A FPGA FFT CORE IMPLEMENTATION

Rolim, Arthur; Lima, Manoel

Federal University of Pernambuco, CIn

ABSTRACT

Advances in field programmable gate array (FPGA) technology, velocity, low power-consumption, short development time, low cost and parallelism, have led to dramatic improvements in single and double precision floating-point performance. This paper presents a Fast Fourier Transform (FFT) on FPGAs using IEEE 754 single and double precision float point arithmetic. A custom VHDL generator of float point arithmetic cores called FloCoPo is formed for general purpose computer arithmetic FPGA implementations. The algorithm is implemented with and without arithmetic improvements on the butterfly unit design and good difference between arithmetic cores quantities was found, witch results on great reduce of FPGA area. A pipelined version from the FFT was chosen, because the arithmetic cores are also pipelined and this reduces main core complexity and due to the fact that, in practice, FFTs often process a stream of data.

1. INTRODUCTION

Fixed-point applications have, for long time, been accelerated with FPGAs. As FPGAs technology matured, the size and the power of parallelism grew along with it. This was the trigger for many application designers to create more applications. It also meant that the performance of FPGAs was growing faster than that of CPUs[1].

In this scenario, floating point numbers, witch demands a great area and high velocity, now can be used with FPGAs. They have the ability to represent a good approximation and dynamic range for real numbers representations, so that floating point algorithms are frequently used in scientific applications, which require millions and millions of calculations per second, such as seismic calculations, image processing and speech recognition [2].

Discrete Fourier Transform (DFT) and its fast algorithm, Fast Fourier Transform (FFT), are the more important algorithms in digital signal processing. For a long time, FFTs were developed using fixed-number arithmetic in FPGAs, although for high precision scientific applications it wasn't accurate enough [3]. With the FPGA evolution, FFT is a next step in exploring the floating point capabilities [3].

This paper will present a brief study about the FFT, design stages for main core implementation and butterfly arithmetic optimizations for FPGA area reduction.

2. FAST FOURIER TRANSFORM

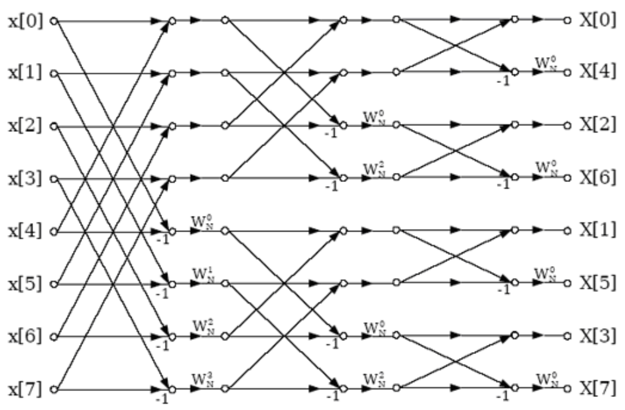

Figure 1. DIF FFT for a length-8 signal

The fundamental calculation of the N point DFT is described as:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}nk}, k = 0,1,..,N-1;$$

where the phase factor is $W_N = e^{-j\frac{2\pi}{n}}$

This algorithm is an optimized implementation of the DFT and provides frequency domain representation for a signal in time domain. The number of complex multiplication and addition operations, due to the phase factor, required by the simple DFT has order of N^2. The FFT reduces the number of computations needed for N points from $O(N^2)$ to $O(N\log_2 N)$. It provides a fast calculation strategy by using symmetry and periodicity properties of the phase factor to calculate the DFT.

The calculation is broken into small regular structures known as butterflies, which differ only in the constants they use. As a calculation method, decimation in frequency (DIF) is used. It rearranges the 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 transformations are combined according to equations (2.1) and (2.2), which lead to the calculation of upper level DFT.

$$X[k] = x[n] + x[n+\frac{N}{2}] \quad (2.1)$$

$$X[k+\frac{N}{2}] = \left(x[n] - x[n+\frac{N}{2}] \right) W_N \quad (2.2)$$

And these equations (2.1) and (2.2) are arranged into butterflies to calculate the FFT. Figure 1 shows this connections for N=8.

3. TESTBENCH PROCESS

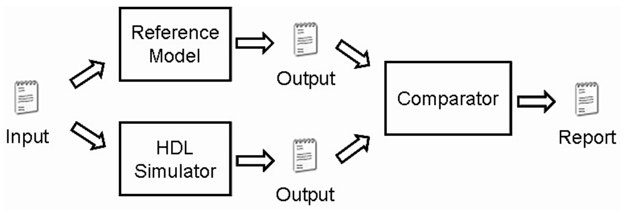

Figure 2. Testbench process

To guarantee that all developed cores are processing and the outputs are correct, a testbench process is used. Reference models in a high level language, like C++ or Java, are careful developed so that a great number of normal and critical situations can be reproduced.

Text files with the input stimulus are created for the cores simulation in a hardware description language (HDL) simulator and reference model. After running all the tests two output files are created: one from the reference model and the other from HDL simulator.

The two output files are compared and a report is created, pointing out if the core and the reference model behave in the same way for every stimulus. Figure 2 describe the whole testbench process.

4. FPGA FFT CORE IMPLEMENTATION

Using the testbench process described above, all cores were implemented and tested. Next section presents how the main cores were developed and describes some details from the internal cores architecture.

4.1. FloPoCo

The arithmetic cores used in this implementation were created with FloPoCo, a float point core generator. This tool grants a great flexibility, because an input receives core operation features, such as: clock, use of pipeline, simple or double precision and FPGA maker brand. The output is a synthesizer VHDL file code with the features of the input.

With this tool it was possible to implement a simple and double precision pipelined FFT without much rework only changing the core generator parameters.

4.2. Basic Complex Operations

Basic arithmetic operations, adding and multiplying, in the FFT are complex. So especial cores were developed to handle these operations.

4.2.1. Complex Adder

A complex sum is described as:

$$\begin{aligned} z_1 = a + bi;\ z_2 = c + di \\ z_1 + z_2 = (a+c) + (b+d)i \end{aligned} \quad (4.1)$$

The core was implemented using two adders from the core generator in parallel. The equation (4.1) shows how the complex adder is mathematically. Figure 3 shows the core structure.

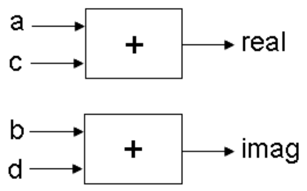

Figure 3. Complex Adder

4.2.2. Complex Multiplier

A complex multiplication is described as:

$$\begin{aligned} z_1 = a + bi;\ z_2 = c + di \\ z_1 \cdot z_2 = (ac - bd) + (ad + bc)i \end{aligned} \quad (4.2)$$

The core was built with four multipliers and two adders, two multipliers and one adder are connected in cascade to compute the real part and the others for the imaginary part, although the real part needs float point 'not' between the stages to adjust the result, as the equation (4.2) and figure 4 shows.

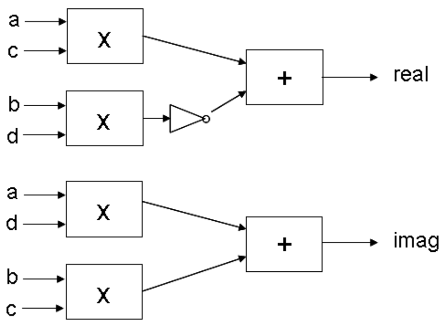

Figure 4. Complex Multiplier

4.3. Butterfly

The butterfly operation is the heart of the FFT and it's composed with two complex adders and one complex multiplier, the only difference between all of them is the 'w' input.

Due the 'w' periodic and sinusoidal nature, some values are always the same for every size of FFT, witch can lead to some arithmetic core usage reducing.

To maintain the synchronization of the pipeline FFT some FIFOs were introduced in the core, although for simplicity they were omitted from the blocks diagram.

4.3.1. Normal Butterfly

A normal butterfly operation is described as:

$$z'_{UP} = z_{UP} + z_{LOW}$$
$$z'_{LOW} = w \cdot (z_{UP} - z_{LOW}) \quad (4.3)$$

In this core the 'w' input can assume any float point value. It has two complex adders and a complex multiplier, as the equations (4.3) and the figure 5 shows.

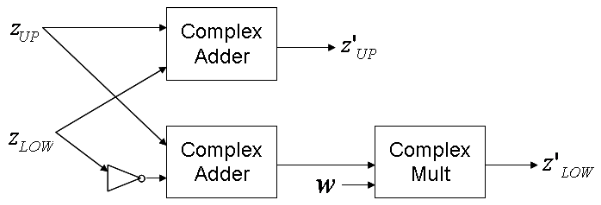

Figure 5. Normal Butterfly

4.3.2. Butterfly (1,0)

When the 'w' assumes the values w=1+0i, the equation (4.3) can be replaced for:

$$z'_{UP} = z_{UP} + z_{LOW}$$
$$z'_{LOW}\ z_{UP} - z_{LOW} \quad (4.4)$$

In this case, the butterfly can execute its operation without the complex multiplication, only using two complex adders, as the equation (4.4) and figure 6 shows.

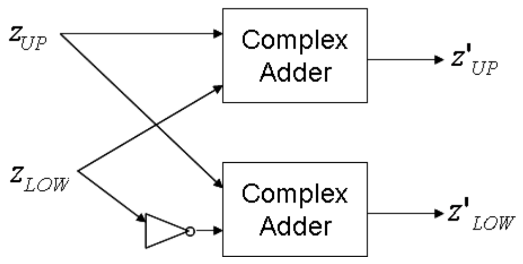

Figure 6. Butterfly (1,0)

4.3.2. Butterfly (0,-1)

When the 'w' assumes the values w=0-1i, the equation (4.3) can be replaced for:

$$z'_{UP} = z_{UP} + z_{LOW}$$
$$z'_{LOW} = -i \cdot (z_{UP} - z_{LOW}) \Rightarrow \quad (4.5)$$
$$-i \cdot (a + bi) = (b - ai)$$

The butterfly can compute its operation without the complex multiplication, but it will have to make some adjust on the core exit to archive the same result. It will have to interchange low subtraction result and use a float point 'not', as the equation (4.5) and figure 7 shows.

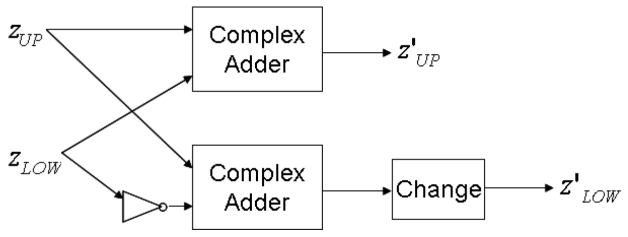

Figure 7. Butterfly (0,-1)

5. RESULTS

Following the figure 1 and the reference model FFT, a three pipelined FFT with simple and double precision were developed.

With the arithmetic optimizations in the butterfly unit, the core main cores of the FFT with N equals 2, 4 and 8 archived a good reduction of float point cores from the normal to the optimized version of the FFT, as the table 1 and 2 shows.

FP Cores Number	FFT-2	FFT-4	FFT-8
FP Adders	6	24	72
FP Multipliers	4	12	48

Table 1. Normal Version

FP Cores Number	FFT-2	FFT-4	FFT-8
FP Adders	4	16	52
FP Multipliers	0	0	8

Table 2. Optimized Version

6. CONCLUSION

In this paper, a pipelined float point FFT implementation on a FPGA was implemented and tested, but to archive full potential from the FPGAs architectures, there are many problems to be solved, like memory access and core size, until its usage became reality in scientific applications.

7. REFERENCES

[1] K. D. "Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance,". *In Proceedings of the ACM International Symposium on Field Programmable Gate Arrays, Monterrey*, CA, February 2004.

[2] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert, "Architectural Modifications to Enhance the Floating-Point Performance of FPGAs," *Sandia National Laboratories and the National Science Foundation*, USA, December 2006.

[3] O. Callanan; D. Gregg; A. Nisbet; M. Peardon, “High Performance Scientific Computing Using FPGAs with IEEE Floating Point and Logarithmic Arithmetic for Lattice QCD,” *International Conference on Field Programmable Logic and Applications*, Madrid, 2006.