FPGA IMPLEMENTATION OF FFT ALGORITHMS USING FLOATING POINT NUMBERS

Hilal Kaptan[1] Ali Tangel[1] Suhap Sahin[2]

[1] *Kocaeli University, College of Engineering, Department of Electronics and Communication Engineering, Veziroglu Yerleskesi, 41040, Izmit, Turkey*

[2] *Kocaeli University, College of Engineering, Department of Computer Engineering, Veziroglu Yerleskesi, 41040, Izmit, Turkey*

[1] *e-posta: hilalkaptan@gmail.com* [2] *e-posta: suhapsahin@kou.edu.tr* [1] *e-posta: atangel@kou.edu.tr*

Keywords: FPGA, Fast Fourier Transform, Floating Point Arithmetic, DSP

ABSTRACT

In this paper, it is shown that FFT algorithms using floating point numbers can be implemented on an FPGA. A custom VHDL library so called Fp_Lib is formed for general purpose computer arithmetic FPGA implementations. This library includes addition, subtraction, multiplication and division modules based on 32 bit single precision IEEE 754 format. The algorithms are implemented on Xilinx Spartan-2 evaluation board, while FFT algorithms are realized on Xilinx's Virtex 2 FPGA (XUP-V2Pro Board). 0.6 µs, and 0.72 µs speeds were obtained for implementing the FFT and IFFT algorithms, respectively. The study is also compared with a similar one in the literature from structural and performance point of view.

1. INTRODUCTION

Fast Fourier Transform (FFT) plays an important role in many signal and image processing, data analyzing for vibration sensors, frequency measurement of earthquakes, and telecommunication systems such as WiMax technology which presents both wide bandwidth and wireless solutions[1-2].

In real time applications, it is necessary to obtain and process the input data as fast as possible to be able to reach the result almost simultaneously. Although ASIC solutions always offer fastest and low power solutions for real time applications, they are unique designs for a specific application. Therefore redesign process of an ASIC for a new application requires much more money and time when comparing with field programmable chips.

FPGA solutions also provide flexible design, low cost, and faster time-to-market features besides allowing parallel process implementations [4]. Due to parallel processing property, they are much faster than traditional microprocessor based solutions [5].

Floating point numbers have ability to represent a good approximation and dynamic range representations of the real numbers, so that floating point algorithms are frequently used in modern applications, which require millions of calculations per second, such as image processing and speech recognition.[3]

In this study, firstly, the realized algorithms of the necessary arithmetic operations used in FFT implementation are presented. Next, these design blocks are used to realize the mathematical expression of the FFT. Finally, the study is compared with the similar ones in the literature from structural and performance point of view.

2. FLOATING POINT ARITHMETIC ALGORITHMS

Many technological systems prefer floating point arithmetic due to having capacity of dynamic and precise representation for numbers [7]. FPGA usage for the implementation of Floating Point Number implementations rather than microprocessor based structures will be the best choice due to parallel processing capability, re-programmability, and higher speed. In spite of their advantage, floating point operators consumes large amount of resources and more time even for an ordinary (low resolution) implementation [6].

2.1. Floating point addition and subtraction

Figure 1 shows the design flow chart of the floating point addition and subtraction algorithm implemented. These algorithms are similar to the ones realized in many processors. Let F_1 and F_2 represent two floating point numbers; F_{top} represents the addition of both number; and $F_{minus} = F_1 - F_2$. F_{minus} can be re-written as $F_{minus} = F_1 + (-F_2)$. The subtraction process is converted to addition form by inversing the sign bit of F_2. For this reason, only addition algorithm is elaborated here. Addition and subtraction algorithms are realized in three steps. F_i represents the number; S_i is the sign, e_i is exponent and f_i is the fraction part of any number. Lets define the inputs as $F_1=(s_1,e_1,f_1)$ and $F_2=(s_2,e_2,f_2)$. The result is represented as $F_{ans}=(s_{ans},e_{ans},f_{ans})= F_1+F_2$ or $F_1+(-F_2)$

The algorithm steps are as follows:

1. Step:

If absolute value of F_1 is smaller than F_2, F_1 and F_2 are interchanged. The right shift amount of f_2 is calculated by subtracting e_1 from e_2. “1” is added to the bits after the sign bit $(1.f_1)$ ve $(1.f_2)$.

2. Step:

$(1.f_1)$ is shifted to the right by the amount of $(e_1- e_2)$. If the sign bits are equal, then $(1.f_1)$ and $(1.f_2)$ are added, if not $(1.f_2)$ is subtracted from $(1.f_1)$. The sign of the resulting number s_{ans} is the sign of the bigger f number.

3. Step:

f_{ans} is shifted to the left until the first bit becomes “1”, and amount of the shift is calculated. e_{ans} is obtained by subtracting the amount of shift from e_1.

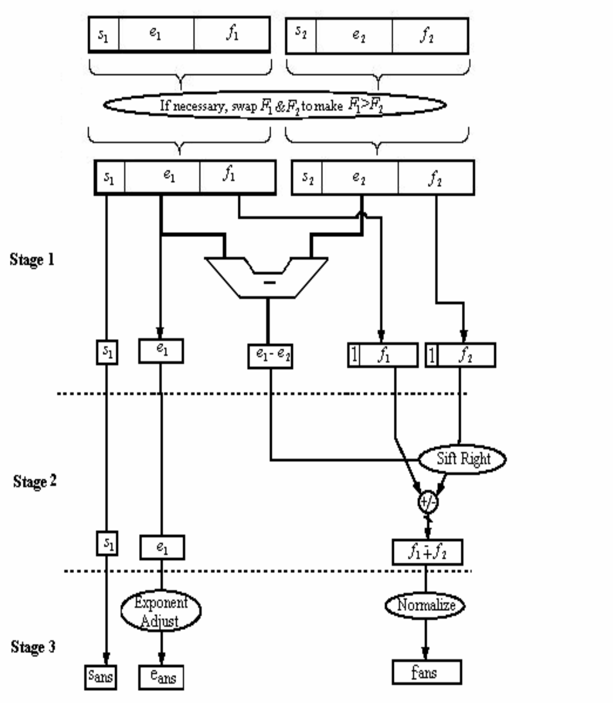

Figure 1. 32-bit floating point adding and subtracting algorithm implemented on an FPGA

2.2 Floating point multiplication

Floating point multiplication shown in Figure 2 is similar to the integer multiplication. Therefore FP multiplication is easier than FP adding or subtracting algorithms here. It is realized in three steps as well.

To make it easy, the algorithm never tests the illegal numbers or negative zero cases. The inputs are same as before, $F_1=(s_1, e_1, f_1)$ and $F_2=(s_2, e_2, f_2)$. The result will be $F_{ans} = (s_{ans}, e_{ans}, f_{ans})= F_1 * F_2$. The algorithm steps will be as follows:

1. Step:

Exponent parts, e_1 and e_2 are added; the resulting number is appointed as e_{ans}. “1” is added to the beginnings of f_1 and f_2, yielding $(1.f_1)$ and $(1.f_2)$.

2. Step:

$(1.f_1)$ and $(1.f_2)$ are multiplied and the first 23 MSB bits out of the resulting 45 bits is appointed as the final result, f_{ans}. The sign bit of the final number, s_{ans} is obtained by EXOR’ing the two numbers.

3. Step:

f_{ans} is shifted to the left until the first bit becomes “1”, and amount of the shift is calculated. e_{ans} is obtained by adding the amount of shift from e_1.

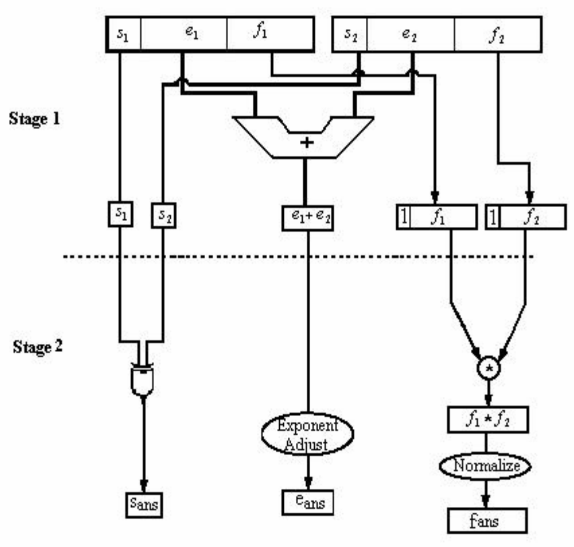

Figure 2. The algorithm flow chart of the 32-bit floating point multiplication implemented on FPGA

2.3. Floating point division

Assume F_1 and F_2 are two floating point numbers, and f_{ans} is the division of them. To make it ease, negative zero and illegal numbers are neglected here as well. The inputs are represented as:

$F_1=(s_1, e_1, f_1)$ and $F_2=(s_2, e_2, f_2)$. s_i is for sign bit ,e_i is for exponent bits, f_i is for fraction part of the FP number, F_i. The result becomes:

$F_{ans} =(s_{ans}, e_{ans}, f_{ans})= F_1/F_2$. The algorithm shown in Figure 3 can be explained as follows:

1. Step:

The exponent parts are subtracted from each other, and “1” is added to the beginnings of the fraction parts, yielding $(1.f_1)$ and $(1.f_2)$.

2. Step:

$(1.f_1)$ bits are shifted to the right by the amount of $(e_1 - e_2)$. If s_1 is equal to s_2, $(1.f_1)$ and $(1.f_2)$ are added. If not, $(1.f_1)$ is subtracted from $(1.f_2)$.

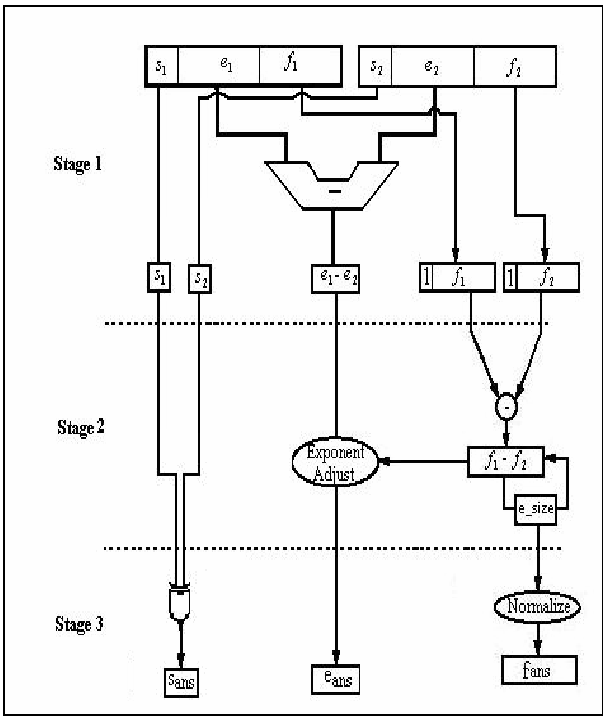

Figure 3. The algorithm flow chart of the division implemented on FPGA

3. Step:

f_{ans} is shifted to the left until the first bit becomes "1", and the amount of the shift is calculated. The sign bits of F_1 and F_2 are EXORed, and is appointed as s_{ans}.

3. FPGA IMPLEMENTATIONS OF FLOATING POINT ALGORITHMS

In this section, VHDL codes are not given, yet the results of hardware applications of the algorithms mentioned above are presented. Digilentic Evalution Board is used for the demonstration. This board includes Xilinx Spartan-2 having 2352 slices and 14 RAM block with 50 MHz clock speed [8].

A custom VHDL library so called *fp_lib* is formed for the hardware applications. The fp_lib has two different algorithms as listed in Table-1. The adding and subtracting algorithm results are summarized in Table-2.

Table-1. Summary of custom arithmetic VHDL algorithm

HDL Design	**Description**
Fp_lib	IEEE 32-bit single precision floating point library
Fp_mul	IEEE 32-bit single precision floating point pipelined parallel multiplier
Fp_add	IEEE 32-bit single precision floating point pipelined parallel adder/substractor

Table-2. 32 bits Floating Point adding/subtracting and multiplication algorithms results on selected FPGAs

	Add., Sub.		**Multiplication**	
Selected Device:	Spartan-2	%	Spartan-2	%
Number Of Slices:	387 out of 2352	13	326 out of 2352	13
Number Of Slice Flip Flops:	106 out of 4704	2	65 out of 4704	1
Number Of 4 Inputs LUTs	903 out of 4704	15	642 out of 4704	13
Number Of Bonded IOBs	103 out of 146	70	103 out of 146	70

4. FAST FOURIER AND INVERSE FOURIER TRANSFORM METHODS

4.1. Fast Fourier Transform (FFT)

Discrete Time Fourier transform provides frequency domain representation for a signal. FFT is an important algorithm to calculate Discrete Fourier Transform (DFT). Discrete Fourier transform of a signal is directly calculated from: X[k]= $\sum_{n=0}^{N-1} X[n] e^{-j(2\Pi/N)kn}$. k=0,1,….,N-1 (4.1)

Here, the phase factor is defined by: $W_N = e^{-j(2\Pi/n)}$

FFT provides a fast calculation strategy by using symmetry and periodicity properties of the phase factor to calculate DFT. As a calculation method, decimation in time is used. This means a remarkable savings over direct computation of the DFT.

4.2. Decimation in time

N – Radix DFT is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, W_N^{kn} \quad \text{, k=0,1,…,N-1}$$

Decimation in time algorithm rearranges the discrete Fourier transform (DFT) equation into two parts: a sum over the even-numbered discrete-time indices n=[0,2,4,...,N–2] and a sum over the odd-numbered indices n=[1,3,5,...,N–1]. These samples are used to calculate Radix-2 DFT.

These transformations are combined according to Equations (4.1) and (4.2), which lead to the calculation of upper level DFT. When sample indexes are handled in binary format, firstly the index bits are reversed before grouping them in pair as shown below. Next, these pairs are inserted into DFT process. Let's assume the input as: x[n]=[1 3 0 2], then

In-order index		Bit reversed index	
Decimal	Binary	Binary	Decimal
0	00	00	0
1	01	10	2
2	10	01	1
3	11	11	3

$x_t[n]=[1\ 0]$ $x_c[n]=[3\ 2]$. By using the DFT definition of these samples, Fourier transformations are obtained as: $X_t[n]=[1\ 1]$ and $X_c[n]=[5\ 1]$.

On the next step, DFT with N=4 of x[n] having 4 samples are obtained by referring the symmetry and periodicity nature of the phase factor. As a result, the following equations are obtained [9].

$$X[k]= X_t[k]+W_N^k X_c[k] \qquad (4.1)$$

$$X[k+N/2]= X_t[k] - W_N^k X_c[k] \qquad (4.2)$$

In this case,   X[k]=[6 1-j -4 1+j] is calculated.

As a result, this simple reorganization and reuse has reduced the total computation by almost a factor of two over direct DFT computation [9].

4.3. Inverse FFT

The inverse Fourier transform maps the signal back from the frequency domain into the time domain. Recall that the equations for an 8-point DFT and Inverse FFT are as follows:

$$X_t[k]=\frac{X[k]+X[k+4]}{2}$$

$$X_ç[k]=\frac{X[k]-X[k+4]}{2W_N^k} \quad , \text{ k= 0,1,2,3}$$

$$X_{tt}[k]=\frac{X_t[k]+X_t[k+2]}{2}$$

$$X_{tç}[k]=\frac{X_t[k]-X_t[k+2]}{2W_{N/2}^k} \quad , \text{ k= 0,1,2,3}$$

$$X_{çt}[k]=\frac{X_ç[k]+X_ç[k+2]}{2}$$

$$X_{çç}[k]=\frac{X_ç[k]-X_ç[k+2]}{2W_{N/2}^k}$$

4.4. FFT Implementation Process

32-bit inputs which have both imaginary and reel parts together with phase factor values calculated using MATLAB are loaded into the registers in the fft.vhd module as shown in Figure 4.

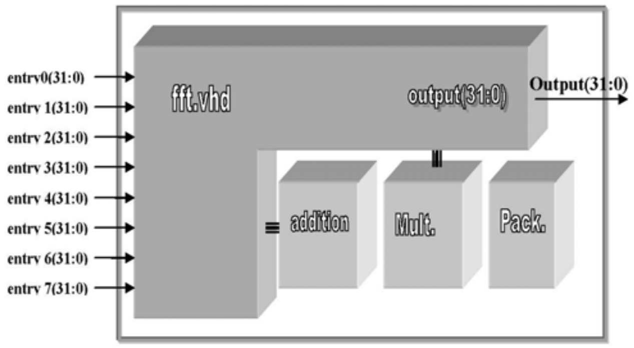

Figure 4. Architectural model of FFT implemented on FPGA

These inputs placed into the registers are firstly separated as odd ones and even ones to be used in decimation process. Next, these data are then sent to the addition module, where their sum and difference values are calculated to obtain their 2-point DFTs. Thirdly, the processed data are serially sent to the multiplication and addition modules by using a counter in a timely manner. By referring to the equation: $X[k]= X_t[k]+W_N^k X_c[k]$ $X_c[k]$ data and the phase factor are multiplied, and the result is added to $X_t[k]$ data. The obtained final result is latched into the output registers in fft.vhd module. The package module is used to introduce the sub- modules to the main program. Table-3 summarizes the hardware synthesis results on Xilinx Virtex II-Pro evaluation board. The experimental set up photo is shown in Figure 5.

Table-3. FFT synthesis results on the FPGA

Selected Device :	**Virtex II-Pro**	**%**
Number Of Slices:	1989 out of 13696	14%
Number Of Slice Flip Flops:	896 out of 27392	3%
Number Of 4-Input LUTs	3627 out of 27392	13%
Number Of Bonded IOBs	33 out of 556	5%
Number of MULT18x18s	16 out of 136	11%
Number of GLCKs	1 out of 16	6%

CONCLUSION

This paper shows that floating point algorithms can be implemented on an FPGA. Processing steps of the required algorithms are explained in detail. As an example of the algorithms, application of FFT and inverse FFT methods are examined. For the representation of digital numbers, IEEE 754 single precision floating point number format was used, and the algorithms are realized based on this format.

This situation inevitably brings about large amount of FPGA resource usage. Hence, some parts of the algorithms have to be embedded in to the FPGA in serial (sequential) nature. This situation means sacrificing the fully parallel usage advantage of FPGAs. Despite its mandatory serial nature, better performance has still been observed when compared to its traditional processor based solutions. Real time working was almost succeeded. 0.6 µs, and 0.72 µs speeds were obtained for implementing the FFT and IFFT algorithms, respectively. In traditional processor solutions, this rate goes up to millisecond levels.

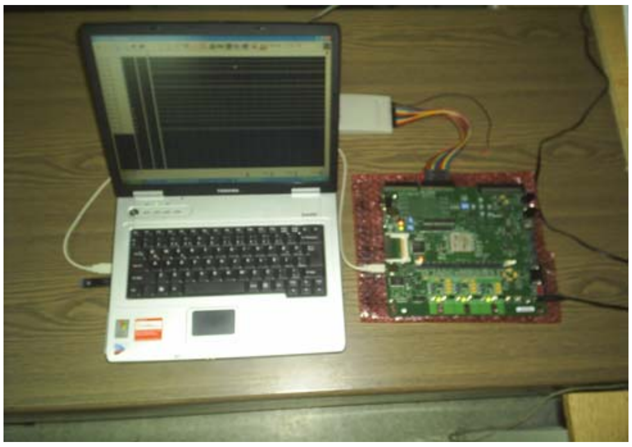
Figure 5. The Virtex-II Pro experimental setup

A similar study has been found from the literature, Haibing et. al [10]. Table 4 compares two designs.

Table-4 Comparison Table

	Architecture	Selected Device	Source Usage Logic Elements, LUTs	Performance
Haibing et. al (2006)	Hybrid Design DSP+FPGA LUT Based Division Algorithm	Altera Cyclone	Adder=883 Multiplication =995 Division =1505	40 ns for adder, multiplier, and division (Faster due to LUT usage for division algorithm)
Proposed	Full FPGA solution Subtractive Based Division Algorithm	Xilinx Spartan2 and Virtex-Pro II	Adder=903 Multiplication =642 FFT=3627	Adder=50ns Multiply=50ns Div.=150ns FFT=600nsec IFFT=720nsec

Moreover, in this study, the earlier VHDL library for implementing FFT algorithm [11], [12] has been modified. For this purpose, addition, subtraction, multiplication and division algorithms in the former library have been re-designed. Improvement on the source usage can easily be seen when compared to the ones declared in [11],[12]. In fact, faster Floating Point algorithms can be found in the literature. However, these algorithms consume much more hardware resources. Therefore, optimization of the source usage was the primary aim of this study.

REFERENCES

[1] E. O. Brigham, *The fast Fourier transform and its applications*, Prentice Hall, 1988.

[2] J. G. Pmakis, *Digital signal processing: principles, algorithms, and applications*., Prentice-Hall Intemational, 1996.

[3] W. B. Ligon, S. McMillan, G. Mpnn, F. Stivers, and K. D. Underwood "A Re-evelation of the Practicality of Floating Point Operations on FPGAs", *Proceedings, IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 206-215, Napa, CA, Apr. 1998. (ICANN'99).

[4] J. Zhu, B. K. Gunther, "Towards an FPGA Based Reconfigurable Computing Environment for Neural Network Implementations", *Proceedings of the Ninth International Conference on Artificial Neural Networks*,1999.

[5] M. Poliac, J. Zanetti, D. Salerno., "Performance Mesuraments of Seismocardiogram Interpretation Using Neural Networks", *Computer in Cardiology, IEEE Computer Society*, pp 573-576, 1993.

[6] S. Şahin, A. Kavak., "Implementation of Floating Point Arithmetic Using an FPGA", *Mathematical Methods in Engineering*. Editors K.TAS, J.A.T.Machado and D.Baleanu, Springer Book, 2007.

[7] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," *IEEE Transactions on VLSI*, Vol. 2, No. 3, pp. 365-367, September 1994.

[8] Xilinx Inc., *The Programmable Logic Data Book*, San Jose, California, 1993.

[9] S. Erturk , *İşaret İşleme*, Birsen yayınevi, 2005.

[10] H. Hu, T. Jin, X. Zhang, Z. Lu, Z. Qian, " A Floating-point Coprocessor Configured by a FPGA in a Digital Platform Based on Fixed-point DSP for Power Electronics", *IEEE IPEMC'2006*

[11] I. Az, S. Sahin, C. Karakuzu, M. A. Cavuslu, "Implementation of FFT and IFFT Algorithms in FPGA", *ISEECE-2006*, pp.7-10

[12] M. A. Cavuslu, S. Dikmese, S. Sahin, K. Kucuk, A. Kavak "Akıllı anten algoritmalarının IEEE 754 Kayan Sayı Formatı ile FPGA Tabanlı Gerçeklenmesi ve Performans Analizi", *III.URSI-Türkiye'2006*, pp.610-612.