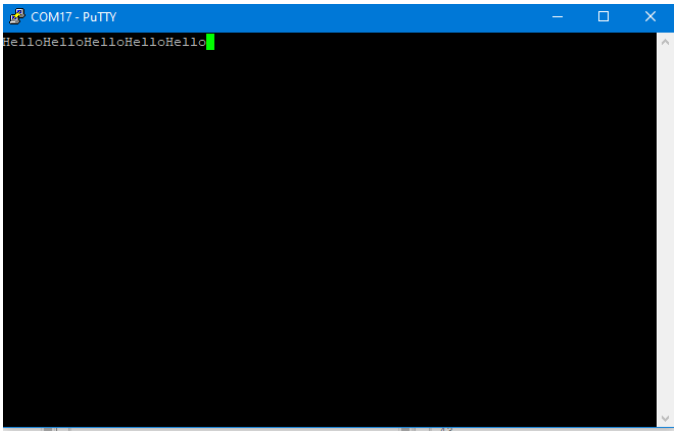

Fig. 2: ”Hello World” UART Reception

GPIO peripheral was added to the design using the Vivado IP Block Design Tool. Because the project was created using the Nexys 7 Development board template, the Block Design Tool could be used to connect the GPIO peripheral to the LEDs and slide switches directly. The completed schematic for the system is shown in Figure **??**.

After adding the GPIO peripheral to the IP block design, the Vivado SDK was used to create a C Program to interact with it. The program monitors the state of the slide switches and writes a string over the UART interface. The string contains the hexadecimal representation of the current value contained on the switches. Additionally, the program writes the state of the switches to the LEDs on the Nexys development board. The serial console output as a result of running the C program

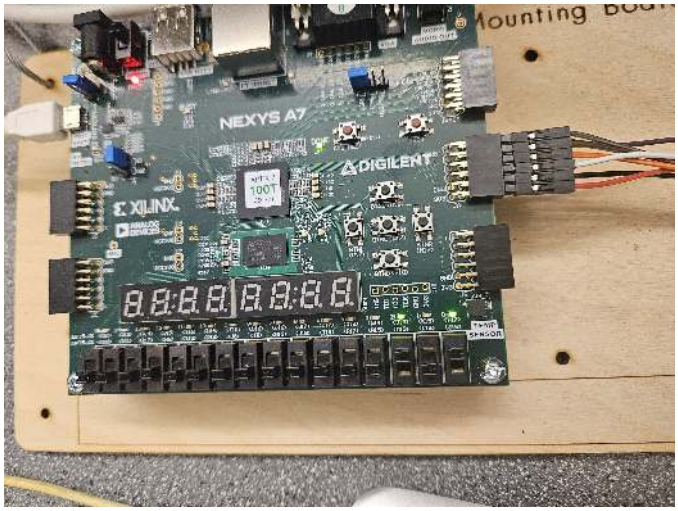

Fig. 3: Microcontroller GPIO

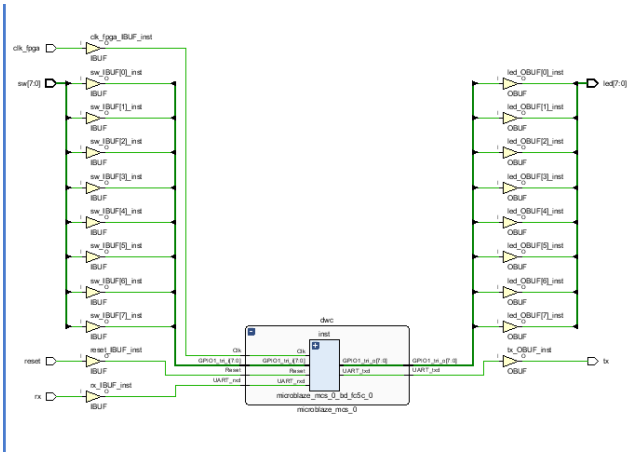

Fig. 4: Step 1 System Schematic

```c
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"


int main()
{
    init_platform();
    for(int i = 0; i < 5; i++){
        xil_printf("i: %d\n\r",i);
    }

    cleanup_platform();
    return 0;
}
```

Fig. 5: Step 2 C Program

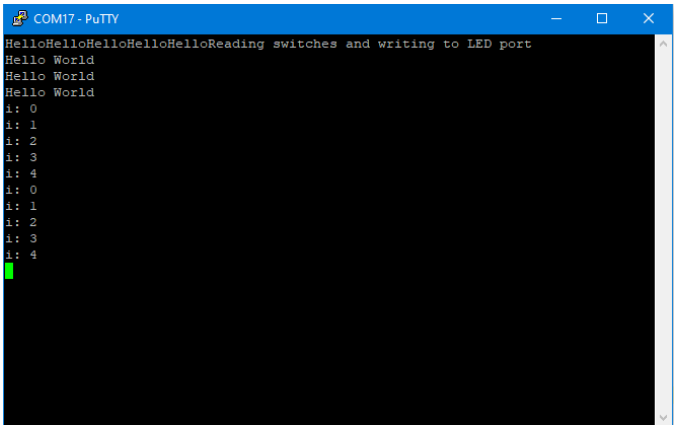

Fig. 6: Step 2 Serial Console

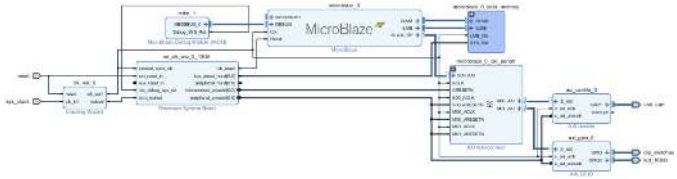

Fig. 7: Step 3 IP Block Design

on the processor is shown in Figure 8.

D. Create and add a custom IP block to the AXI system

In Step 4, a custom 16×16 signed multiplier IP block was created using Vivado's "Create and Package New IP" wizard, which generated an AXI4-Lite peripheral scaffold with four 32-bit registers. The provided multiplier VHDL code was added as a design source, and the peripheral's VHDL was manually edited to instantiate the multiplier, connecting the upper and lower 16 bits of register 0 as inputs and routing the 32-bit product to the register 1 read path. After re-packaging the IP, it appeared in the Vivado IP catalog and was added to the existing MicroBlaze block design from Step 3, where

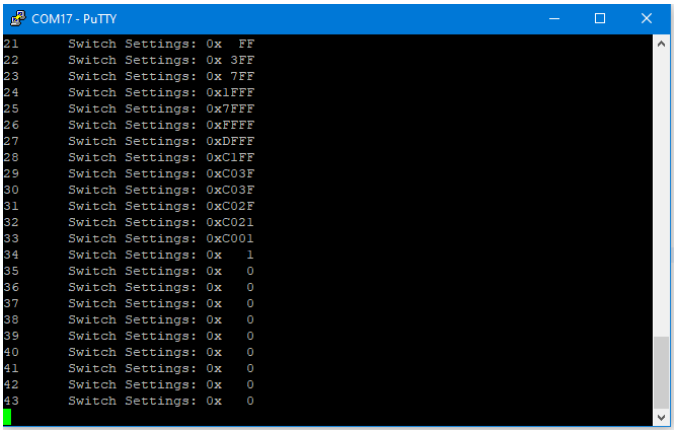

Fig. 8: Step 3 Serial Console

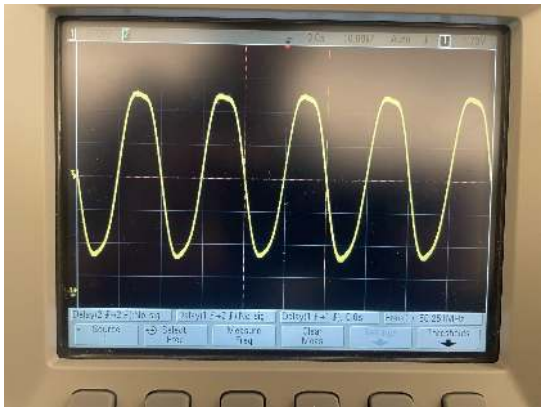

Fig. 9: Speed of Counter LSB - 50MHz

Connection Automation handled the AXI bus wiring. A new bitstream was generated and programmed onto the FPGA, and a short C program in the SDK was written to test the multiplier by writing two packed 16-bit operands as a single 32-bit write and reading back the resulting product. Several test cases were run to verify correct signed multiplication behavior.

E. Binary Counter and ILA demo using IP blocks

In Step 5, a new Vivado project was created to demonstrate the use of the Integrated Logic Analyzer (ILA) IP block, a powerful on-chip debugging tool that allows internal FPGA signals to be captured and examined in real time. A block design was built using a 32-bit binary counter, the System ILA core, two Slice IP blocks, and a Clocking Wizard, with the counter's enable input tied to slide switch SW0 on the NEXYS A7 board. The Slice blocks were configured to extract specific bit ranges from the counter output, routing bits 26-23 to the onboard LEDs and bits 7-0 to the JB PMOD connector. After generating and programming the bitstream, the ILA waveform viewer in Vivado's Hardware Manager was used to capture and display all 32 bits of the running counter, with a sample depth of 1024. Toggling SW0 allowed the counter to be started and stalled, and the LSB frequency of the counter was measured on the PMOD output. The output of the LSB of the counter is shown in Figure 9. The frequency corresponds to $\frac{100MHz}{2^1} = 50MHz$.

The toggling frequency of bit 26 can be calculated in a similar way:

$f = \frac{100MHz}{2^{26}} = 1.49Hz$

The result from displaying the ILA signals using Vivado are shown in Figure 10

III. Questions

A. What is a microcontroller? How is it different from a general purpose microprocessor?

To explain what a microcontroller is, the microprocessor must be explained first. This is because microcontrollers contain a microprocessor. A microprocessor is a digital circuit that executes instructions formatted according to the ISA (instruction set architecture) the processor was designed for.

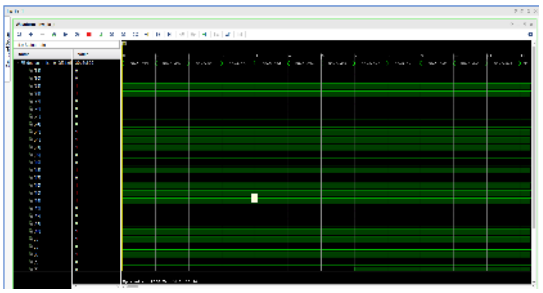

Fig. 10: 32 bit Counter ILA Results

Typically a microprocessor is contained within a single silicon chip. A microcontroller is a larger single chip system that, in addition to the microprocessor, contains peripheral circuits that can be used to control circuits external to the chip. For example, the STM32 line of microcontroller is used in many ECE courses at UVic. This microcontroller contains many peripherals such as timers, UART, SPI, and ADC. The STM32 is built off the ARM Cortex M line of microprocessors. In this case, the microcontroller and microprocessor are designed by completely separate companies and integration is possible because both companies adhere to standardized protocols (such as AMBA).

B. What is a UART

UART is a serial communication protocol that stands for Universal Asynchronous Receiver-Transmitter. It is used for communication between devices, such as sensors and microcontrollers. The UART IC contains a transmitter and receiver register as well as two shift registers for each direction of data flow (TX and RX), and finally a clock generator is provided that is then divided down to a chosen baud rate for the system.

The UART data frame consists of a start bit, followed by the data bits, parity bit (optional), followed by the stop bit.

C. What is an ILA?

The acronym ILA stands for Integrated Logic Analyzer. An ILA allows the designer to directly probe individual wires or registers of interest within their FPGA. Vivado provides an IP block that can be used to synthesize an ILA onto the FPGA. The ILA functions like a complete oscilloscope and logic analyzer. It can be used to view waveforms from any gate or wire of interest. It also allows configuration of the trigger condition just like a real oscilloscope. It contains capture memory which is typically implemented using the Block RAM on the FPGA. Every clock cycle, samples from all the probes are stored in the block RAM and can be viewed from within the Vivado ILA viewer.

IV. Conclusion

This lab demonstrated the design and implementation of soft-core processor systems using the Vivado IP library on an FPGA platform. Both a standalone MicroBlaze microcontroller and a customizable MicroBlaze processor system were successfully synthesized and programmed, highlighting the flexibility of soft-core architectures compared to fixed hardware solutions. The integration of peripheral IP blocks such as UART and GPIO illustrated how complex embedded systems can be rapidly developed using standardized interfaces like AXI.

Additionally, the lab provided experience in extending processor functionality through the creation and integration of a custom AXI IP block, reinforcing the concept of hardware/software co-design. The use of the Integrated Logic Analyzer (ILA) enabled real-time observation of internal FPGA signals, demonstrating an effective method for debugging and validating digital designs.

Overall, this lab emphasized the advantages of FPGA-based system design, including modularity, scalability, and the ability to tailor hardware to specific application requirements. The combination of software development and hardware configuration provides a powerful approach for implementing embedded systems.

V. Appendix A: Programming Assignment Code Listings

Modify the constraints file so that the microcontroller outputs are directed to the JA PMOD on the NEXYS board

The constraints file modifications are shown in Figure 11.

Write a C program to output an 8 bit binary counter on the GPIO outputs. Measure the speed of the counter.

The 8-bit binary counter was implemented in C by using a for loop that iterated an integer from 0 to 255, with each value written to the GPIO port. The constraints file was modified to route the output signals to the JA port, as shown in Figure 11, allowing measurement of the least significant bit (LSB) using an oscilloscope (Figure 12). The measured frequency of the LSB was approximately 820 kHz, which was significantly lower than the 50 MHz observed for the 8-bit hardware counter. This difference is due to the MicroBlaze executing instructions sequentially and incurring overhead from looping and GPIO writes over the bus, whereas the hardware counter increments every clock cycle using dedicated parallel logic.

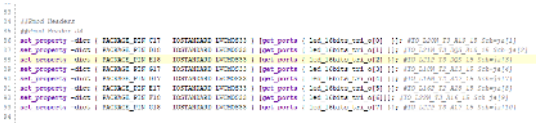

Fig. 11: Constraints File Modification

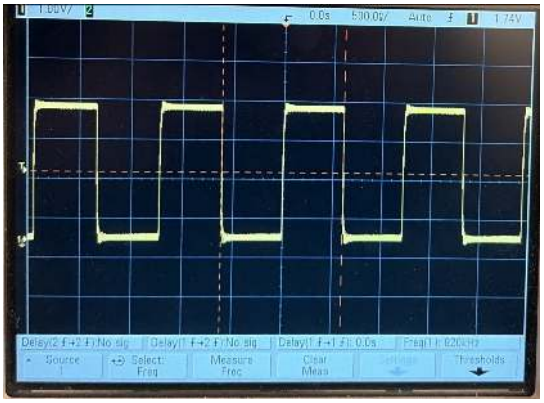

Fig. 12: 8-bit counter in C: LSB Frequency

Listing 1: 8-bit Counter C program

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpio.h"
#include "microblaze_sleep.h"


/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are defined here such that a user can easily
 * change all the needed parameters in one place.
 */
#define GPIO_OUTPUT_DEVICE_ID XPAR_GPIO_0_DEVICE_ID
#define GPIO_INPUT_DEVICE_ID XPAR_GPIO_1_DEVICE_ID
#define GPIO_BITWIDTH 16 /* This is the width of the GPIO */

/* The following constant is used to determine which channel of the GPIO is
 * used if there are 2 channels supported in the GPIO. Must be either 1 or 2.
 */

#define INPUT_SWITCH_CHANNEL 1 //in block design defined as GPIO
#define OUTPUT_LED_CHANNEL 2 //in block design defined as GPIO_2

/************************** Variable Definitions ************************/

/*
 * The following are declared globally so they are zeroed and so they are
 * easily accessible from a debugger
 */
XGpio GpioOutput; /* The driver instance for GPIO Device configured as O/P */
XGpio GpioInput; /* The driver instance for GPIO Device configured as I/P */

/*****************************************************************************/


int main()
{
 u8 i;

    u32 DataRead;
    int Status;


    init_platform();
    xil_printf("Initialize\n\r");

    /*
  * Initialize the GPIO driver so that it's ready to use,
  * specify the device ID that is generated in xparameters.h
  * also set its data direction
  */
  Status = XGpio_Initialize(&GpioOutput, GPIO_OUTPUT_DEVICE_ID); //output
  if (Status != XST_SUCCESS) {
   return XST_FAILURE;
  }
  /* Set the direction for all signals to be outputs */
  XGpio_SetDataDirection(&GpioOutput, OUTPUT_LED_CHANNEL, 0x0);


  Status = XGpio_Initialize(&GpioInput, GPIO_INPUT_DEVICE_ID); //input
  if (Status != XST_SUCCESS) {
   return XST_FAILURE;
  }
  /* Set the direction for all signals to be inputs */
  XGpio_SetDataDirection(&GpioInput, INPUT_SWITCH_CHANNEL, 0xFFFFFFFF);

 for(i = 0; i < 256; i++)
 {
 XGpio_DiscreteWrite(&GpioOutput, OUTPUT_LED_CHANNEL, i);
 }

cleanup_platform();
return 0;
}
```