

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 SDK 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

Follow the steps taken in the previous laboratory to open and export the hardware design for Laboratory 2 and then launch the SDK. Create a new Application Project and name it “ArithOperations”. To allow the SDK tool to automatically include some necessary header files, always start creating all your applications with a Hello World template (see Lab 1 instructions on how to do this). Once you do this, the ArithOperations folder appears in Project Explorer window. Expand the **src** folder and right-click on the **helloworld.c** file. Select **Rename...** and rename the file to **arith_operations.c**. Then open the file and delete the “`void print(char *str);`” and “`print(“Hello World\n\r”);`” lines. Note that any code you write in the `main()` function must reside between the `init_platform()` and `cleanup_platform()` functions and that methods are created in the same global program scope as the `main()` and not inside it.

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 1(a) and 1(b), making sure you add appropriate comments where necessary. You are required to write the codes for multiplication, division, and modulation as they are quite similar to those for addition and subtraction.

```
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 1(a)

```

s32 multiplier(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 1(b)

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 Figure 2. You also need to make explicit declarations of the methods you have just added to the source file (see Figure 2). 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 the SDK as `xil_printf()`. See the note on “A Brief Introduction to C Programming” for more information on formatted printing. An explicit declaration of this function also needs to be added before `main()` (see Figure 2).

```

#include <stdio.h>
#include "platform.h"
#include "xil_types.h" // Added for integer type definitions

// Explicit declarations of used functions
s32 adder(s32 num1, s32 num2);
s32 subtractor(s32 minuend, s32 subtrahend);
s32 multiplier(s32 multiplicand, s32 multiplier);
s32 divider(s32 dividend, s32 divisor);
u32 modulator(u32 dividend, u32 divisor);
void xil_printf(const char *ctrl1, ...);

int main()
{
    init_platform();
}

```

Figure 2

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 Figure 4). Once this step is completed, configure the FPGA and run the application by following the steps described in Laboratory 1. Figure 5 shows what the Console should look like when the application is successfully run. You can play around with the arguments in the operation functions. If you have successfully carried out the activity on the arithmetic operations you can now move on to the program flow operation.

```

int main()
{
    init_platform();

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

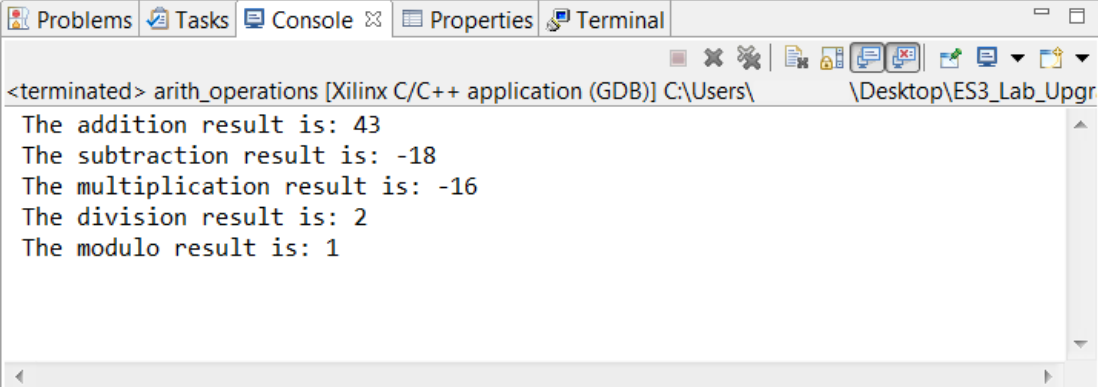
    add_result = adder(20, 23);
    sub_result = subtractor(1, 19);
    mul_result = multiplicator(-2, 8);
    div_result = divider(20, 10);
    mod_result = modulator(5, 2);

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

    cleanup_platform();
    return 0;
}

```

Figure 4



```

<terminated> arith_operations [Xilinx C/C++ application (GDB)] C:\Users\
\Desktop\ES3_Lab_Upgr
The addition result is: 43
The subtraction result is: -18
The multiplication result is: -16
The division result is: 2
The modulo result is: 1

```

Figure 5

II. Program Flow Control

Objectives

- To learn how to execute programs based on conditions
- To learn the concept of modularity and code reusability

Activity Summary

1. Create ProgFlowCntl 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 activities. A software processor like the MicroBlaze executes instructions in sequential order, that is, one after the other. Unlike in 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 the instructions were executed in sequential order, from top to bottom in any subroutine. For instance, in the `main()` method of Figure 4, the `init_platform()` function is executed first, then the `s32 add_result` declaration which reserves a memory space for the addition result, until the last instruction `return 0`. If there is however, 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. Some of these are the if-else, while loop, for loop, do-while loop, switch (case), break, and continue statements; and the ternary (?:) operator. We will use a few 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 1 shows the codes assigned to the arithmetic operations.

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

Table 1: 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 **ProgFlowCntl** and add a C source file with the name **prog_flow_cntl.c**. You can create this from the Hello World template as you did previously. The function `uartReceive()` that allows you to receive the operation code from the 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**. **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.**

Now, go back to the **prog_flow_cntrl.c** file in the C/C++ Editor and type the codes in Figure 6 in the **main()**. Remember to copy the arithmetic operation methods from the **arith_operations.c** file to the new application. You can either create a new source file with the methods and add it to the project or simply copy the methods and put them below the **main()** in your new code. Run the application after saving it, and interact with it by clicking and typing one of the operation codes in the Console.

```
char arith_op_type;
char value1, value2;

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 6

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.

The same functionality in Figure 6 can be achieved by using the switch statement. The structure of the code required to do this is given in Figure 7. You are required to complete this code. Moreover, instead of simply typing in the arguments of the arithmetic operations, you are required to get the numbers to be operated on from the system keyboard. You should also perform some necessary checks to confirm that the values received are numbers. Also, note that the value received from the keyboard is the ASCII equivalent of the character and you need to convert this to the actual number before carrying out the operation. The ASCII to integer (**atoi**) function can be used to do this; however, remember to include the **stdlib.h** library that has the definition of this function. **If for any reason the **atoi()** function does not work, you can simply subtract 48 (the decimal ASCII code for the number ‘0’) from the value received. This converts the value received from the ASCII code to the actual number.** You can use the char variables **value1** and **value2** already declared in Figure 6 to receive the numbers. The following steps are suggested:

- Read the ASCII value of the first number from the UART
- Convert the received ASCII character to a number
- Read and convert the second number
- Read the operation type
- Use the switch statement to decide which operation to perform

```
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 7

III. Interrupt Handling

Objectives

- To learn how to handle interrupts
- To learn how to execute programs in interrupt service routines (ISRs)
- To learn the concept of modularity and code reusability

Activity Summary

1. Create a new application called InterruptFlow for interrupt handling
2. Use the arithmetic operations in an ISR
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

There are usually two ways for an application to interact with an event external to the application. The first method called “Polling” relies on checking at a regular interval for the occurrence of the event. This puts a significant load on the processor especially if the polling interval is quite high. Also, there is a possibility that the event may be missed. On the other hand, an interrupt can be used, where the processor receives a dedicated signal only when the interrupt occurs. Apart from allowing the processor to do other useful tasks, interrupts guarantee that events cannot be missed. However, a major downside here is that the processor may be interrupted in the middle of a very critical task. One other catch with interrupts is that the developer has to be sure the interrupt can be fully serviced before the next one occurs. Overall, the decision of whether to use polling or interrupt depends on the type of application and the decision has to be made by the application developer (you in this case). In receiving the values entered on the PC keyboard, we are simply polling the UART waiting for data to be received. We can afford to that as our application is not critical and we do not have so much to process.

A hardware timer has been implemented for you on the FPGA. This timer gives an interrupt every 0.1 seconds. Our objective in this session is to perform arithmetic operations only when the interrupt occurs. The numbers to be operated on have to be obtained from the keyboard ahead of the interrupt. If the numbers and the operation type have not been received before the interrupt, the ISR should simply print an error message and return without performing the arithmetic operations. In order to use the interrupt you need to setup the interrupt system. A function (**int** **setUpInterruptSystem()**;) that does this has already been written for you. You also need to define a method in your application that will be executed only when the interrupt occurs. This method (**void** **hwTimerISR(void *CallbackRef)**;) has already been *declared* (but *not defined*) for you and passed to the processor as interrupt handler for the hardware timer. These two methods are in the file **xinterruptES3.c** located in the Lab 2 – Codes folder. You have to add this file to your project. You need to call the **setUpInterruptSystem()** method early in your code. You must also define the **hwTimerISR()** method in your application and put a code in it that you want executed when the timer interrupt occurs. The code structures required for this session have already been written for you. Copy the file named **interrupt_flow_1.c** located in Lab 2 - Codes folder and fill in the appropriate places. The following guideline should help you:

- Read the first and second numbers and the operation type from the UART [Hint: What you need here is the same code you developed in section II]
- Use the ISR to perform the arithmetic operations

Note that when a variable is modified by both the main code and an ISR, the keyword **volatile** has to be written before or after the data type declaration to prevent the compiler from optimizing useful and intended codes away. Moreover, since the arithmetic variables have to be accessed by the `main()` and the ISR, they have to be global variables (declared before the `main()`).

If you have run the application, you might have noticed that the error message comes in-between the typing of values and operation type on the keyboard if you are not fast enough. Definitely you cannot be fast enough as the interrupt is 0.1 seconds. This demonstrates to you what could happen if interrupts are not well handled. By the way, from the processor's perspective the 0.1 seconds interval is a lot of time considering that the processor is been clocked at a frequency of 100 MHz (this is the clock frequency supplied in the hardware design). With this frequency we have 10 million clock cycles in 0.1 seconds. As most instruction in the MicroBlaze instruction set execute in just one clock cycle, you could imagine how many instructions you can execute in 0.1 seconds. However, because of the human factor (you simply cannot type fast enough presumably), the 0.1 seconds is not enough. Now, modify the code to include an interrupt counter variable so that the ISR does not do the arithmetic operation until the counter value is 40. This has been done for you (see the code template named **interrupt_flow_2.c** in the Lab 2 - Codes folder). With this you will be ignoring 39 interrupts to have 4 seconds to enter the values. If you are still not fast enough, the error message will be displayed, but this time, you should be able to complete the value entering afterwards.