Dual Fixed-Point Calculator

Chandra Kasimkota, Oshin Gupta, Tahmid Uddin
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: kasimkota@oakland.edu , oshingupta@oakland.edu, tahmiduddin@oakland.edu

***Abstract*—In FPGA embedded systems, designers usually have to make a compromise between numerical precision and logical resources. For instance, floating point features a large dynamic range at the expense of a large resource usage. On the other hand, a fixed point requires fewer resources, but it delivers a low dynamic range. In this project, we will work on developing Dual Fixed-Point Calculator to show how using Dual Fixed-Point Arithmetic leads to higher precision by overcoming the limited dynamic range of fixed point while at the same time not requiring as many hardware resources as a floating point.**

Introduction

Standard Numerical Representations:
Fixed and Floating point:

Floating point features a large dynamic range at the expense of a large resource usage On the other hand fixed point requires fewer resources, but it delivers low dynamic range.

Dual Fixed point (DFX):

This numerical representation is a compromise between Fixed and Floating point number systems, it overcomes the limitations of Fixed and Floating point numbers. so the Dual Fixed-Point calculator project will use the DFX number system.

The DFX has the Exponent and Significand, depending on the exponent value DFX can have two scalings num0 and num1.

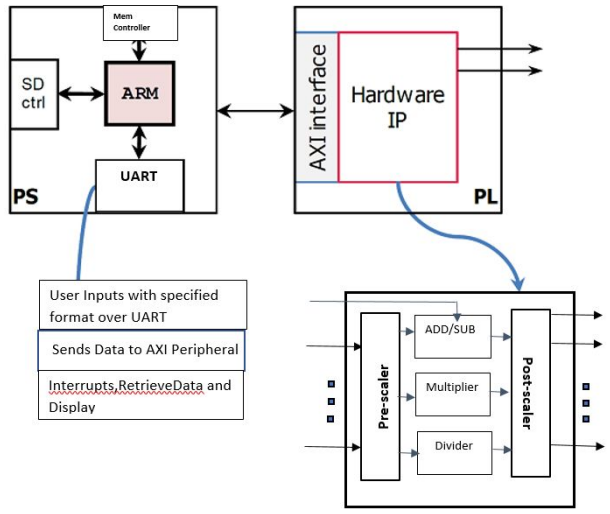
In this project, Dual Fixed- Point Calculator will be implemented which will include three hardware modules- adder/subtractor, multiplier and divider. The goal is to design, verify and implement four arithmetic operations – addition, subtraction, multiplication and division.

The input will be taken through the user via SDK Terminal and the final output will also be displayed on terminal. To understand the concept of DFX and implement adder/subtractor, we will be closely following the information in the units 'Computer Arithmetic' and 'Digital Design'[1][2].

The motivation for choosing this topic is to take advantage of DFX arithmetic over fixed point and floating-point arithmetic. Floating point (FP) and fixed point (FX) arithmetic are the standard numerical representations. Floating point features a large dynamic range at the expense of a large resource usage. On the other hand, a fixed point requires fewer resources, but it delivers a low dynamic range. Dual fixed point (DFX) arithmetic is an alternative representation that overcomes the limitations of FX (it greatly improves dynamic range) without the high resource complexity of floating point [1].

Methodology

DFX Calculator Architecture including the HW and SW:



Hardware Flow

-Input data is read from axi_wdata
-Digital systems perform processes(DFX Multiplier/DFX Adder)
-Output data of digital systems is sent to multiplexer(2to1)
-FSM will trigger output FIFO to read first set of data then the next set
-Data is output on axi_rdata

DUAL FX ADDER/SUBTRACTOR:

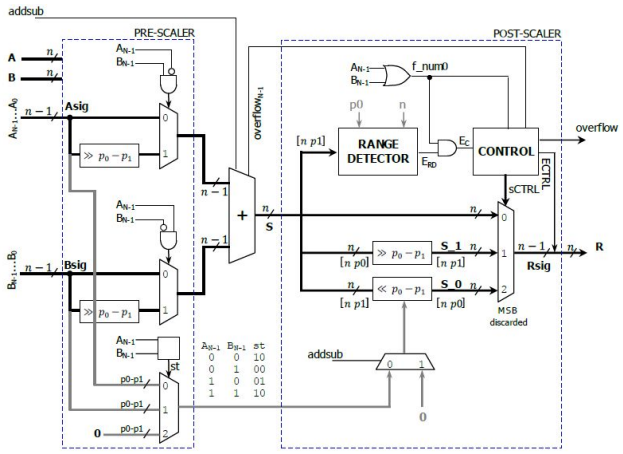


Figure 1 Architecture DUAL FX ADDER/SUBTRACTOR:[3]

DFX Adder/Subtractor is based on three main blocks- pre-scalar, fixed point addition/subtraction and post-scalar. In pre-scalar, we get rid of the exponent bits of the two n-bit operands. This module is required to analyze the operands and to determine in which range -po 0r p1- one must perform addition/subtraction. In the second module, it is determined which operation has to be performed. It basically depends on the value of 'addsub', value 0 corresponds to addition and 1 corresponds to subtraction. In the third module, which is post scalar, n-bit FX result is converted to n-bit DFX. In some cases, there can be overflow when the result cannot fit as a num1 in DFX[1].For this project, reference for the DFX adder subtractor is taken from Unit 7- Applications from the section Dynamic dual fixed point adder[3].

DUAL FX MULTIPLIER

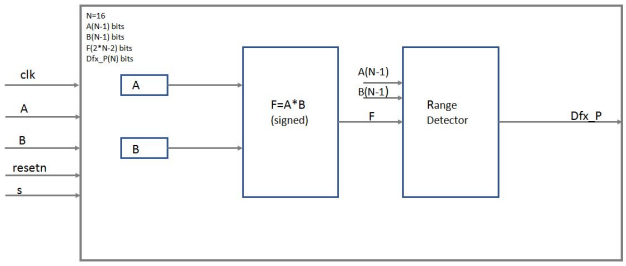


Figure 2 High level DUAL FX MULTIPLIER

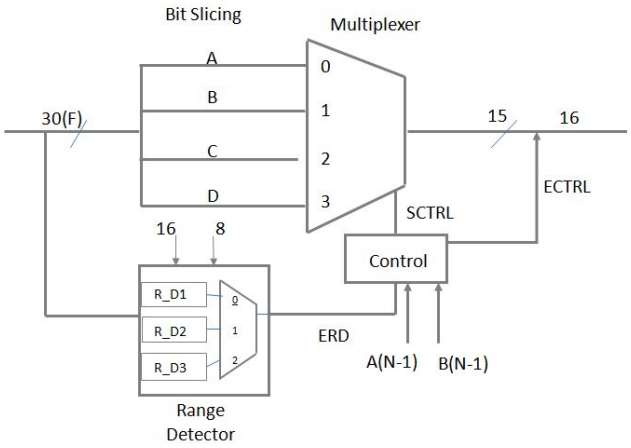
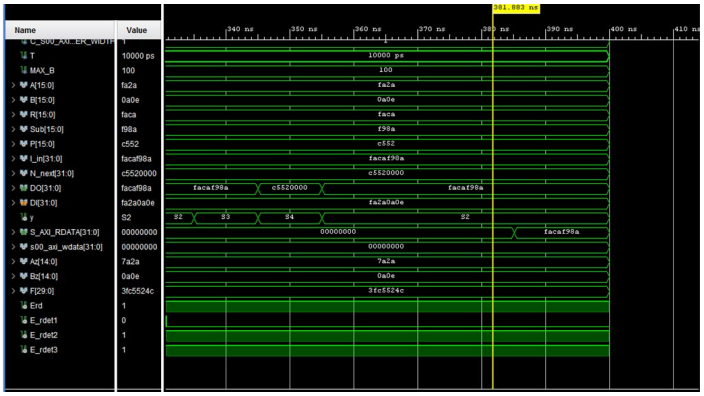


Figure 3 Range detector + bit slicing

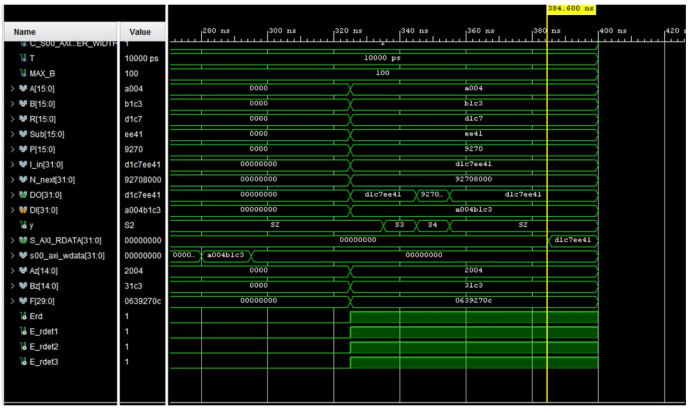
In DFX multiplier, signed multiplication is performed on two operands A and B. For signed multiplication, 'use ieee.std_logic_signed.all' library is included in the file. The product of signed multiplication is then fed into Range Detector. The range detector module is responsible for performing bits slicing based on the possible outcomes due to the input number representation, determining if there is overflow, determining if the num1 candidate can be represented as a num0 candidate and changing the sign of the candidates if the sign signal is asserted.

TESTBENCH SIMULATIONS

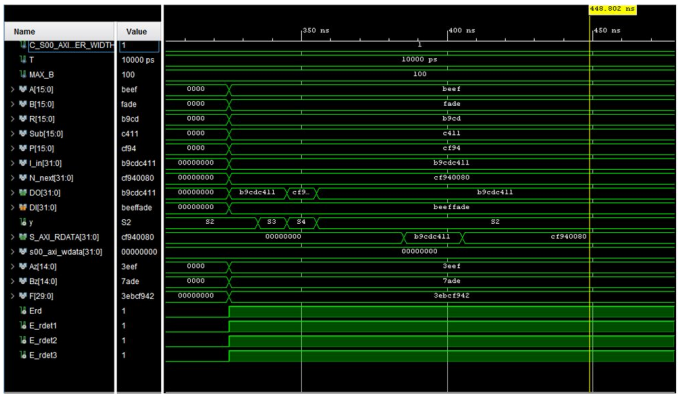
Input A = FA2A ; Input B = 0A0E



Input A= A004, Input B= B1C3



Input A= BEEF, Input B=FADE



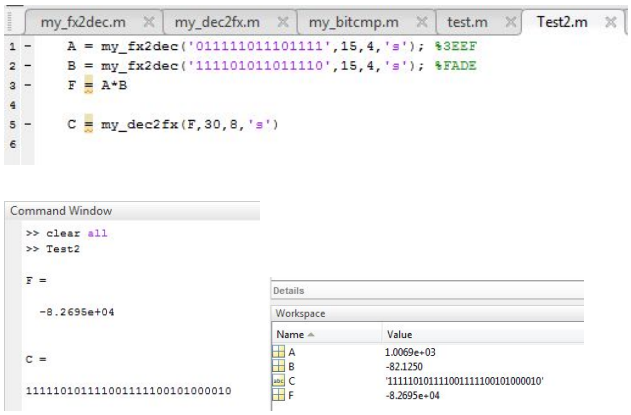
Verification of Peripheral

Manual:

Input A= BEEF, Input B= FADE

```
A=BEEF (1011 1110 1110 1111)
B=FADE (1111 1010 1101 1110)

A(n-1): 1,   B(n-1):1

F= A*B (signed multiplication)
 =(3EEF)*(7ADE)
 =3EBCF942   % 11111010111100111111001010000010

 The number is in format[30 8]
 For bit slicing, [30 8],E=1, 18 downto 4

 FX{30 8]->DFX [16 8 4]
 since num1, therefore,

 3EBCF942->1100111110010100 (CF94)
```

Matlab:



```
my_fx2dec.m | my_dec2fx.m | my_bitcmp.m | test.m | Test2.m

1 -    A = my_fx2dec('011111011101111',15,4,'s'); %3EEF
2 -    B = my_fx2dec('111101011011110',15,4,'s'); %FADE
3 -    F = A*B
4
5 -    C = my_dec2fx(F,30,8,'s')
6
```

```
Command Window
>> clear all
>> Test2

F =

  -8.2695e+04


C =

11111010111100111111001010000010
```

Workspace

Name	Value
A	1.0069e+03
B	-82.1250
C	'11111010111100111111001010000010'
F	-8.2695e+04

System Overview

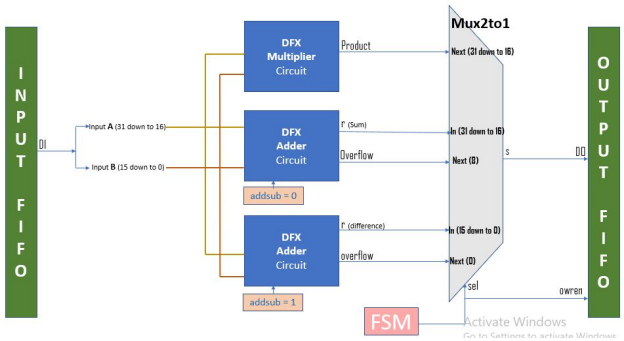


Figure 4 System Diagram

The output of the IFIFO (DI) consists of the two inputs (A & B) for which operations will be conducted on. The inputs will then feed into the digital system operation circuits and then the outputs will be allocated on the 2 to 1 multiplexor. The FSM will control when the OFIFO should read: first, the OFIFO will read the first set of data of the mux, once that is complete, it will then read the second set of data.

In the FSM below, once the input data is ready, it will be read and after calculations are performed, the output data will be sent to the multiplexer. At State 3, the output of the multiplexer will be read by DO, and then at State 4, the second set of data from the multiplexer will be ready by DO.

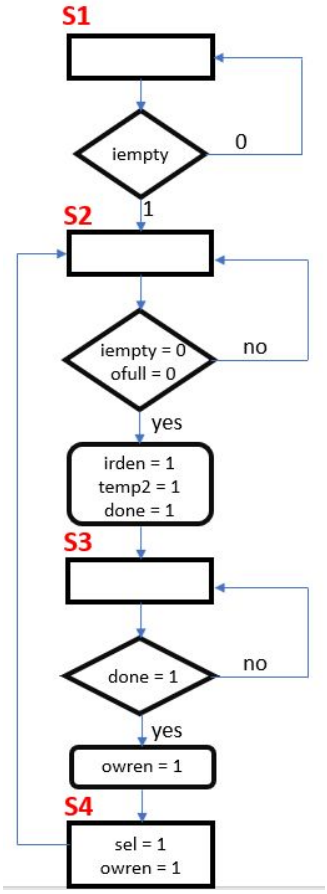


Figure 5 FSM

Software Flow

Overview

For the software portion of this project, the user inputs A & B in the code, for which then the various calculations will be presented. The results will then be printed in the SDK Terminal.

Software Application

1.) Take User for input of A & B.
2.) Send inputs as a dataset to AXI Interface.
3.) Read output from AXI Interface.
4.) Parse the data according to each operation.
5.) Determine if there is overflow.
6.) Print results to User.

Analysis of Code

```
//--------------------------------INPUT DATA HERE-------------------------------------
   //Enter Input A & Input B as 1 word.
   //Example: Input A = FA2A ; Input B = A0A0E ===> input = 0xFA2A0A0E

//  input= 0xFA2A0A0E; // no overflow
//  input= 0xA004B1C3; // has adder overflow
//  input= 0xBEEFFADE; // has subtracter overflow


//--------------------------------AXI WRITE DATA--------------------------------------

    MYDFXADDSUBINTR_mWriteMemory(baseaddr + 0*4, (input));
    xil_printf ("Input A = %04X, Input B = %04X\r\n", (input&0xFFFF0000)/0x10000, input&0x0000FFFF);

//--------------------------------AXI READ DATA---------------------------------------
    Mem32Value1 = MYDFXADDSUBINTR_mReadMemory(baseaddr);
    Mem32Value2 = MYDFXADDSUBINTR_mReadMemory(baseaddr);

    sum = (Mem32Value1&0xFFFF0000)/0x10000; //Parsing for solutions
    diff = Mem32Value1&0x0000FFFF;
    prod = (Mem32Value2&0xFFFF0000)/0x10000;
    of = Mem32Value2&0x0000FFFF; //Overflow

//-------------------------------PRINTING SOLUTIONS-------------------------------
    if (of == 0x00000100) //If Addition has OVERFLOW
    {
        xil_printf ("Sum = Overflow, Difference = %04X, Product = %04X\r\n", diff, prod);
    }

    if (of == 0x00000001) //If Subtraction has OVERFLOW
    {
        xil_printf ("Sum = %04X, Difference = Overflow, Product = %04X\r\n", sum, prod);
    }

    if (of == 0x00000000) //If NO OVERFLOW
    {
        xil_printf ("Sum = %04X, Difference = %04X, Product = %04X\r\n", sum, diff, prod);
    }
```

In the above code, we have four sections: User Input Data, AXI Write Data, AXI Read Data, and Printing Solutions.

For the User Input, the User will need to enter two inputs of 2 bytes each into one word.

e.g. If Input A is “FA2A” and Input B is “C1C3”, then the input entry should be “0xFA2AC1C3”.

In AXI Write Data & AXI Read Data, the data sent to the AXI peripheral and the output is received, where then the 32 bit output is parsed into the sum, difference, product, and overflow (allocations of each bit can be seen in the System Diagram).

Lastly in Printing Solutions, the results are printed to the user in the SDK terminal. A check is conducted to see if the overflow bits are set for either adder or subtractor. If so, the result for the respective operation will display “overflow”

The code is primarily based on the AXI Full Interface examples (e.g. AXI4 Full: Pipelined 2D Convolution).

SDK Terminal Results

Input and Output of no overflow

```
Input A = FA2A, Input B = 0A0E
Sum = FACA, Difference = F98A, Product = C552
```

Input and Output of adder overflow

```
Input A = A004, Input B = B1C3
Sum = Overflow, Difference = EE41, Product = 9270
```

Input and Output of subtractor overflow

```
Input A = BEEF, Input B = FADE
Sum = B9CD, Difference = Overflow, Product = CF94
```

Conclusion

When user writing two 16 bit numbers into an address, DFX calculator process the data for addition/subtraction and multiplication and display results over the serial COM port

Challenges

-Multiplier:

Implementing range detector for different combinations of operands(num0xnum0, num0xnum1 and num1xnum1).

Bit slicing concept execution.

Next Steps

-Adding the Divider Circuit: This would need for the creation of the digital system for the divider, as well as a larger multiplexer to account for another set of output data.

-Adding User Inputs: Initially, the system was to take user inputs from SDK Terminal but due to time constraints and setup issues, this was not able to be implemented.

The following code was put together:

```
Xil_printf("Enter 16 bit hex Input A \n")
unsigned char inputA;
inputA = XUartLite_RecvByte(BASEADDRESS);
//reading the data
Xil_printf("Enter 16bit hex Input B \n")
unsigned char inputB;
inputB = XUartLite_RecvByte(BASEADDRESS+1*4);
//reading the data
```

-Adding on-the-fly reconfiguration of P0 & P1: This would allow the user to set the variables. The effort will require adding a reconfigurable portion.

References:

[1]Daniel Llamocca, "Notes – Unit 1" in ECE-5736: Reconfigurable Computing. ECE Department, Oakland University.

[2]Daniel Llamocca, "Notes – Unit 2" in ECE-5736: Reconfigurable Computing. ECE Department, Oakland University

[3]Daniel Llamocca, "Notes – Unit 7" in ECE-5736: Reconfigurable Computing. ECE Department, Oakland University.

[4]Daniel Llamocca, "Notes – Unit 5" in ECE-5736: Reconfigurable Computing. ECE Department, Oakland University.

[5]Daniel Llamocca, "Notes – Unit 3" in ECE-5736: Reconfigurable Computing. ECE Department, Oakland University.