Distributed Arithmetic for the Design of High Speed FIR Filter using FPGAs

*Syed Shahzad Shah, Saqib Yaqub, and Faisal Suleman
Chameleon Logics, #301, Kiran Plaza F-8 Markaz, Islamabad
Email: *chameleon.logics@isb.comsats.net.pk*

***Abstract*--Implementing hardware design in Field Programmable Gate Arrays (FPGAs) is a formidable task. Computation algorithms are required that exploit the FPGA architecture to make it efficient in terms of speed and/or area. This paper describes Distributed Arithmetic (DA) technique for the implementation of a Finite Impulse Response (FIR) filter. Different techniques of DA are explored and implemented using VHDL in an FPGA. These implementations clearly show that the DA technique is the best choice for implementing FIR filters with Xilinx FPGAs. One of these implementations, namely, Serial Distributed Arithmetic (SDA) exhibits efficiency in terms of area. The other implementation, named Parallel Distributed Arithmetic (PDA) shows a very high sample rate. Linear-phase response FIR filters were also developed and implemented in an FPGA using the DA technique.**

***Index terms*—FPGAs, FIR Filter, Distributed Arithmetic**

I. INTRODUCTION

FPGA size has seen a tremendous growth in past years. In 1984, Xilinx launched the industry's first FPGA that could implement few thousands of gates. Today, an FPGA (Xilinx's Virtex) is capable of implementing one million gates. The reasons for such a quantum leap are simple:

- Technological improvements.
- Quick prototyping demands for complex digital systems targeting Application Specific Integrated Circuits (ASICs).
- Suitable for low volume, high-density and quick turn-around of complex digital systems.
- In-system reprogrammability.

FPGAs [1-4] are an array of programmable logic cells interconnected by a matrix of programmable switches. The logic cells communicate with the outside world using programmable input/output blocks. The architecture of FPGAs makes them suitable for dedicated functions like Digital Signal processing (DSP). Most of the DSP algorithms require multiplication and addition in real-time. The unit carrying out this function is called MAC (multiply accumulate). Three choices of technology exist for the implementation of DSP algorithms. These are:

- Programmable DSP chips.
- ASICs.
- FPGAs.

Programmable DSP chips typically have only one MAC unit that can perform one MAC in less than a clock cycle. DSP processors or programmable DSP chips are flexible, but they might not be fast enough. The reason is that the DSP processor is general purpose and has architecture that constantly requires instructions to be fetched, decoded and executed.

ASICs can have multiple dedicated MACs that perform DSP functions in parallel. But, they have high cost for low volume production and the inability to make design modifications after production makes them less attractive.

The FPGA architecture allows multiple MACs and pipelining. Their ability to be modified easily makes them an ideal candidate for DSP functions. The only drawback is the speed, and this can easily be overridden by using computational algorithms suitable for FPGAs.

Digital Filtering [5] is the process of spectrum shaping of signal waveforms for wide applications. Several authors [6, 7, 8, 9, 10] have implemented FIR filters in FPGAs. The maximum clock rate reported is 70MHz [10]. These authors have used different techniques of making efficient use of the FPGA architecture. This paper describes one such computational technique known as Distributed Arithmetic [10, 11]. The technique exploits the LUT scheme of the FPGAs for faster multiplication. The implementation of the DA technique in this work results in the clock rate of 130MHz. The reason for acquiring such fast speed is careful implementation and semi-hand-crafted layout. The sample rate is equivalent to the clock rate.

The paper is organized as follows: FIR filter theory will be discussed briefly in Section I. Usually the implication of number system in the implementation of the FIR filter is discarded [6-10]. Section III, therefore, will be devoted to the number system and its implications. The DA technique will be discussed in Section IV where equations will be developed for signed integer format in contrast to the equations developed for signed fractional numbers in [10, 11]. Section V will explore various implementations of the FIR filter using the DA technique. The area and speed efficient versions of these filters will

*Corresponding Author email: chameleon.logics@isb.comsats.net.pk. All the authors work for Chameleon Logics.
Tel: +92-51-262753.

be discussed. Equations will also be developed for these implementations. All the filters are implemented in XC4010XL-PC084 using Foundation Series 2.1i by Xilinx. Section VI will formalize the results, conclusions and the future directions of this work.

II. FIR FILTER

The FIR filter [5] is essentially a discrete convolution of the input signal with a set of coefficients. Mathematically, the FIR filter is defined as:

$$y(n) = x(n)h_0 + x(n-1)h_1 + x(n-2)h_2 + x(n-3)h_3 + \ldots . x(n-m)h_m \quad (1)$$

where y (n) is the output. x is the data input and h is the co-efficient. The signal flow graph of equation 1 is shown in figure 1.

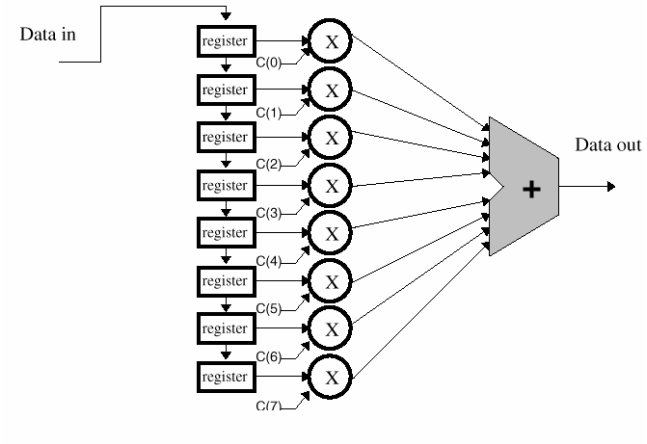


Figure 1: FIR filter

Figure 1 is an 8-tap filter. The signal flow graph and the mathematical equation clearly show that, there are registers, multipliers and an adder involved in the implementation of the filter. The efficient implementation of the above was carried out keeping in view, the architecture of the FPGA. In general, implementing any hardware algorithm for the FPGA is a two dimensional process. One dimension is the algorithm itself and the second dimension is the FPGA. The implementation of the algorithm demands a very clear and thorough understanding of the FPGA architecture so that the translation of the algorithm into the FPGA is carried out, efficiently. It should be noted well that since the architecture of the FPGA is fixed as opposed to gate array, therefore an algorithm might not be efficient at all for the FPGA.

III. NUMBER SYSTEM

a. Overview

The mathematical function is broadly classified into four functions i.e., addition, subtraction, multiplication and division. All these four operations can be executed in three operation modes corresponding to the three major types of number representation. These are fixed-point, decimal numbers and floating-point representation. Decimal data is used for arithmetic, shifting and editing operation on decimal data. Decimal data may be represented in either packed or unpacked format. Floating point representation is used for scientific purposes and can be normalized or unnormalized. Fixed-point binary numbers can have integer format, where the radix point is to the right of the number, fractional format, where the radix point is to the left of the number, and mixed format where the radix point is in the middle. This section presents definition of signed and unsigned fixed-point binary number system representations.

The discussion on number system is very important. The selection of binary number representation is step-one for the implementation of any digital hardware system. This is because different number systems require different way to manipulate them, which in turns needs modifications to the hardware system being designed. For example, the US "Patriot" anti missile system failed to track and intercept Scud missile due to loss in precision in converting integers to a floating-point number representation. The tracking system had velocity representation in 24-bit floating-point representation while the time was represented in 24-bit integer numbers. Scud velocity was determined by change in position with respect to time. Therefore, the time representation had to be converted into floating-point and then manipulated. This conversion resulted in loss of precision. The loss of precision was very high as the system kept running for 100 hours or so.

In general, there are six types of fixed-point number representation, which will be briefly touched, and some concepts of Finite Precision Math will also be delivered. In the discussion to follow, n is the number of bits, p is a subset of integers, b is the number of bits to the right of the decimal point and Z is the set of integers.

i. Unsigned Integers
Smallest number = 0
Largest number = $2^n - 1$

ii. Signed Integers
Smallest = $-2^n - 1$
Largest = $2^n - 1$

iii. Fractional Unsigned

iv. Fractional Signed

Smallest = 2^{-n} Smallest = -1
Largest = $1 - 2^{-n}$ Largest = $1 - 2^{-n}$

v. Mixed Format (Unsigned)
Smallest = 0
Largest = $\{ p \mid 2^b \mid 0 \le p \le 2^n - 1, p \in Z \}$

e. Mixed Format (Signed)
Smallest = 0
Largest=$\{ p \mid 2^b \mid -2^{n-1} \le p \le 2^{n-1} - 1, p \in Z \}$

b. Concepts of Finite Precision Math

i) Precision
Precision is the maximum number of non-zero bits representable. It measures the number of bits that can be used to represent each number. Precision determines the extent of errors caused by integer truncation or rounding called quantization error. For example, an A(13,2) number has a precision of 16 bits. For fixed-point representations, precision is equal to the word-length (where A is the signed number representation, a is the integer bits and b is the fractional bits).

ii) Resolution
Resolution is the smallest non-zero magnitude representable. For example, an A(13,2) has a resolution of $½^2 = 0.25$.

iii) Range
Range is the difference between the most negative number representable and the most positive number representable.

iv) Accuracy
Accuracy is the magnitude of the maximum difference between a real value and it's representation. For example, the accuracy of an A(13,2) number is 1/8. Note that accuracy and resolution are related as follows:
A(x) = R(x)/2
where A(x) is the accuracy of x and R(x) is the resolution of x.

v) Dynamic Range
It is the range of numbers that can be represented before overflow or underflow. Dynamic range determines where the signal needs to be scaled. It is the ratio of the maximum absolute value representable and minimum positive (i.e., non zero) absolute value representable. For A(a, b), dynamic range is 2^N and for unsigned fixed-point, dynamic range is $2^N - 1$. For N of any significant size, the "-1" is negligible and therefore signed and unsigned representations of the same word-length have practically the same dynamic range.
Therefore, the basic task of converting an algorithm into fixed-point arithmetic is that of determining the word length, accuracy, and range required for each of the arithmetic operations involved in the algorithm. Some of the filters developed in this paper have signed integer format with full precision. The careful study of the number system in the beginning also helps in the manipulation of overflows. If any other number system needs to be adapted suitable changes need to be made. It will be informative to add that the Texas Instruments TMS320C50 is a signed integer format DSP. Motorola 56K series performs arithmetic on signed fractional numbers.

IV. DISTRIBUTED ARITHMETIC

Distributed Arithmetic differs from conventional arithmetic only in the order in which it performs operations. The arithmetic sum of products that defines the response of linear, time-invariant networks (filters) can be expressed as [11]:

$$y(n) = \sum_{k=1}^{k} A_k x_k(n) \qquad (2)$$

where:
$y(n)$ = response at time n.
$x_k(n)$ = kth input variable at time n.
A_k = weighting factor of kth input variable that is constant for all n, and so it remains time invariant.
A_k for the FIR filter is the filter coefficients and we assume in this work that the coefficients are already known from programs like MATLAB or any other DSP tools for PLD [9]. An appropriate DSP-design tool helps you evaluate the filter's response characteristics. The number of filter taps, cutoff frequencies, roll-off characteristics, bits of resolution, and ripple in the passband and stopbands can all be determined with these programs.
It can be observed easily from equation 2 that a single output response requires the accumulation of K product terms. In DA, the task of summing product terms is replaced by table look-up procedures that are easily implemented in the FPGAs configurable logic block (CLB) look-uptable architecture.

We start by defining the number format of the variable to be signed integers. The variable, $x_k(n)$, may be written in the integer format as shown in equation 3.

$$x_k = -x_{k(B-1)}.2^{B-1} + \sum_{b=0}^{(B-2)} x_{kb}\,2^b \tag{3}$$

where x_{kb} is a binary variable and can assume values of only 0 and 1. A sign bit of value 2 **(B-1) is indicated by $x_{k(B-1)}$. Note that the time index, n, has been dropped because it is not needed to continue the derivation. The final result is obtained by first substituting eq. 3 into eq. 2:

$$\begin{aligned} y &= \sum_{k=1}^{k} A_k\left[-x_{k(B-1)}\cdot 2^{(B-1)} + \sum_{b=0}^{(B-2)} x_{kb}\cdot 2^b\right] = \\ &-\sum_{k=1}^{k} x_{k(B-1)}\cdot 2^{(B-1)}\cdot A_k + \sum_{k=1}^{k}\sum_{b=0}^{B-2} x_{kb}\cdot 2^b\cdot A_k \end{aligned} \tag{4}$$

Equation 4 can be expressed as:

$$\begin{aligned} y = &-\left[x_{10}\cdot A_1 + x_{20}\cdot A_2 + x_{30}\cdot A_3 + \ldots\ldots\ldots + x_{k0}\cdot A_k\right] \\ &+\left[x_{11}\cdot A_1 + x_{21}\cdot A_2 + x_{31}\cdot A_3 + \ldots\ldots\ldots + x_{k1}\cdot A_k\right]2^1 \\ &+\left[x_{12}\cdot A_1 + x_{22}\cdot A_2 + x_{32}\cdot A_3 + \ldots\ldots\ldots + x_{k2}\cdot A_k\right]2^2 \\ &\ldots \\ &+\left[\begin{matrix} x_{1(B-2)}\cdot A_1 + x_{2(B-2)}\cdot A_2 + x_{3(B-2)}\cdot A_3 + \ldots\ldots \\ + x_{k(B-2)}\cdot A_k \end{matrix}\right]2^{B-2} \\ &-\left[\begin{matrix} x_{1(B-1)}\cdot A_1 + x_{2(B-1)}\cdot A_2 + x_{3(B-1)}\cdot A_3 + \ldots\ldots \\ + x_{k(B-1)}\cdot A_k \end{matrix}\right]2^{B-1} \end{aligned} \tag{5}$$

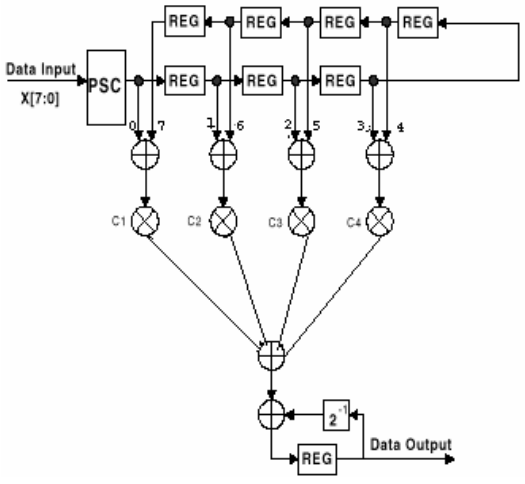
Each term within the brackets of equation 5 denotes a binary AND operation involving a bit of the input variable and all the bits of the constant. The plus signs denote arithmetic sum operations. The exponential factors denote the scaled contributions of the bracketed pairs to the total sum. A look-up table, can be constructed that can be addressed by the same scaled bit of all the input variables and can access the sum of the terms within each pair of brackets. Such a table will henceforth be referred to as a Distributed Arithmetic look-up table or DALUT [11]. DALUT technique eliminates the need of a multiplier in an FPGA. Add-shift or Booth multiplication techniques in FPGA are very slow due to the architecture of the FPGA. Complete working example of equation 1 and equation 5 can be found in Appendix A.

V. Digital Filter In an FPGA

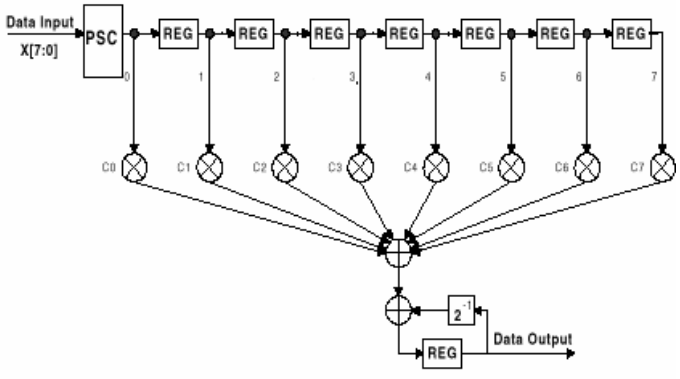
a. Overview

When building a digital Filter in an FPGA, the design can take advantage of the parallel structures and Distributed Arithmetic algorithms to exceed the performance of multiple general-purpose DSP devices. FPGA is one source used to implement and increase the function's data bandwidth and throughput by several orders of magnitude over off-the-shelf DSP solutions. In this section, we shall discuss different implementations of the 8-tap 8-bit FIR filter using the DA technique.

The 8-Tap FIR is a discrete-time filter in which the output is an explicit function only of the present and previous inputs to compute the weighted average of 8-bit sample points. Since the FIR response, mathematically, contains only feed-forward terms, the FIR is unconditionally stable and can be designed to exhibit a linear phase response. A symmetrical filter or a linear phase response filter is one with coefficient or tap weights symmetric about the mid point. Thus A_1=A_8, A_2=A_7, A_3=A_6, A_4=A_5 for the 8-Tap symmetric filter. A Non-symmetrical filter is the one that does not fulfill the above condition.



(a)



(b)

Figure 2: 8-bit-8-tap FIR Filter Signal flow diagram a) with symmetrical coefficients b) with non-symmetrical coefficients.

b. DA-based FIR Filters

When the target device is an FPGA having a LUT based structure, then for a non-symmetrical FIR filter the LUT contents for the b data bit are:

$$[sumb] = x_{1b} \cdot A_1 + x_{2b} \cdot A_2 + x_{3b} \cdot A_3 + x_{4b} \cdot A_4 + x_{5b} \cdot A_5 + x_{6b} \cdot A_6 + x_{7b} \cdot A_7 + x_{8b} \cdot A_8 + \ldots\ldots \quad (6)$$

Fortunately, the symmetry of the FIR filter can be exploited with an added advantage. This is a special feature of the DA technique. Pursuing the symmetric filter, the LUT content for the b data bit is: (we are now able to halve the number of coefficients).

$$[sumb] = (x_{1b} + x_{8b}) \cdot A_0 + (x_{2b} + x_{7b}) \cdot A_1 + (x_{3b} + x_{6b}) \cdot A_2 + (x_{4b} + x_{5b}) \cdot A_3 + \ldots\ldots \quad (7)$$

The sums within the parentheses represent the b output bit of the parallel adders that are summing the data appearing at symmetric filter taps.

The FPGA has the ability to implement a FIR filter function using one of the several DA techniques, depending on the performance required. Parallel Distributed Arithmetic techniques are used to achieve the fastest sample rates, while lower rates can be sustained with a Serial or Distributed Arithmetic techniques that uses less FPGA resources. In all the implementations, following resources of the FPGA are used:
4-input LUT - used as RAM (for shift register).
4-input LUT - used as ROM (for DALUT).
4/3-input LUT - used as combinational logic (for DALUT).
Registers - for pipelining.
From the implementation point of view, following settings in the Foundation 2.1i have been made:
Effort level (synthesis) : HIGH
Optimized for : SPEED
All outputs set to : Fast Slew rate
Place and Route level: HIGH
Delay-based clean-up: ON (Set to 5)

i) 8-Bit 8-Tap Symmetrical FIR Filter using Serial Distributed Arithmetic (SDA).

SDA technique results in an area efficient design of an FIR filter. The coefficients chosen for this design are $A_1=A_8 = -10$, $A_2=A_7 = 20$, $A_3=A_6 = 25$, $A_5=A_4 = 30$ which can be easily modified. The architecture following the general equation 7 is shown in fig. 3 [12] and implemented in VHDL. All the results are tabulated in table 1. The maximum number that should be stored in the LUT is 75, so there is no problem of overflow for an 8-bit number. The chip layout is shown in figure 4.

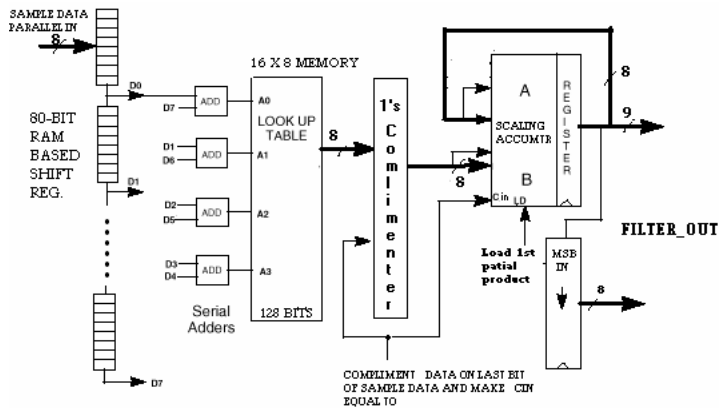


Figure 3: 8-Bit 8-Tap Symmetrical FIR Filter using SDA

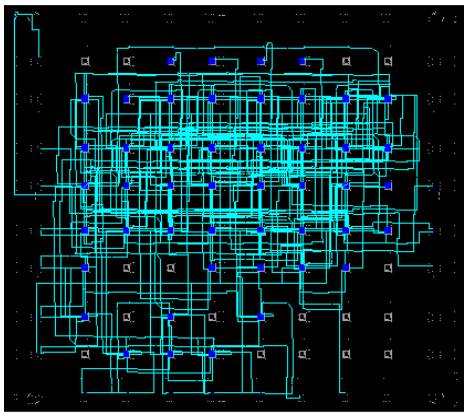


Figure 4: Layout of 8-Bit 8-Tap Symmetrical FIR Filter using SDA

ii) 8-Bit 8-Tap Symmetrical FIR Filter using Parallel Distributed Arithmetic (PDA).

As discussed earlier, the coefficients of a symmetrical FIR filter are equal from the midpoint. Reference equations 6 and 7 we can write:

$$(sumb) = x_{1b} \cdot A_1 + x_{2b} \cdot A_2 + x_{3b} \cdot A_3 + x_{4b} \cdot A_4 + x_{5b} \cdot A_5 + x_{6b} \cdot A_6 + x_{7b} \cdot A_7 + x_{8b} \cdot A_8$$

$$(sumb) = (x_{1b} + x_{8b}) \cdot A_1 + (x_{2b} + x_{7b}) \cdot A_2 + (x_{3b} + x_{6b}) \cdot A_3 + (x_{4b} + x_{5b}) \cdot A_4$$

The sums within the parenthesis represent the b output bit of the parallel adders that are summing the data appearing at the symmetrical filter taps. Special care must be taken to avoid any overflow in these adders. Nine bit adders instead of 8 can safely assure that an overflow will never occur.

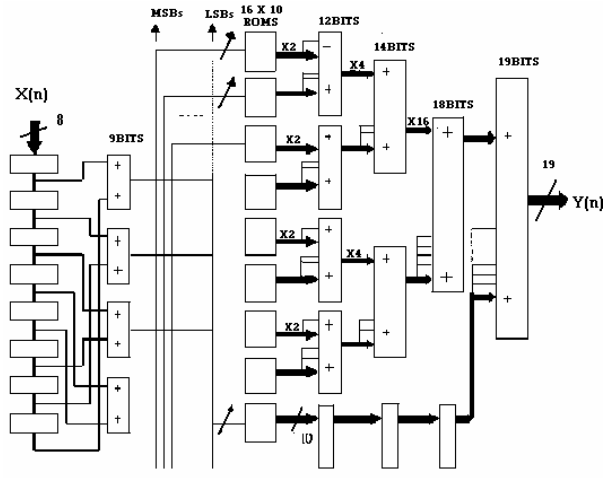


Figure 5: 8-Bit 8 Tap symmetrical FIR Filter using PDA

The weight of the sign bits increase by 1 i.e. 8. Nine LUTs are needed of 4-bit address i.e. number of LUTs increases by 1 (as B=9) but LUT size reduces to 4-input (as K=4) in place of 8. The symmetry of FIR filter exploited to a great extent! Bits of same weight from adder's output are applied at the LUT inputs in pairs of four.

The input data is loaded parallel (8 bits at a time) into an 8-bit register that can shift its contents to the next register. A chain of 8 such registers are provided. Word growth between stages is indicated by sign extensions, scaling by powers of 2 means that the adder sizes increase. Although 8-bit output precision would have been sufficient in many cases, a full precision of 19 output bits is provided. Note that in order to provide the output of the lower LUT at proper times to the input of the-19 bit adder, three stages of pipeline registers are inserted in between. This synchronizes all LUTs outputs. The chip layout is shown in figure 6. Compare with figure 4 where the CLB count is low and less routing resources are used. The technique is the same as in (a) but parallel nature of the technique is exploited. This filter is the fastest among all the FIR filters so far implemented [6-10].

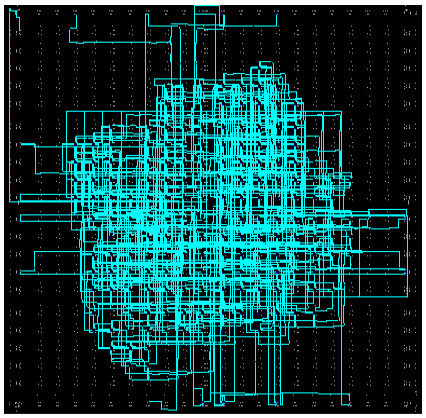


Figure 6: Layout of an 8-Bit 8-Tap FIR Filter using PDA

iii) 8-Bit 8-Tap non-Symmetrical FIR Filter using Serial Distributed Arithmetic (SDA).

When the filter has non-symmetrical coefficients there is no need of adders at the input of LUTs, however the size of LUT increases significantly (256x8 ROM). To compensate for this increase in hardware (CLBs) we use two LUTs of 16x8 and an adder instead of using a 256x8 LUT.The architecture of the filter, following equation 6 is given in figure7.

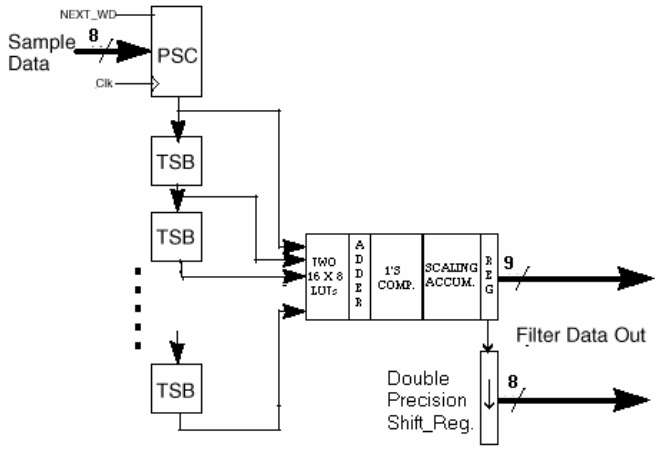


Figure 7: 8-Bit 8-Tap non-symmetrical FIR Filter using SDA

iv) 8-Bit 8-Tap non-Symmetrical FIR Filter using Parallel Distributed Arithmetic (PDA).

Reference equation 5, for 8 bits (B=8) and 8 taps(k=8) we can write the exact equation as:

$$\begin{aligned} y &= (sum0) + (sum1)2^1 + (sum2)2^2 + (sum3)2^3 + ... \\ &+ (sum6)2^6 - (sum7)2^7 \\ y &= \{(sum0) + (sum1)2\} + \{(sum2) + (sum3)2\}2^2 + \\ &\{(sum4) + (sum5)2\}2^4 + \{(sum6) - (sum7)2\}2^6 \\ y &= [\{(sum0) + (sum1)2\} + \{(sum2) + (sum3)2\}2^2] + \\ &[\{(sum4) + (sum5)2\} + \{(sum6) - (sum7)2\}2^2]2^4 \end{aligned} \quad (8)$$

Instead of using 8 address lines for LUTs (ROMs), we used two LUTs of 4 address lines each and an adder to generate the partial sums [(sum0) to (sum7)] because this technique uses fewer resources (CLBs).

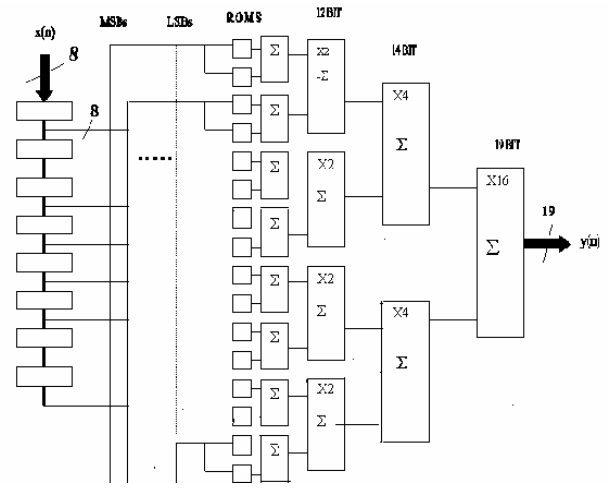


Figure 8: 8-Bit 8-Tap non-symmetrical FIR Filter using PDA

v) Results of all the implementations

The FIR filters implemented are tabulated for reference and guidance in table 1.

DA Technique	No. of CLBs	Max. Freq. (MHz)	Precision	Sample rate per second	FFs	No. of output bits	No. of gates
SDA(Sym.)	41	32	Not Full	$32x10^6$	36	17	1271
PDA(Sym.)	180	130	Full	$130x0^6$	341	19	6779
SDA(NonSy)	55	25	Full	$25x10^6$	40	19	1567
PDA(NonSy)	208	110	Not Full	$110x0^6$	391	17	9376

Table 1

VI. CONCLUSIONS AND FUTURE DIRECTIONS

Implementing an algorithm into the FPGA requires lots of effort. With reference to the FIR filter developed in this work, the activities involved in the successful implementation 4of efficient FIR filters are:

- Select the coefficients and study the filter characteristics by using MATLAB or DSP toolbox for PLDs (Only this step was not performed during this work).
- Select the FPGA and the algorithm for implementation. The efficiency of the end device will depend on this step. Comment: Experience in computational algorithms pays during this step.
- Carefully construct the architecture in VHDL/Verilog so that overflow, underflow are avoided and quantisation errors, if any are within acceptable limits.
- Synthesize the HDL description with reference to the FPGA.
- Place and route the design using PAR tool.
- Floor plan the design (auto or hand-craft) using Floorplanner.
- Timing analysis using Timing analyzer.
- Download the design into the FPGA with a cable.

All the above-mentioned steps require some degree of expertise. The advances in the EDA have made steps 4-8 quite easy. However, for an extra ordinary performance, you need to constraint your design and floor plan it yourself.

Symmetrical and non-symmetrical filters were implemented during this work. These filters were based on Parallel Distributed Arithmetic for very high speed and high gate count and Serial Distributed Arithmetic for low speed and low gate count. Maximum working frequency of 130MHz was attained. Based on the coefficients and the input data, the adder size can further be reduced and therefore, the FPGA resources and delay could be reduced. The number of outputs can be manipulated for the size of the coefficients. The least significant bits can be discarded, called as truncation, which will induce errors. This results in output noise.

ACKNOWLEDGEMENTS

The authors would like to say thanks to Charles R. Yates for his advice and expertise on the number systems with reference to the FIR filters. We are also thankful to the Application Engineers from Xilinx who helped during this work.

REFERENCES

[1] Stephen Brown, Jonathan Rose, "FPGA and CPLD Architectures: A Tutorial," *IEEE Design and Test of Computers*, vol. 13, no. 2, pp. 42-57, Summer 1996.

[2] Stephen Trimberger, "A Reprogrammable Gate Array and Applications," *Proceedings of the IEEE*, vol.81, no. 7, pp. 1030-1041, July 1993.

[3] Jonathan Rose, Abbas El Gamal, Alberto Sangiovanni, "Architecture of Field-Programmable Gate Arrays," *Proceedings of the IEEE*, vol.81, no. 7, pp. 1030-1041, July 1993.

[4] Stephen Brown, "FPGA Architectural Research: A Survey," *IEEE Design and Test of Computers*, vol. 13, no. 4, pp. 9-15, Winter 1996.

[5] N. K. Bose, *Digital Filters: Theory and applications*, 1985.

[6] Joseph B. Evans, "Efficient FIR Filter Architectures Suitable for FPGA Implementation," *IEEE Trans. Circuits and system,* July 1994.

[7] Chi-Jui Chou, Satish Mohanakrishnan, Joseph B. Evans, "FPGA Implementation of Digital Filters," *Proceedings 1993 Int. Conf. Signal Proc. App. And Tech,* 1993.

[8] Raymond J. Andraka, "FIR Filter Fits in an FPGA using a Bit serial Approach," *Proceedings 1993 Int. Conf. Signal Proc. App. And Tech,* 1993.

[9] Doug Conner, " Fast and Flexible: FIR filters in reconfigurable logic," *EDN*, July 4, 1996.

[10] Bernie New, " A distributed arithmetic approach to designing scalable DSP chips," *EDN*, August 17, 1995.

[11] Mintzer, L., " FIR filters with the Xilinx FPGA," *FPGA'92 ACM/SIGDA Workshop on FPGAs*, pp. 129-134, 1992.

[12] Newguard Bruce, " Seminar: Signal Processing with Xilinx FPGAs" *Xilinx Publications*, June 1996.

APPENDIX A

In this appendix, the authors would like to present a working example of the DA technique and the FIR filter. To keep things simple, we shall use 4-tap, 4-bit FIR filter based on SDA technique.

Let us suppose that our coefficients are:
Y_A= -2 =1110.
Y_B= -1 =1111.
Y_C= 7 = 0111.
Y_D= 6 =0110.
The LUT will have the following entries:

Address	**Function**	**Value**
0000	None	000000
0001	Y_A	111110
0010	Y_B	111111
0011	$Y_B + Y_A$	111101
0100	Y_C	000111
0101	$Y_A + Y_C$	000101
0110	$Y_C + Y_B$	000110
0111	$Y_C + Y_B + Y_A$	000100
1000	Y_D	000110
1001	$Y_D + Y_A$	000100
1010	$Y_D + Y_B$	000101
1011	$Y_D + Y_B + Y_A$	000011
1100	$Y_D + Y_C$	001101
1101	$Y_D + Y_C + Y_A$	001011
1110	$Y_D + Y_C + Y_B$	001100
1111	$Y_D + Y_C + Y_B + Y_A$	001010

Note that the values are stored with 6-bit numbers. This is done to avoid overflow. The addition of four 4-bit numbers require maximum of 6-bits to avoid overflow. Let us assume that at time 0, the input data = -5 =1011, the addresses for the LUT will be: 0001, 0001, 0000, and 0001. The corresponding values will be the output of the LUT as follows: 111110, 111110, 000000, and 111110. The MAC will do the calculations as follows:

11111110 2-bits of sign extension with 111110, the LSB is stored as a result's LSB
1111110 1-bit of sign extension
---------- addition
11111101 1-bit of sign extension, the LSB is stored as a result
0000000 1-bit sign extension
---------- addition
11111110 1-bit of sign extension, the LSB is stored as a result
0000010 1-bit sign extension, this is the 2's complement of 111110.
---------- addition
0000001 The 2-bits of MSB are discarded and therefore the result is 000001010 = 10 (as per equation

1). Let us say that the next data sample = 3 = 0011. The new addresses will be (note that the first data sample has moved to the next register): 0011, 0011, 0000, and 0010. The above technique could be employed to obtain a result of –1.