Laboratory 2 – Arithmetic Operations and Program Flow Control

I. Arithmetic Operations

Objectives

- To execute software programs to perform arithmetic operations: addition, subtraction, multiplication, division, and modulation
- To learn how to write methods and make method calls
- To learn the concept of modularity and code reusability

Activity Summary

1. Launch Xilinx Vitis after opening and exporting the hardware platform
2. Create ArithOperations application and write arithmetic operation codes
3. Program the FPGA (if not already programmed), setup Run Configuration, and run ArithOperations application
4. Repeat activity 3 for changes made to the application source file

Activities

1. **Open and Export Hardware Design**

   Follow the procedure described in Lab 1 instructions to unzip and open **Lab 2 hardware Files** and export the hardware design files **Top.xsa (including bitstream)**.

As we mentioned in Lab 1 instructions to export the hardware platform and create the XSA file:

1. Click **File → Export → Export Hardware** (Figure 1a)
2. The Export Hardware Platform wizard opens (Figure 1b). Click **Next** to start the process.
3. In the Output window (Figure 1c), select **Include bitstream** to ensure the complete hardware implementation is included, then click **Next**.
4. In the Files window (Figure 1d), specify the XSA filename (default: "top") and export location. The system shows "The XSA will be written to: top.xsa". Click **Next**.
5. Review the summary (Figure 1e) showing that a hardware platform named 'top' will be created as top.xsa with post-implementation model and bitstream. Click **Finish** to complete the export.

The XSA file is now ready for use in Vitis IDE.

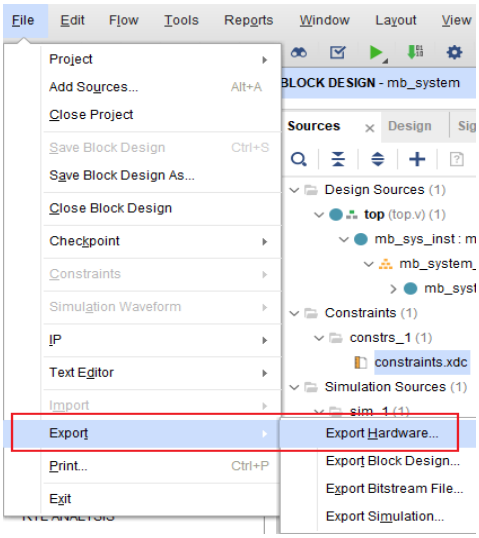

Figure 1(a)

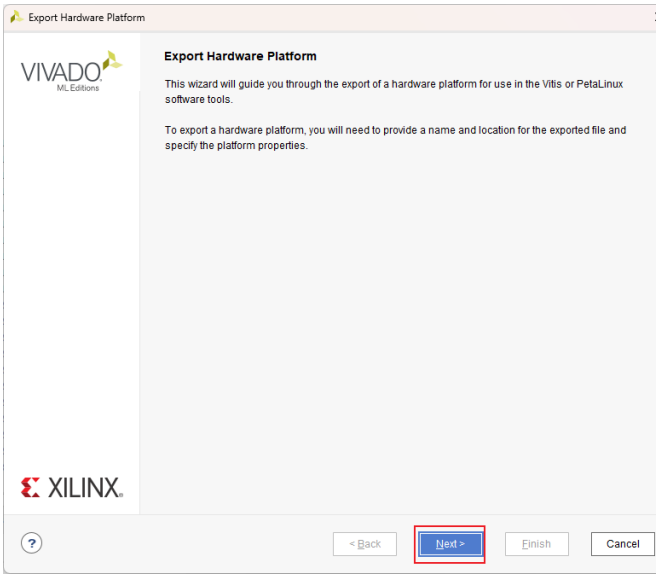

Figure 1(b)

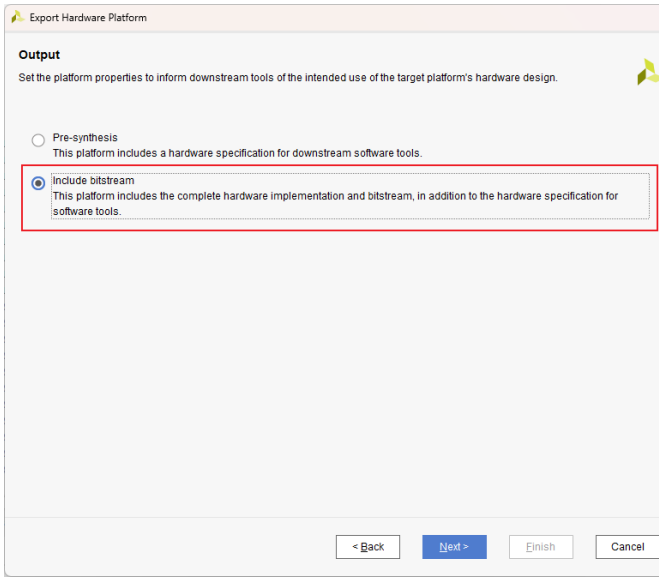

Figure 1(c)

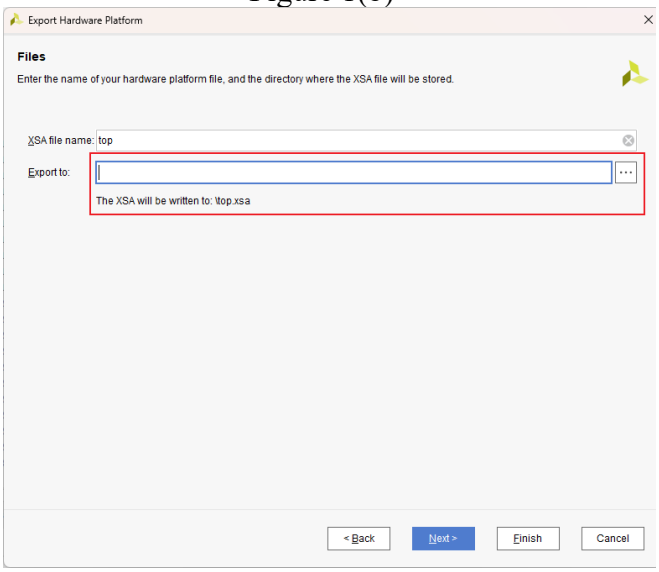

Figure 1(d)

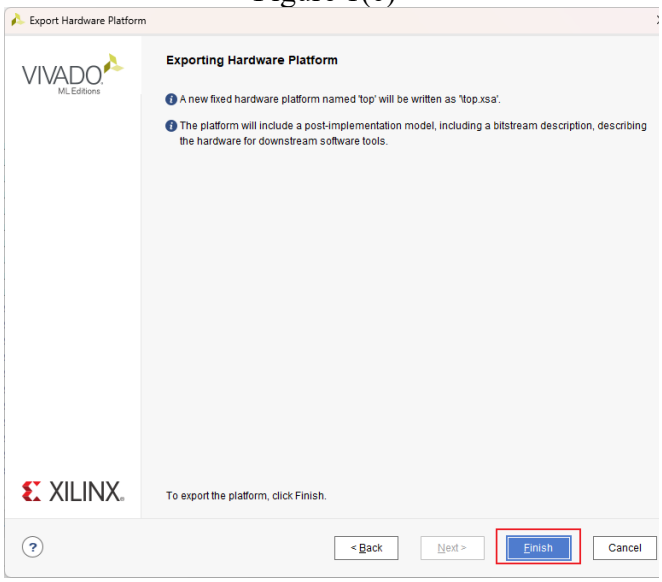

Figure 1(e)

2. **Launch Vitis**

Create the new folder in the PC, named Lab2, then open Vitis and select the file path of Lab2 as the workspace, and select **Launch**.

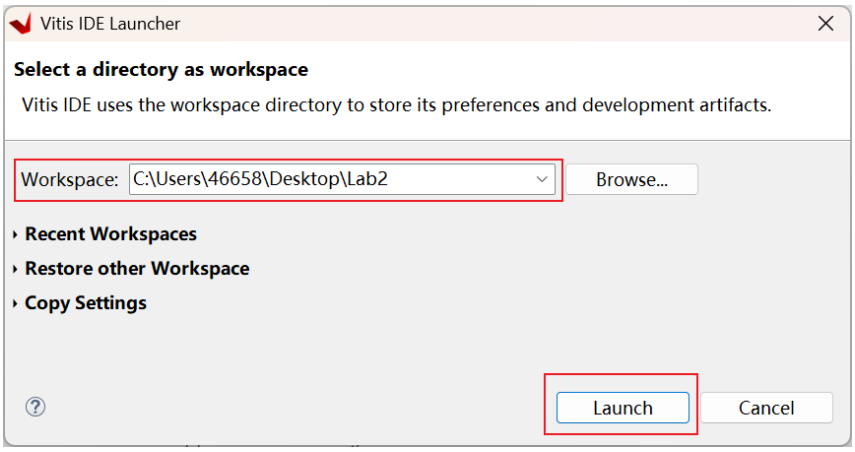

Figure 2

3. **Create a New Application Project**

Based on your images, here's the revised description for creating an application project:

Create a New Application Project

Launch Vitis IDE and follow these steps to create an application project:

Step 1: Launch Vitis IDE The Vitis IDE main interface will open (Figure 3a), displaying the welcome page. In the PROJECT section, click **Create Application Project** to create an application project.

Step 2: Select Platform In the New Application Project wizard (Figure 3b), select **Create a new platform from hardware (XSA)**, then click the **Browse...** button to locate the XSA file exported earlier from Vivado (the top.xsa file as shown in Figure 3c).

Step 3: Configure Platform On the platform selection page (Figure 3d), browse and select the top.xsa file from your project directory. The XSA file contains the hardware specification exported from Vivado.

Step 4: Configure Application Project On the Application Project Details page (Figure 3e), enter the application project name (such as "ArithOperations"). The system will automatically create an associated system project "ArithOperations_system" with target processor shown as "microblaze_0".

Step 5: Select Domain On the domain selection page (Figure 3f), the system displays details for the "standalone_microblaze_0" domain, including processor microblaze_0 and 32-bit architecture.

Step 6: Choose Template On the template selection page (Figure 3g), select the **Hello World** template from the embedded software development templates. This template creates a "Hello World" program written in C.

Step 7: Complete Creation Click **Finish** to complete the application project creation. Vitis IDE will create the application project and open the complete development environment, displaying the project structure in the Explorer.

Once created, you can begin developing and debugging your Lab2 embedded application.

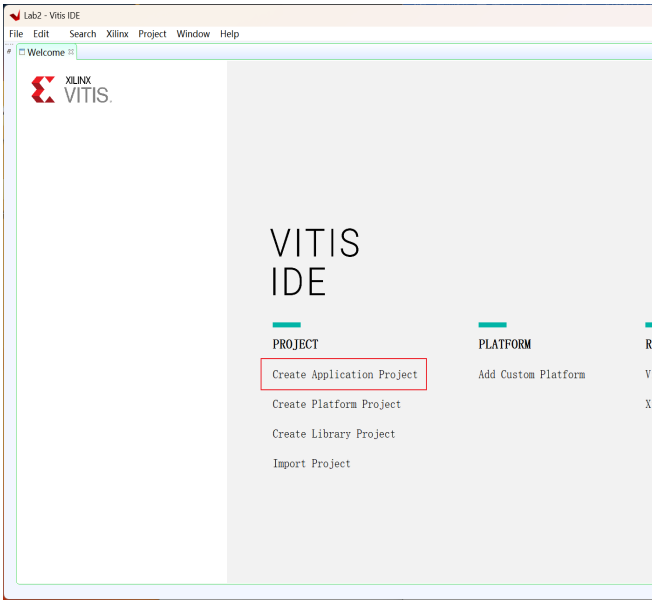

Figure 3(a)

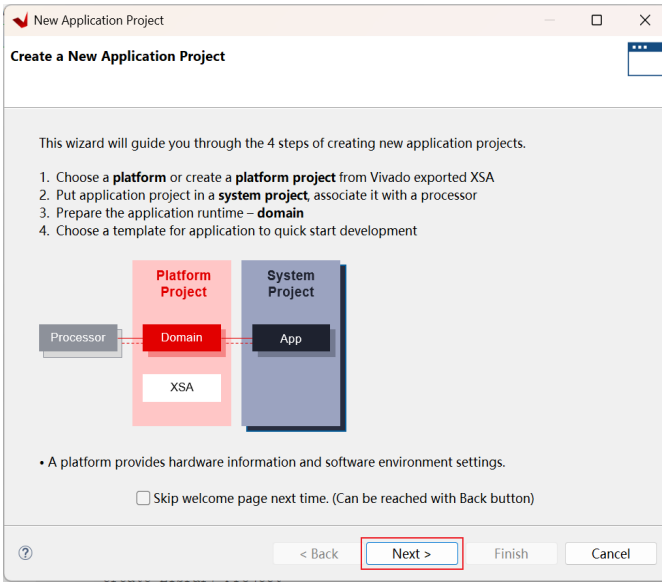

Figure 3(b)

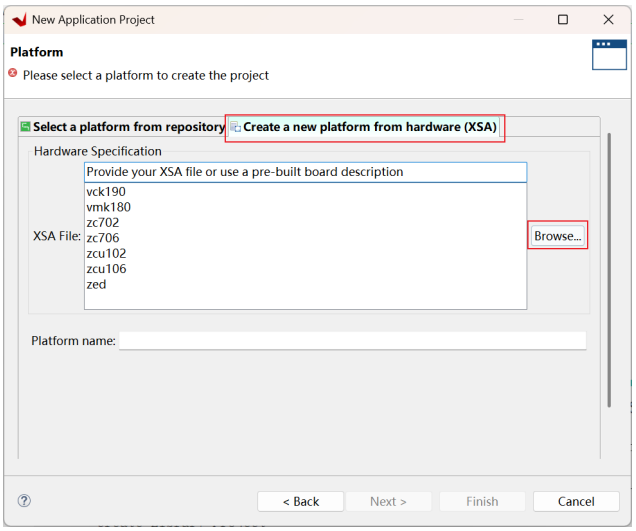

Figure 3(c)

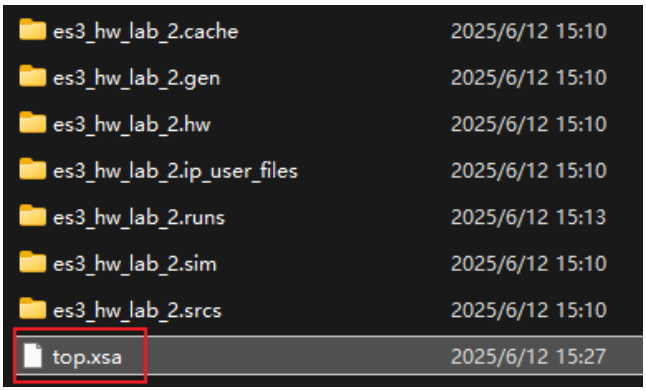

Figure 3(d)

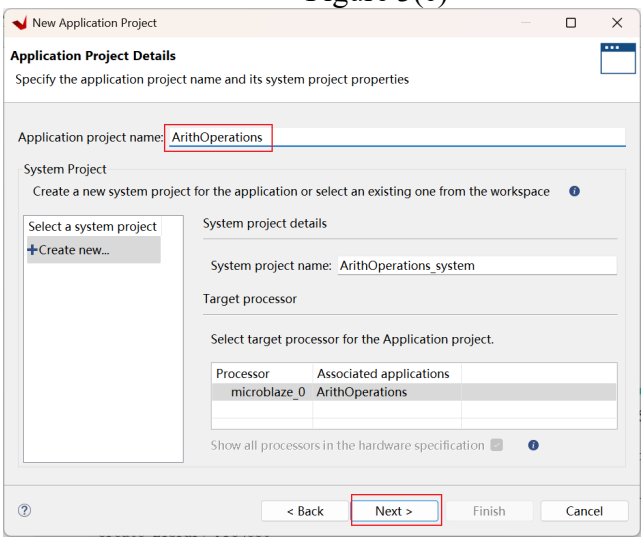

Figure 3(e)

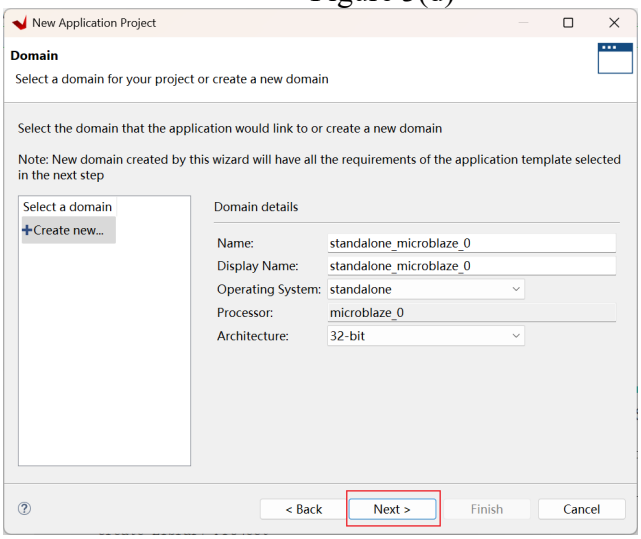

Figure 3(f)

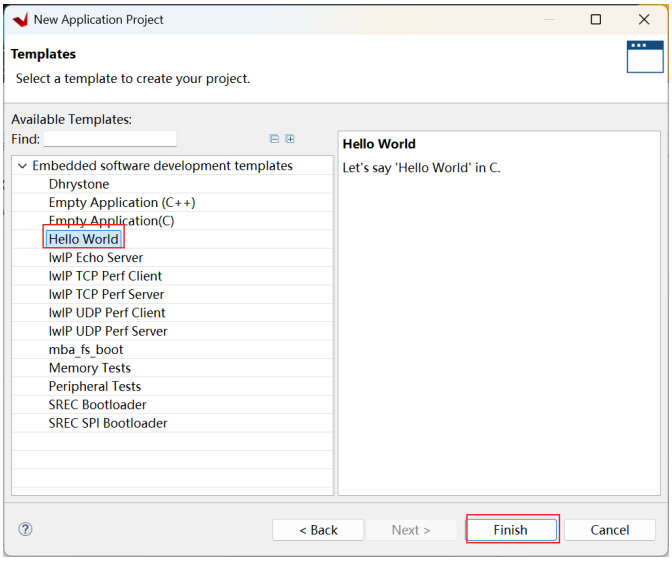

Figure 3(g)

4. Rename the Source File

(1) As shown in Figure 4, in the **Project Explorer window**, expand **ArithOperations → src**

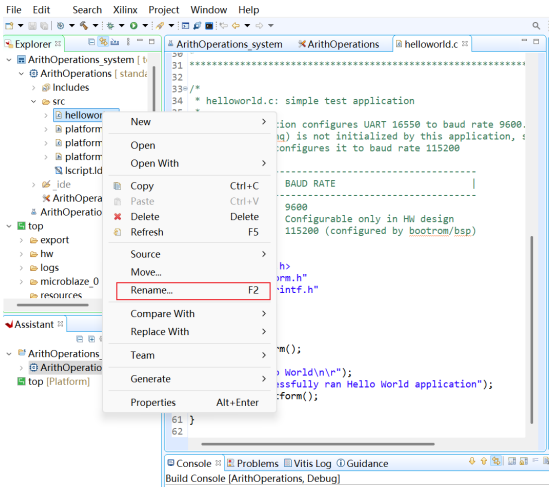

Figure 4

(2) Right-click on **helloworld.c**, select **Rename**, and change the filename to **arith_operations.c**

5. **Modify the Source Code**

(1) Open **arith_operations.c** in the editor.

(2) Remove the following two lines of code:

print("Hello World\n\r");

print("Successfully ran Hello World application");

6. **Coding Guidelines**

When implementing your code, ensure that all code within the main() function is placed between init_platform(); and cleanup_platform();. Additionally, define any additional functions at the global program scope (same level as main()), not inside the main() function, as shown in Figure 5.

```
int main()
{
    init_platform();

    // Setup the interrupt system
    setUpInterruptSystem();

    while(1)
    {
        valuesReceived = FALSE;
        /*
         * << Write the code to receive value1,
         *    value2 and the operation type here >>
         */
        valuesReceived = TRUE;
        // Wait for interrupt to be serviced
        while (interruptServiced == FALSE);
    }

    cleanup_platform();
    return 0;
}

s32 adder(s32 augend, s32 addend)
{
    //Write the adder function codes here
}

s32 subtractor (s32 minuend, s32 subtrahend)
{
    //Write the subtractor function codes here
}
```

Figure 5

In today's session we will first create functions to perform five arithmetic operations: addition, subtraction, multiplication, division, and modulation. The operations will be performed on integer variables. We will create these operations as methods that will be called in the main body and then print the result to the Console. Now, create the arithmetic operation methods shown in Figures 4(a) and 4(b),making sure you add appropriate comments where necessary. These methods can be written **after including the libraries** or **after the main() function**. You are required to write the codes formultiplication, division, and modulation yourselves as they are similar to those for addition and subtraction. The arithmetic operators for the program are shown as Table 1.

```
s32 adder(s32 augend, s32 addend)
{
    s32 sum;
    sum = augend + addend;
    return sum;
}

s32 subtractor(s32 minuend, s32 subtrahend)
{
    s32 difference;
    difference = minuend - subtrahend;
    return difference;
}
```

Figure 4(a)

```
s32 multiplicator(s32 multiplicand, s32 multiplier)
{
    // Write the code for multiplication here

}

s32 divider(s32 dividend, s32 divisor)
{
    // Write the code for division here

}

u32 modulator(u32 dividend, u32 divisor)
{
    // Write the code for modulation here

}
```

Figure 4(b)

C Basic Arithmetic Operators

Operation	Operator	Example	Description
Addition	+	int c = a + b;	Adds a and b
Subtraction	–	int c = a – b;	Subtracts b from a
Multiplication	*	int c = a * b;	Multiplies a by b
Division	/	int c = a / b;	Divides a by b
Modulus	%	int c = a % b;	Computes a mod

Table 1. Arithmetic operators.

Then, include the header file that contains the basic types for Xilinx software IP. This file includes definitions for the various integer types we are using. For instance, u32 and s32 mean unsigned and signed 32-bit integers, respectively. To include this file, add #include "xil_types.h" before main() as shown in **Part 1 (Header file) in Figure 5**. You also need to make explicit declarations of the functions you have just added to the source file (see **Part 1 (Header file) in Figure 5**). While some compilers allow the use of functions without explicit declarations, it is good practice to make these declarations. The print() function we used in the previous laboratory allows the printing of only strings. In order to print integer variables, we need a print function that allows the integer-formatting of the value to be printed. We have such a function in Vitis called xil_printf(). An explicit declaration of this function also needs to be added before main() (see **Part 1 (Header file) in Figure 5**).

Part 1(Header file)

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

s32 adder (s32 num1, s32 num2);
s32 subtractor (s32 minuend, s32 subtrahend);
s32 multiplicator (s32 multiplicand, s32 multiplier);
s32 divider (s32 dividend, s32 divisor);
u32 modulator (u32 dividend, u32 divisor);
void xil_printf (const char *ctrl1,...);
```

Part 2(Main function)

```c
int main()
{
    init_platform();

   s32 add_result;
   s32 sub_result;
   s32 mul_result;
   s32 div_result;
   u32 mod_result;

   add_result = adder (20, 23);
   sub_result = subtractor (1, 19);
   mul_result = multiplicator (-2, 8);
   div_result = divider(20, 10);
   mod_result = modulator (5, 2);

   xil_printf("addition result: %d \n\r", add_result);
   xil_printf("subtraction result: %d \n\r", sub_result);
   xil_printf("multiplicaiton result: %d \n\r", mul_result);
   xil_printf("division result: %d \n\r", div_result);
   xil_printf("modulo result: %d \n\r", mod_result);

    cleanup_platform();
    return 0;
}
```

Part 3(Function)

```c
s32 adder (s32 num1, s32 num2)
{
    s32 sum;
    sum = augend + addend;
    return sum;
}
s32 subtractor (s32 minuend, s32 subtrahend)
{
    s32 diff;
    diff = num1 - num2;
    return diff;
}
```

Please add functions for Multiplication, Division, and Modulus in the Function section.

Figure 5

The next thing to do is to declare integer variables to hold operation results, to call the operation subroutines, and to display the results (see **Part 2 (Main function) in Figure 5**). Once this step is completed, configure the FPGA and run the application by following the steps described in Lab 1. **Part 3 in Figure 5** should include the function definitions themselves. These functions, which you've declared earlier, contain the actual code that performs the arithmetic operations. This section is where you would write the logic for each function, such as adding, subtracting, or multiplying two numbers. The function definitions must match the declarations in **Part 1 (Header file) in Figure 5** in terms of their return type and parameters.

Once you have completed writing the **arith_operations.c** code, Vitis IDE will display the complete development environment. The project structure shows both the system project and application project. First, you need to program the FPGA with the hardware design by right-clicking on the ArithOperations project and selecting **Program Device** from the context menu (Figure 6a). This step configures the FPGA with the hardware bitstream containing the MicroBlaze processor and other peripherals. Next, to build the application, right-click on the ArithOperations project and select **Build Project** from the context menu (Figure 6b). This compiles the C source code into an executable file (.elf) that can run on the MicroBlaze processor.

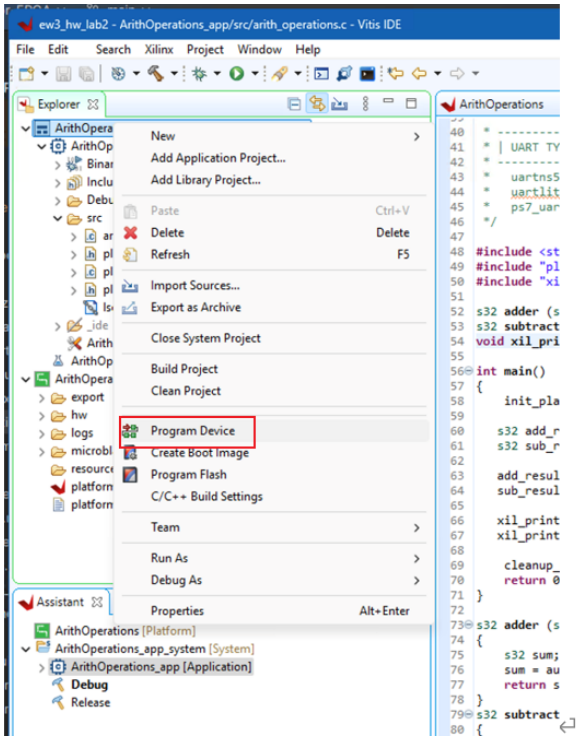

Figure 6(a)

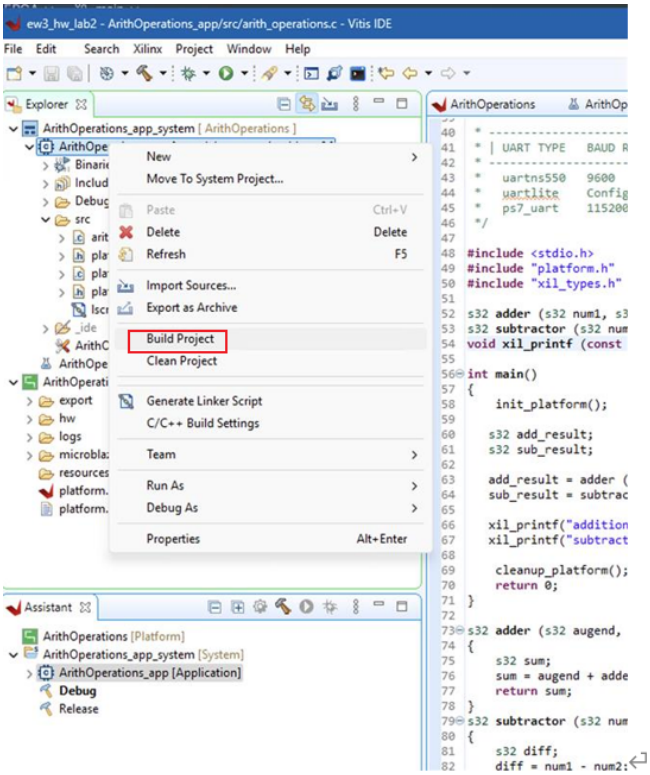

Figure 6(b)

Figure 8 shows what the Console should look like when the application is successfully run. Following the same procedure as described in **Section 4 Step 5: Open Terminal View** and **Step 6: Configure Serial Terminal** of Lab 1 instructions, open the terminal to view the output. Additionally, follow **Step 7: Configure Run Settings** and **Step 8: Set Up Debug Configuration** as outlined in Section 4 to properly configure and run the application. However, I have still included the corresponding operation screenshots as shown in Figure 7(a)-(d). You can play around with the arguments in the operation functions. If you have successfully carried out the activity on arithmetic operations, you can now move on to program flow control.

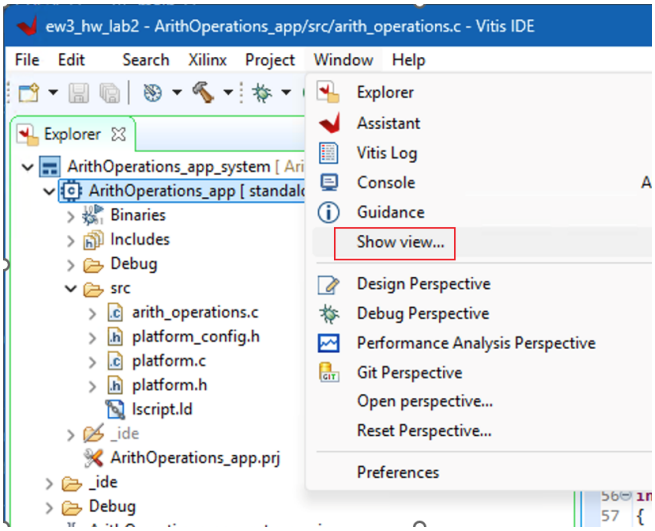

Figure 7(a)

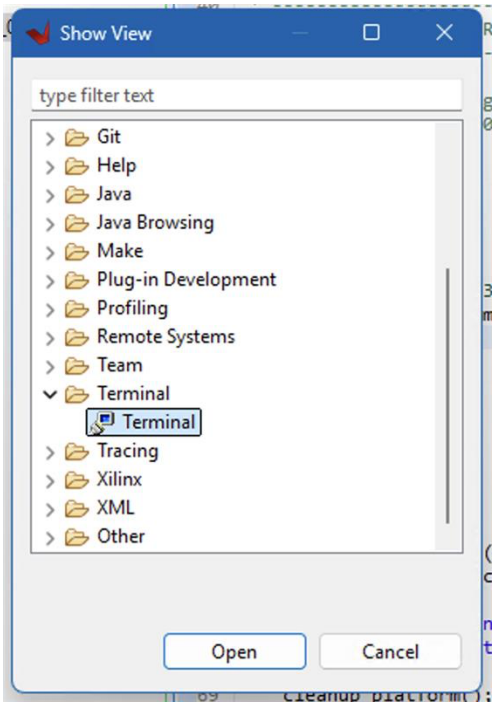

Figure 7(b)

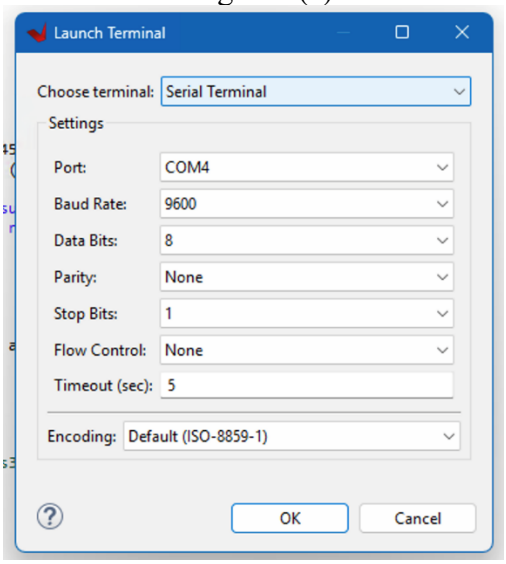

Figure 7(c)

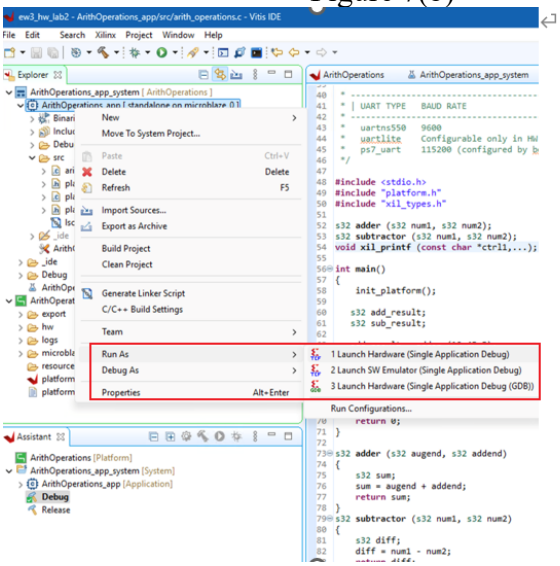

Figure 7(d)

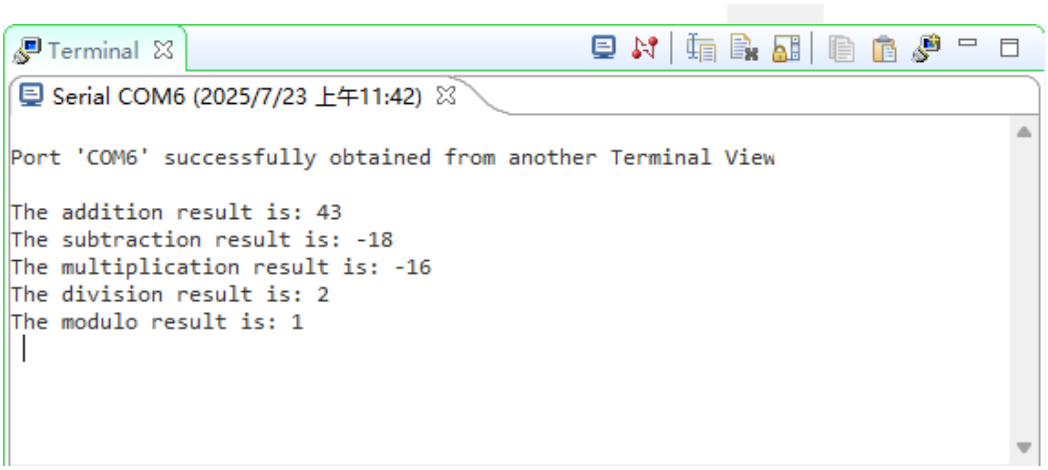

Figure 8

II. Program Flow Control

Objectives

- Build a controlled arithmetic unit on MicroBlaze and run this on the FPGA
- To learn the concept of modularity and code reusability

Activity Summary

1. Create ProgFlowCntrl application for program flow operations
2. Use the arithmetic operations in a program flow algorithm
3. Program the FPGA (if not already programmed), setup Run Configuration, and run the application
4. Repeat activity 3 for changes made to the application source file

Activities

Now we move to program flow control activities. A software processor like MicroBlaze executes instructions in sequential order, that is, one after the other. Unlike hardware, instructions are never executed in parallel. As a result, in the Hello World application that we worked with in Lab 1 and the arithmetic operations we just executed, all instructions were executed in sequential order, from top to bottom in any subroutine. For instance, in the main() method of Figure 5, the init_platform() function is executed first, then the s32 add_result declaration which reserves memory space for the addition result, until the last instruction return 0.

However, if there is a need to control the execution order of instructions, the C programming language offers a number of syntaxes and constructs that can be used to achieve this. These include if-else, while loop, for loop, do-while loop, switch (case), break, and continue statements, as well as the ternary operator. We will use some of these to design an application that reuses our arithmetic methods, deciding which one to use based on the type of operation selected from the PC keyboard input. Table 2 shows the codes assigned to the arithmetic operations.

Operation Type	Code
Addition	0
Subtraction	1
Multiplication	2
Division	3
Modulation	4

Table 2: Arithmetic operation type codes

The activity here involves receiving the operation code through the Universal Asynchronous Receiver/Transmitter (UART) from the system keyboard, and depending on the code received, the corresponding operation is performed by calling the appropriate arithmetic operation. Create a new application with the name **ProgFlowCntrl** and rename **helloworld.c** to **prog_flow_cntrl.c**.The function uartReceive()that allows you to receive the operation code from UART has already been written for you. However, to use it, you need to add the file that contains it to your project.

The file is named **xuart_receiver.c** and can be found in the "Lab 2 - Codes" folder. A simple way to add the file to your project is to copy it from the Lab 2 – Codes folder, right-click on the **src** folder and select **Paste**.
This file contains a Xilinx UART Lite driver that implements the uartReceive() function for single character reception using polled mode. UART (Universal Asynchronous Receiver-Transmitter) is a serial communication protocol that enables data transmission between devices without a shared clock signal, using TX and RX lines for bidirectional communication. **Note that the uartReceive() function is intended to only receive the ASCII code of a single character. So you should only use numbers between 0 and 9 in your operations** because the function is designed with single-byte reception (TEST_BUFFER_SIZE = 1) and numerical digits '0'-'9' (ASCII 48-57) provide a controlled input range for simplified processing.
References:
https://xilinx.github.io/Embedded-Design-Tutorials/docs/2021.1/build/html/docs/Introduction/Zynq7000-EDT/2-using-zynq.html

Now, go back to the **prog_flow_cntrl.c** file in the C/C++ Editor and type the codes in Figure 7 in the main() function. Remember to copy the arithmetic operation methods from the **arith_operations.c** file to the new application. Specifically, you need to copy the addition, subtraction, multiplication, division, and modulo functions that you previously created. You can either create a new source file with these methods and add it to the project, or simply copy the five arithmetic functions (add, subtract, multiply, divide, and modulo) and place them in the function definition section below the main() function in your new code.

The while(1) loop is an infinite loop that allows the continuous execution of the block of codes enclosed in the curly brackets. After receiving the operation type, the "if statement" is then used to determine which of the operations to perform. Note that only one of the operations can be performed;if the operation code is not between 0 and 4, an error message is printed with the value of the code received.

Make sure to include all necessary function declarations at the top of the file if you choose to place the functions below main(). The arithmetic functions should be placed in the global scope, not inside the main() function, following the same structure as shown in the previous figure 5.

After copying all the arithmetic operation functions, save the file and run the application. You can then interact with it by clicking in the Console window and typing one of the operation codes (0 for addition, 1 for subtraction, 2 for multiplication, 3 for division, 4 for modulo) to test the program flow control functionality.

```
char arith_op_type;


while(1)
{
    xil_printf("Enter an operation type and press enter:\n\r");
    arith_op_type = uartReceive();

    if (arith_op_type == '0') {
        add_result = adder(20, 23);
        xil_printf("The addition result is: %d \n\r", add_result);
    }
    else if(arith_op_type == '1') {
        sub_result = subtractor(1, 19);
        xil_printf("The subtraction result is: %d \n\r", sub_result);
    }
    else if(arith_op_type == '2') {
        mul_result = multiplicator(-2, 8);
        xil_printf("The multiplication result is: %d \n\r", mul_result);
    }
    else if(arith_op_type == '3') {
        div_result = divider(20, 10);
        xil_printf("The division result is: %d \n\r", div_result);
    }
    else if(arith_op_type == '4') {
        mod_result = modulator(5, 2);
        xil_printf("The modulo result is: %d \n\r", mod_result);
    }
    else {
        xil_printf("Error! The operation type (%c) is wrong!\n\r", arith_op_type);
    }

}
```

Figure 7

The same functionality in Figure 7 can be achieved by using the switch statement. The structure of the code required to do this is given in Figure 8. You are required to complete this code. Also the difference between while loop and switch loop can be referred as:
https://support.ptc.com/help/arbortext/r8.2.1.0/en/index.html#page/Program/acl_ref/help5670.html

You can achieve the same functionality using a switch statement as shown in Figure 8. Complete this code following the suggested steps:

- Read the first number from UART and convert from ASCII to integer
- Read the second number and convert it
- Read the operation type
- Use the switch statement to select the operation

```c
switch (arith_op_type)
{
    case '0' :
        // Perform the addition operation here
        break;

    case '1' :
        // Perform the subtraction operation here
        break;

    case '2' :
        // Perform the multiplication operation here
        break;

    case '3' :
        // Perform the division operation here
        break;

    case '4' :
        // Perform the modulation operation here
        break;

    default  :
        // You can write codes to handle errors here
        break;
}
```

Figure 8

Note: Use atoi() function (include stdlib.h) or subtract 48 from the ASCII value to convert characters to numbers. The variables value1 and value2 are already declared in Figure 8 for receiving the numbers.
ASCII to Number Conversion Explanation: When receiving numeric characters through UART, they are received as ASCII values. For example, the character '0' has ASCII value 48, '1' has 49, etc. To convert these to actual numeric values, you can either:

1. Use atoi() function after including <stdlib.h>
2. Subtract 48 from the ASCII value (e.g., 53('5') - 48 = 5)

References:
ASCII Table and Character Codes: https://www.asciitable.com/

Tips: The **atoi** function, an abbreviation of ASCII to Integer, is a standard library function defined in the C header <stdlib.h>. Its primary purpose is to convert a string representation of a numerical value, expressed as a sequence of ASCII characters, into its corresponding integer value in base 10. For example:

```c
const char *numStr = "12345"; // Given string value
int num = atoi(numStr); // Convert string to integer
printf("Converted value: %d\n", num); // Print the result
```

III. Interrupt Handling

Objectives

- Learn and understand the role and necessity of interrupts in FPGA-based systems (BASYS 3 + MicroBlaze).
- Study the advantages of interrupts in terms of responsiveness, resource utilization, and system scalability.
- Understand the fundamental design principles of safe and reliable Interrupt Service Routines (Ref: https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm).

Necessity and Importance

Improved Responsiveness

- Peripherals generate interrupt requests (IRQs) allowing immediate processor response
- Minimizes latency, crucial for real-time applications

Efficient Resource Utilization

- Processor can perform other tasks or enter low-power states when idle
- Reduces wasted cycles and power consumption

Event Reliability and Scalability

- Peripherals actively signal the processor, preventing missed events
- Enables simultaneous management of multiple peripherals via interrupt controller

Hardware-Software Synchronization

- Bridges the speed gap between fast processors (100MHz) and slower human inputs
- Supports modular design with clear priorities for real-time systems

Interrupt Processing Path

1. Peripheral Event generates an IRQ (e.g., timer overflow, UART input, button press).
2. Interrupt Controller (e.g., Xilinx INTC) receives the signal, determines vector and priority.
3. MicroBlaze Processor acknowledges, saves current program state (registers, program counter), and jumps to ISR.
4. ISR Execution: perform essential operations only (read data, clear interrupt flags, set event indicators for main program).
5. Return to Main Program: processor restores context and continues; complex processing is handled in main program using flags or queues set by ISR.

Principles of Designing Reliable ISRs

- Keep It Short and Simple: restrict ISR to essential, deterministic operations only
- Avoid Blocking Operations: do not use sleep, wait, blocking I/O, or non-reentrant functions (e.g., printf)
- Proper Variable Handling: declare shared variables as volatile; protect multi-byte updates with critical sections
- Clear Interrupt Sources: explicitly acknowledge and clear interrupt flags to enable future triggers
- Memory Management: avoid large local arrays, dynamic allocations, and excessive stack usage
- Use Deferred Processing: shift complex operations to main program using flags, queues, or semaphores

Example (BASYS 3 / MicroBlaze)

Note: This is a complete real-hardware Interrupt example that helps understand the workflow and logic of Interrupts. However, this code is for illustrative purposes only; please do not attempt to run it. Specific exercises will be conducted in Lab 3 - IO Operations - LED Application.

Refer to the following: MiniZed Interrupt Tutorial
https://community.element14.com/challenges-projects/design-challenges/pathprogrammable3/b/blog/posts/how-to-enable-interrupts-with-minized

```
/*************************** Include Files *********************************/
#include "xparameters.h"
#include "xstatus.h"
#include "xintc.h"
#include "xil_exception.h"
#include "xtmrctr.h"
#include <stdio.h>

/************************** Constant Definitions ***************************/

/*
 * Hardware device IDs - these come from xparameters.h
 * generated by Vivado hardware design
 */
#define INTC_DEVICE_ID          XPAR_INTC_0_DEVICE_ID
#define TIMER_DEVICE_ID         XPAR_TMRCTR_0_DEVICE_ID
#define TIMER_INTERRUPT_ID      XPAR_INTC_0_TMRCTR_0_VEC_ID

/* Timer configuration */
#define TIMER_COUNTER_0         0
#define TIMER_LOAD_VALUE        0xF0000000  // ~1 second delay

/**************************** Global Variables *****************************/

/* Hardware instances */
static XIntc InterruptController;       // Interrupt controller instance
static XTmrCtr TimerCounter;            // Timer counter instance

/* Application data - shared between main and ISR */
volatile char value1, value2;           // Input numbers (as characters)
volatile char operation;                // Operation: '0'=add, '1'=subtract
volatile int result;                    // Calculation result

/* Synchronization flags - volatile because accessed by both main and ISR */
volatile int data_ready = 0;            // Main sets: data ready for processing
volatile int processing_done = 0;       // ISR sets: calculation completed
volatile int interrupt_count = 0;       // Debug: count interrupts

/************************** Function Prototypes ****************************/

   /* Main application functions */
   int main(void);
   char get_user_input(void);

   /* Interrupt system setup (based on Xilinx official example) */
   int setup_interrupt_system(void);

   /* Interrupt service routine */
   void timer_interrupt_handler(void *CallbackRef, u8 TmrCtrNumber);
```

```c
/* Math functions */
int add_numbers(int a, int b);
int subtract_numbers(int a, int b);

/*****************************************************************************/
/**
* Main Function - Application Entry Point
*
* @return   XST_SUCCESS if successful, otherwise XST_FAILURE
*****************************************************************************/
int main(void)
{
    int Status;

    printf("\n=== MicroBlaze Real Interrupt Calculator ===\n");
    printf("Using AXI Interrupt Controller + AXI Timer\n");
    printf("Operations: 0=add, 1=subtract\n");
    printf("Numbers: 0-9 only\n\n");

    /*
     * STEP 1: Initialize interrupt system
     * This sets up the real hardware interrupt controller
     */
    Status = setup_interrupt_system();
    if (Status != XST_SUCCESS) {
        printf("ERROR: Failed to setup interrupt system\n");
        return XST_FAILURE;
    }

    /*
     * STEP 2: Main application loop
     * This demonstrates interrupt-driven processing
     */
    while (1) {

        /* Get user input */
        printf("First number (0-9): ");
        value1 = get_user_input();

        printf("Second number (0-9): ");
        value2 = get_user_input();

        printf("Operation (0=add, 1=subtract): ");
        operation = get_user_input();

        /*
         * Signal that data is ready and start timer
         * This will trigger a REAL hardware interrupt after delay
```

```
        data_ready = 1;
        processing_done = 0;

        printf("Starting timer... waiting for REAL interrupt!\n");

        /* Start hardware timer - this will generate interrupt */
        XTmrCtr_Start(&TimerCounter, TIMER_COUNTER_0);

        /*
         * Wait for interrupt to complete processing
         * The main program is "blocked" here, but the interrupt
         * can still occur and "interrupt" this waiting loop!
         */
        while (processing_done == 0) {
            /* Main program waits here */
            /* Timer interrupt will automatically call timer_interrupt_handler() 
        }

        /* Display result calculated by interrupt service routine */
        printf("Result calculated by ISR: %d\n", result);
        printf("Total interrupts so far: %d\n\n", interrupt_count);

        /* Stop and reset timer for next calculation */
        XTmrCtr_Stop(&TimerCounter, TIMER_COUNTER_0);
        XTmrCtr_Reset(&TimerCounter, TIMER_COUNTER_0);
    }

    return XST_SUCCESS;
}

/*****************************************************************************/
/**
* Setup Interrupt System - Based on Xilinx Official Example
*
* This function initializes the real hardware interrupt system:
* 1. Initialize AXI Interrupt Controller
* 2. Initialize AXI Timer
* 3. Connect interrupt handlers
* 4. Enable interrupts at all levels
*
* @return   XST_SUCCESS if successful, otherwise XST_FAILURE
*****************************************************************************/
int setup_interrupt_system(void)
{
    int Status;

    printf("Setting up REAL hardware interrupt system...\n");

    /*
```

```c
/*
 * PART 1: Initialize Interrupt Controller Hardware
 */
Status = XIntc_Initialize(&InterruptController, INTC_DEVICE_ID);
if (Status != XST_SUCCESS) {
    printf("ERROR: Interrupt controller initialization failed\n");
    return XST_FAILURE;
}

/* Test interrupt controller hardware */
Status = XIntc_SelfTest(&InterruptController);
if (Status != XST_SUCCESS) {
    printf("ERROR: Interrupt controller self-test failed\n");
    return XST_FAILURE;
}

/*
 * PART 2: Initialize Timer Hardware
 */
Status = XTmrCtr_Initialize(&TimerCounter, TIMER_DEVICE_ID);
if (Status != XST_SUCCESS) {
    printf("ERROR: Timer initialization failed\n");
    return XST_FAILURE;
}

/* Configure timer options */
XTmrCtr_SetOptions(&TimerCounter, TIMER_COUNTER_0,
                   XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);

/* Set timer load value (determines interrupt frequency) */
XTmrCtr_SetResetValue(&TimerCounter, TIMER_COUNTER_0, TIMER_LOAD_VALUE);

/*
 * PART 3: Connect Timer Interrupt to Interrupt Controller
 * This creates the connection: Timer -> Interrupt Controller -> CPU
 */
Status = XIntc_Connect(&InterruptController, TIMER_INTERRUPT_ID,
                       (XInterruptHandler)XTmrCtr_InterruptHandler,
                       &TimerCounter);
if (Status != XST_SUCCESS) {
    printf("ERROR: Failed to connect timer interrupt\n");
    return XST_FAILURE;
}

/*
 * PART 4: Register our custom interrupt handler with timer
 * When timer interrupt occurs:
 * 1. XTmrCtr_InterruptHandler() is called first (Xilinx driver)
 * 2. Then timer_interrupt_handler() is called (our custom code)
 */
XTmrCtr_SetHandler(&TimerCounter, timer_interrupt_handler, &TimerCounter);
```

```
    /*
     * PART 5: Start Interrupt Controller in REAL mode
     * XIN_REAL_MODE = use real hardware interrupts (not simulation)
     */
    Status = XIntc_Start(&InterruptController, XIN_REAL_MODE);
    if (Status != XST_SUCCESS) {
        printf("ERROR: Failed to start interrupt controller\n");
        return XST_FAILURE;
    }

    /* Enable timer interrupt in the interrupt controller */
    XIntc_Enable(&InterruptController, TIMER_INTERRUPT_ID);

    /*
     * PART 6: Enable Interrupts at CPU Level
     * This connects the interrupt controller to the MicroBlaze CPU
     */

    /* Initialize CPU exception table */
    Xil_ExceptionInit();

    /* Register interrupt controller handler with CPU */
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                 (Xil_ExceptionHandler)XIntc_InterruptHandler,
                                 &InterruptController);

    /* Enable CPU to accept interrupts */
    Xil_ExceptionEnable();

    printf("SUCCESS: Hardware interrupt system ready!\n");
    printf("- AXI Interrupt Controller: Initialized\n");
    printf("- AXI Timer: Initialized\n");
    printf("- CPU Exceptions: Enabled\n\n");

    return XST_SUCCESS;
}

/*****************************************************************************/
/**
* Timer Interrupt Service Routine (ISR)
*
* THIS IS A REAL INTERRUPT HANDLER!
* - Called automatically when AXI Timer generates interrupt
* - Interrupts whatever the main program was doing
* - Performs the calculation
* - Signals completion to main program
```

```c
/**
* Timer Interrupt Service Routine (ISR)
*
* THIS IS A REAL INTERRUPT HANDLER!
* - Called automatically when AXI Timer generates interrupt
* - Interrupts whatever the main program was doing
* - Performs the calculation
* - Signals completion to main program
*
* @param CallbackRef - Reference passed during handler registration
* @param TmrCtrNumber - Timer counter number that generated interrupt
******************************************************************************/
void timer_interrupt_handler(void *CallbackRef, u8 TmrCtrNumber)
{
    interrupt_count++;  // Count interrupts for debugging

    printf("*** REAL INTERRUPT OCCURRED! (Count: %d) ***\n", interrupt_count);

    /* Check if there's data ready to process */
    if (data_ready == 1) {

        printf("ISR: Processing calculation...\n");

        /* Convert ASCII digits to integers */
        int num1 = value1 - '0';  // '5' becomes 5
        int num2 = value2 - '0';  // '3' becomes 3

        /* Perform calculation based on operation */
        if (operation == '0') {
            result = add_numbers(num1, num2);
            printf("ISR: Performed addition: %d + %d = %d\n", num1, num2, result);
        }
        else if (operation == '1') {
            result = subtract_numbers(num1, num2);
            printf("ISR: Performed subtraction: %d - %d = %d\n", num1, num2, result);
        }
        else {
            result = -999;  // Error code for invalid operation
            printf("ISR: Invalid operation, result = %d\n", result);
        }

        /* Clear data ready flag and signal completion */
        data_ready = 0;
        processing_done = 1;

        printf("ISR: Calculation complete, signaling main program\n");
    }
    else {
        printf("ISR: No data ready, ignoring interrupt\n");
    }

    printf("*** ISR FINISHED - RETURNING TO MAIN PROGRAM ***\n");
}
```

```c
/*****************************************************************************/
/**
* Get User Input
*
* Simple function to get a single character from user
*
* @return   Character entered by user
******************************************************************************/
char get_user_input(void)
{
    char input;
    scanf(" %c", &input);  // Space before %c ignores whitespace
    return input;
}

/*****************************************************************************/
/**
* Math Functions - Simple arithmetic operations
******************************************************************************/
int add_numbers(int a, int b)
{
    return a + b;
}

int subtract_numbers(int a, int b)
{
    return a - b;
}
```

Summary

- On BASYS 3 and similar FPGA platforms, interrupts are essential for efficient, low-latency event handling.
- Adherence to the principle "minimal ISR, deferred processing" ensures both performance and system stability.
- Proper design enhances responsiveness, scalability, and determinism, while misuse can compromise system stability.

References:

You can go to the lab2 reference video in the learn section to watch the video, or directly copy the link to your browser to view the video:

1. https://www.youtube.com/watch?v=17H8tnK0m8g (INTERRUPTs of the MicroBlaze System)
2. https://www.youtube.com/watch?v=dTp0c41XnrQ&ab_channel=AmanBytes (Ref1 Intro to C)
3. https://www.youtube.com/watch?v=KJgsSFOSQv0&ab_channel=freeCodeCamp.org (Ref2 Intro to C)
4. https://support.ptc.com/help/arbortext/r8.2.1.0/en/index.html#page/Program/acl_ref/help5670.html (Intro to if, while, for, switch)
5. https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm
6. https://www.asciitable.com/ (ASCII Table and Character Codes)
7. https://community.element14.com/challenges-projects/design-challenges/pathprogrammable3/b/blog/posts/how-to-enable-interrupts-with-minized (MiniZed Interrupt Tutorial)