FPGA Algorithm Design Exploration with Graphical Programming

Jim Lewis
National Instruments Corp.
11500 N. MoPac Expwy, Austin, TX 78759, USA
jim.lewis@ni.com

Abstract: The use of high-level languages for Field-Programmable Gate Array (FPGA) system modeling and rapid prototyping is the subject of considerable research, usually with the focus on speed of development or advances in synthesis techniques. Here we present a case study in which a high-level graphical programming language is used to investigate design tradeoffs involved in the FPGA implementation of a floating-point adder. Although this type of implementation is best realized at a Hardware Description Language (HDL) or lower level, we find that it is possible to rapidly develop a rough map of the algorithm design space for a given set of requirements, thus ensuring that time spent in low-level optimization is utilized efficiently.

Introduction: In this paper, we will use the LabVIEW graphical dataflow programming language (G) along with the add-on FPGA module [1] from National Instruments to perform our high-level prototyping. The LabVIEW environment [2] provides libraries and add-on modules that facilitate rapid development of test, measurement, and control applications. This package allows users with little or no digital design experience to implement simple algorithms on an FPGA as part of a real-time control system or as a reconfigurable data acquisition hardware solution. The floating-point design application in this paper lies outside the scope of the product's intended use, but leads to some interesting insights.

The paper is organized as follows: First we give a brief introduction to some of the LabVIEW graphical programming constructs used to represent hardware functionality; then we discuss the issues associated with performing floating-point arithmetic on an FPGA as well as our motivation for studying this topic; finally, we present the case study which results in a better understanding of the costs and tradeoffs associated with various features available in an IEEE-754 compliant floating-point adder. The areas we will explore include barrel shifter optimizations and cost analysis of implementing features such as rounding and exception handling.

LabVIEW FPGA Module Overview: The LabVIEW FPGA Module, an add-on to the LabVIEW graphical programming environment, allows the user to develop applications using a subset of LabVIEW functionality and target them to various devices in the National Instruments Reconfigurable I/O (RIO) hardware family. The application, also known as a virtual instrument (VI) can be run on the PC in an emulation mode in order to verify correct functionality before compiling for FPGA execution. It is also possible to create a supervisory control host application to run under Windows and pass data or commands to and from the FPGA hardware. Figure 1 illustrates the development flow for this platform. The ability to continuously emulate on the host during development allows you to efficiently debug logic errors by using a test bench in LabVIEW to verify each component as it is created.

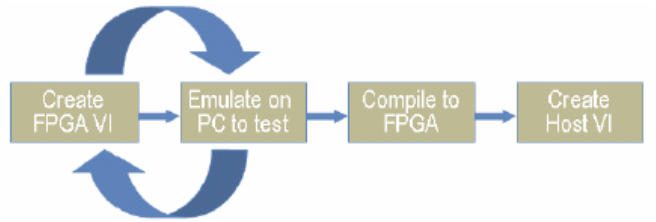

Figure 1: Application development flow for systems using the LabVIEW FPGA Module [1].

Figure 2 illustrates a simple counter application developed using the LabVIEW FPGA Module. This code implements a period measurement by measuring the number of clock cycles between rising edges detected on a digital input line. The resolution of the measurement is determined by the FPGA clock rate (with a default rate of 40 MHz). The enclosing structure on the diagram is referred to as a Single-Cycle Timed Loop (SCTL), and is the means by which we can control the placement of registers in the FPGA. All code within the SCTL must execute within a single clock cycle when targeted to an FPGA device. The terminations of wires at the right-hand boundary of the SCTL represent registered data on the FPGA, and pass the value from the right-hand boundary to the left-hand boundary during each iteration of the loop. Green wires represent 1-bit Boolean data, while blue wires

represent 8, 16, or 32-bit signed or unsigned integer data.

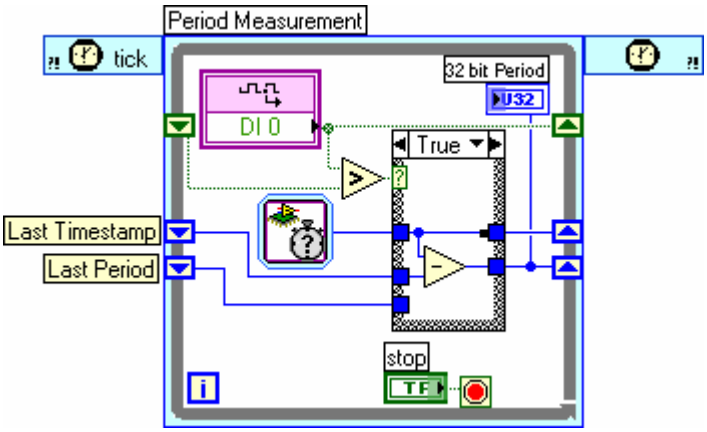

Figure 2: LabVIEW FPGA code for simple period measurement. [1]

Floating-Point Arithmetic Overview: The use of floating-point arithmetic in FPGA implementations is generally avoided due to its high cost in terms of logic and latency. There are also some potential benefits, however, including the following:

- Most algorithms are developed in floating point and require a time-consuming conversion process to fixed point before implementation in hardware.
- Some algorithms may have an inherently large dynamic range within a data path, making it difficult or impossible to implement in fixed point.
- The FPGA can be used to provide a dedicated floating-point unit to an integrated fixed-point processor.

The IEEE-754 single precision floating-point format is illustrated in Figure 3. The format consists of a 23-bit fractional field *f*, an 8-bit biased exponential, and a sign bit. Floating-point numbers are normalized such that the most significant bit (msb) of the mantissa is a 1. This bit, since known, is not stored as part of the fractional field.

In order to provide for efficient implementation of certain floating-point functions, as well as for programming convenience, we chose a modified format illustrated in Figure 4. Here the implied msb is appended to the fractional field along with eight extra bits of precision. This allows the convenience of working with a 32-bit integer and provides some extra precision that can be useful in design schemes where we avoid rounding during intermediate operations in a floating-point algorithm. Three optional status bits provide information on characteristics of the number, indicating whether it is nonzero, a nonnumber, or infinite. This format is implemented in LabVIEW as a strict type definition, allowing for easy modification based on which floating-point features are supported.

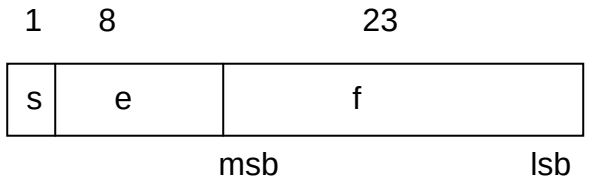

Figure 3: IEEE Single Format [2].

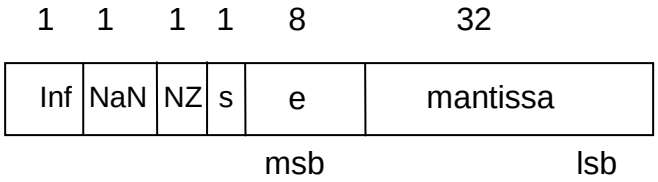

Figure 4: Sample FPGA-specific format.

In this paper, we will focus on floating-point addition since it presents some interesting implementation challenges. The layout of most floating-point addition implementations [4,5] is similar to that in Figure 5. Two of the most expensive aspects of the addition operation are the shifting required to align the mantissas before adding them, and the normalization of the resulting number. We will use the LabVIEW FPGA Module to

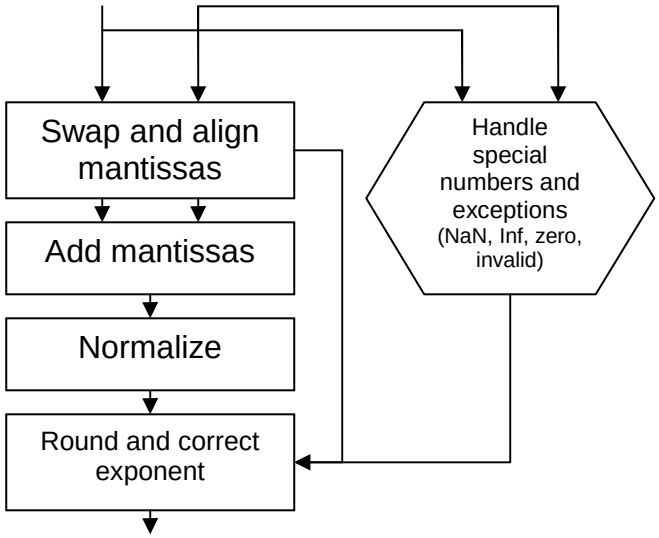

Figure 5: Basic floating-point addition steps.

[1] This is an example that ships with the LabVIEW FPGA module.

investigate the use of several different shifting techniques in order to determine which best fits our application requirements.

In most special-purpose applications, it does not make sense to provide a fully-compliant floating-point implementation. The designer may choose to implement only the floating-point features required to ensure correctness for a particular algorithm. In the second part of the paper, we will implement some of these features in order to provide an estimate of the cost (in terms of FPGA area usage) of compliance with various IEEE-754 requirements.

Barrel Shifter Optimizations: The design of a barrel shifter for FPGA implementation presents a challenge in that the optimal method depends on a number of factors, including the application requirements for area versus execution speed, available hardware resources, the range of allowable shifts, and width of the data to be shifted. For our case study of floating-point addition, we will investigate the design space for a shifter that minimizes FPGA area usage subject to the following specifications:

- When integrated into a floating-point adder design, the mantissa swap, alignment and addition must execute within one cycle of a 40 MHz clock.
- The shifted data is 32 bits wide.
- The shifter performs a logical right shift only.
- The right shift distance is specified by an 8-bit unsigned number. All shift distances greater than 31 produce an all-zero result.

We implemented the shifter using four different techniques:

- Fully parallel (31 shifts and a multiplexer)
- Hybrid (7 shifts with 3 multiplexers)[2]
- Logarithmic (5 shifts with 6 multiplexers) [6,7]
- Multipliers (using 2 Virtex-II MULT18x18 blocks)

Figures 6 through 8 on the following page illustrate the first three methods, as programmed using LabVIEW FPGA. The code in these particular implementations maps quite well to VHDL, so the synthesis results provide a reasonable comparison of the four techniques. The graphical language format allows for quick prototyping and self-documentation of each algorithm.

[2] Design based on a Xilinx VHDL synthesis template for a 16-bit barrel shifter.

Table 1 summarizes the results for each method. The parallel shifter is clearly the most expensive in terms of area, due to both its parallelism and the size of the 32x32 multiplexer. Although it did meet the 40 MHz timing constraint when implemented in a floating-point adder, its timing performance is significantly worse than the hybrid and logarithmic implementations.

The hybrid and logarithmic shifters exhibited roughly equivalent performance in terms of area and timing, with the logarithmic showing a slight advantage in area. To verify the results, we rewrote these two shift algorithms in VHDL and incorporated them into a LabVIEW test bench using the HDL Interface feature. Although the results were reversed, they were again quite similar. If further low-level optimization is required, both implementations would merit attention.

Using the dedicated 18x18 multipliers on the Xilinx Virtex-II FPGA resulted in the smallest area for the shifter, as expected, but suffered from poor timing performance due to the logic required to create the appropriate multiplicands, perform the two 16-bit multiplications and combine the results into a 32-bit word. This might be a viable choice for a pipelined design, but here we are trying to minimize latency, so it is not appropriate for our application.

Method	Area of shifter (slices)	Area of adder (slices)	Timing met?
parallel	200	430	yes
hybrid	79	323	yes
(VHDL)	67	300	yes
logarithmic	72	301	yes
(VHDL)	79	313	yes
multiplier	36	267	no

Table 1: Compilation results for logical shifters, with equivalent VHDL results for the hybrid and logarithmic implementations.[3]

Floating-Point Adder Design: We have examined the task of optimizing one component of a floating-point adder, but still face many design decisions before we can decide on an effective strategy to meet our application requirements, which are simply to provide a minimum area implementation subject to a maximum latency constraint of two clock cycles at 40 MHz. Additionally, we will have to meet some accuracy metric specified by the designer of the floating-point algorithm. With unlimited hardware resources, we

[3] All benchmarks are approximate results returned by Xilinx tools set to optimize for area and meet a 40 MHz timing constraint on a Virtex-II part (xc2v1000-4-fg456).

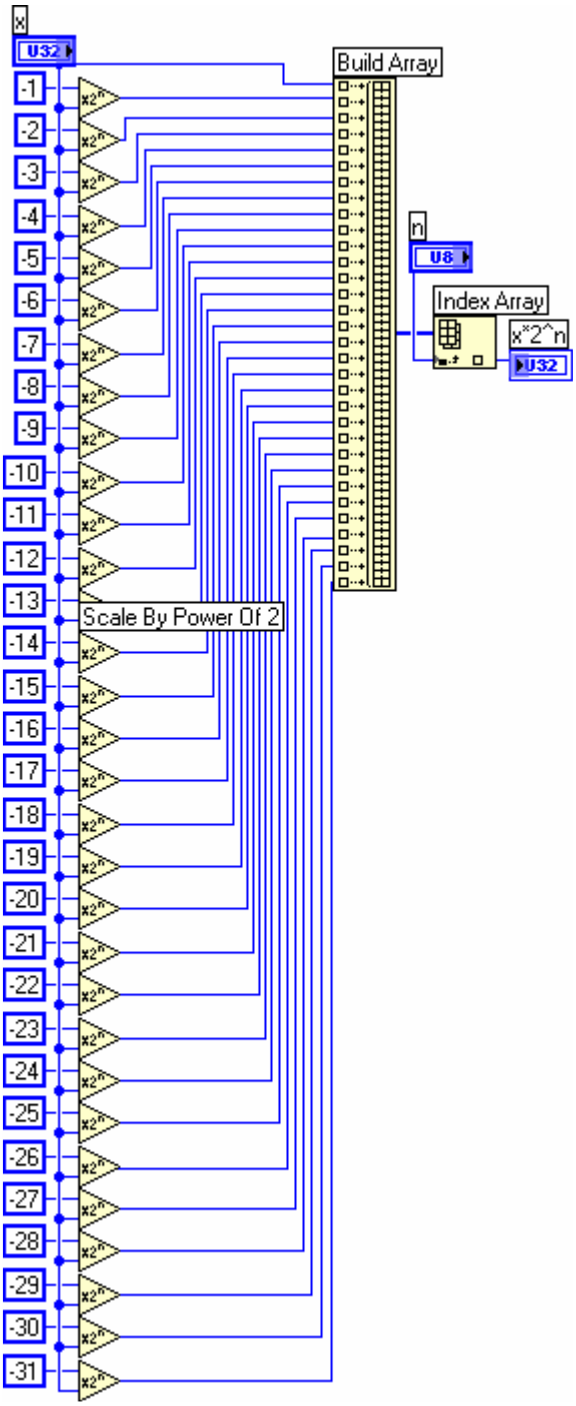

Figure 6: Parallel shift.

Note: The requirement to provide an all-zero output for out-of-range values of the shift *n* is satisfied by the behavior of the Index Array primitive.

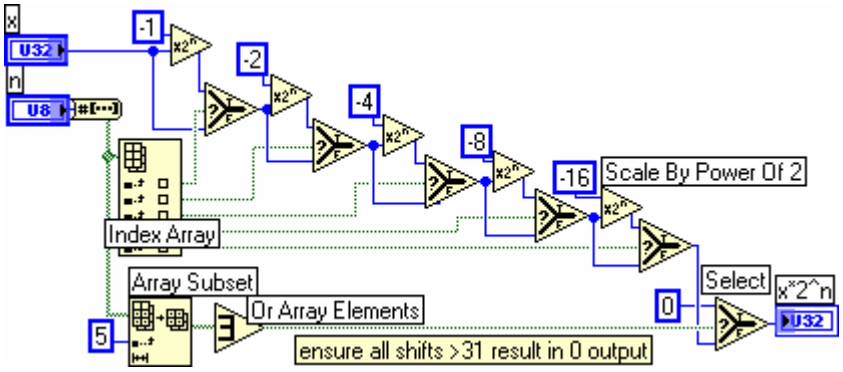

Figure 7: Logarithmic shift.

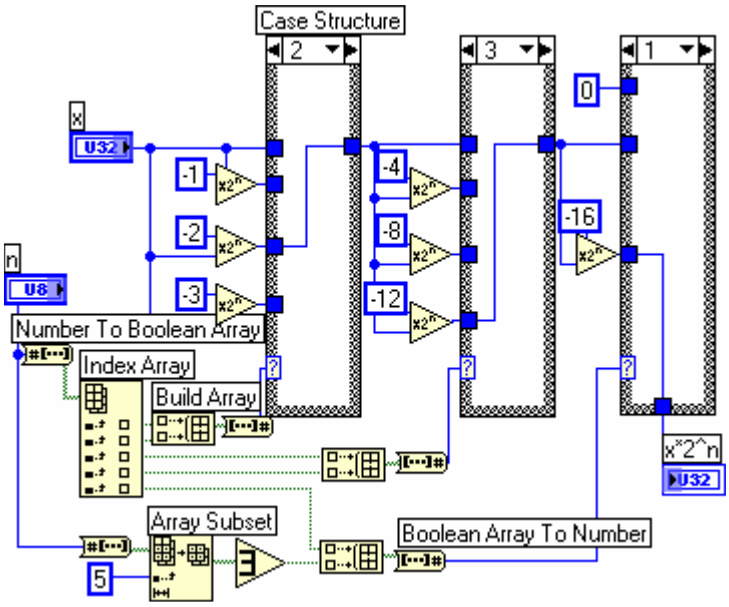

Figure 8: Hybrid shift, showing execution flow for a right shift of 30 (2+12+16).

would simply implement fully-compliant IEEE-754 floating-point operations at the same precision as the original well-tested algorithm and move on to the next project. In practice, we will probably need to move to a lower precision and abandon features such as gradual underflow or even rounding. Each deviation from the original IEEE-754 compliant algorithm increases the testing burden required to prove that the application requirements have been met.

In order to evaluate the tradeoffs involved, we need to have an idea of the relative cost of each feature. Ideally, we would have at our disposal a dense plot of some cost function involving hardware resources, accuracy, and development effort and would simply pick the combination that minimizes the overall design cost. Here we will illustrate an example of evaluating the resource cost of a few floating-point features using the LabVIEW FPGA Module for rapid prototyping. Its functional simulation capability allows us to evaluate the accuracy of each implementation by creating a test bench to run fast bit-true simulations and compare results against the reference algorithm.

In the previous section, we analyzed the impact of various shift optimizations on our floating-point adder design; now we will further characterize the design options by implementing rounding and exception handling in order to get an estimate of the incremental resource cost associated with each feature.

Figure 9 illustrates the top-level LabVIEW code for a floating-point adder that is compliant with all IEEE-754 requirements with the exception of support for denormalized numbers (gradual underflow). In addition, we imposed a requirement that its storage format and handling of NaNs exactly matches that of LabVIEW single precision arithmetic executing on a Windows platform. This incurs some extra hardware cost, but can simplify the verification process. Each of the loop structures on the diagram represents logic that executes in one clock cycle on an FPGA device.

Table 2 shows a summary of the results of these implementations. The results in terms of area are roughly comparable to those found in commercially available floating-point IP cores. Our prototype implementations are not pipelined, however, and have very limited throughput capabilities compared to commercial IP cores (which typically optimize throughput at the expense of increased latency). LabVIEW FPGA currently has limited support for pipelining, although it is possible. Figure 10 shows a first step one might take in pipelining the design. Here the four major tasks are parallelized, using shift registers on the boundary of the timed loop structure to represent pipeline registers in the design. This increases the throughput by a factor of about 2.5 (to approximately 52 MFLOPs). Further pipelining would require decomposing the blocks into smaller units and repeating the registering process.

Implementation	Area (slices)
Basic	301
+Rounding	343
+NaN, Inf, & Exceptions	410
Basic + rounding with two pipeline stages	373

Table 2: Summary of floating-point add implementations.

Finally, we verified each implementation by comparing its bit-true simulation results to those obtained using LabVIEW's single precision add function running under Windows. Figure 11 is a simple test bench that could be used to verify a fully-compliant adder. The test bench becomes more complex as we deviate from the standard, since our expected results in hardware are no longer exactly the same as those expected from a standard single precision adder.

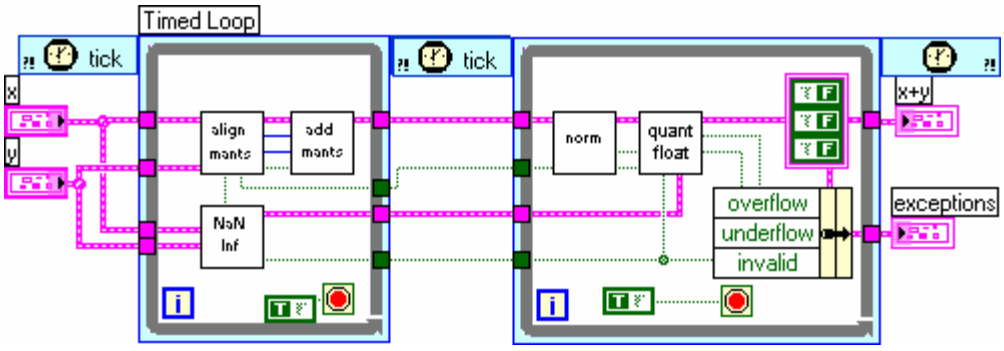

Figure 9: Floating-point adder with support for rounding, NaN, Inf, and exceptions.

Summary: We have suggested an approach to hardware design that utilizes an intuitive rapid prototyping environment (originally intended mainly for system and algorithm developers with little or no hardware experience) to provide insight into and characterization of various design tradeoffs. Our case study focused on a low-level arithmetic application, where the hardware designer could benefit from rapid exploration of design tradeoffs before choosing an approach for further optimization. We found that our high-level approach allows substantial insight into implementation decisions and can provide results comparable to those available with HDL-level coding in many cases.

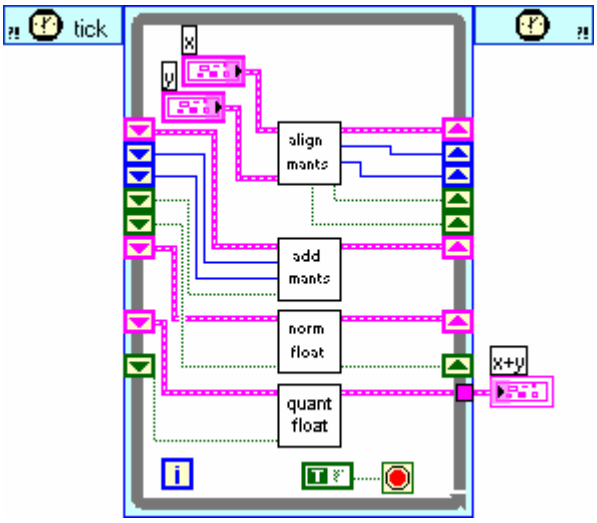

Figure 10: Pipelined implementation of basic adder with rounding.

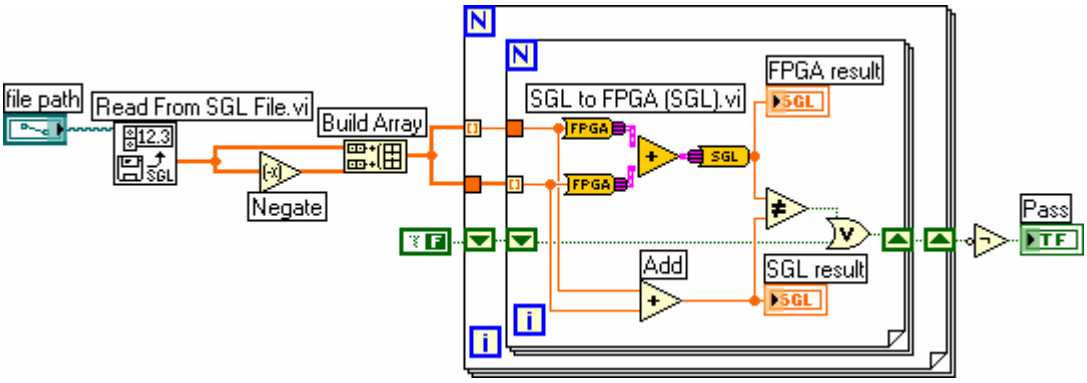

Figure 11: Simple pass/fail test bench that reads a test vector from disk and compares results of all positive and negative permutations with LabVIEW's add

References:

[1] National Instruments, LabVIEW FPGA Module data sheet, available from www.ni.com/labview.

[2] National Instruments, LabVIEW User Manual, April 2003, available from www.ni.com/support.

[3] ANSI/IEEE Std 754-1985, "IEEE Standard for Binary Floating-Point Arithmetic".

[4] A. Guyot, Arithmetic notes and exercises, available from http://tima-cmp.imag.fr/~guyot/Cours.

[5] N. Sharazi, A. Walters and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines", IEEE Symposium on FPGAs for Custom Computing Machines, Napa, California, Apr 1995.

[6] I.O.Flores and M. Jimenez, "Scalable Pipeline Insertion in Floating Point Units for FPGA Synthesis", Computer Research Conference 2003, University of Puerto Rico, Mayaguez Campus.

[7] Heo, S., "A low-power 32-bit data path design: Master's thesis", Massachusetts Institute of Technology, August 2000.