An Elementary Transcendental Function Core Library for Reconfigurable Computing

Robin Bruce, Malachy Devlin & Stephen Marshall

***Abstract*— FPGAs have established performance advantages over other processing technologies. Difficulties in achieving high design productivity counterbalance these performance advantages. High-level languages (HLLs) targeting FPGAs together with low-level core libraries have the potential to overcome these productivity challenges. It is desirable to possess an industry standard for the integration of core libraries into FPGA HLLs. The OpenFPGA CORELIB group is working towards this end. The authors present an implementation of a core library: A library of floating-point elementary transcendental functions targeted at DIME-C, an FPGA HLL, and Xilinx Virtex-4 FPGAs. The paper contrasts three methods of creating pipelined mathematical cores: Using DIME-C creation, using VHDL and using System Generator. Implementation results, comparisons with software and general conclusions about elementary functions on FPGAs are given.**

***Index Terms*— Field-programmable gate arrays, Floating-point arithmetic, Reconfigurable architectures, Elementary function approximation**

I. Introduction

FPGA-based reconfigurable computers have widely recognised performance advantages over microprocessor-based systems. Best recognised are their capabilities with regard to bit-level manipulations and integer arithmetic. High memory bandwidths and close coupling to input and output allow FPGA-based systems to offer up to orders of magnitude speed-up in certain application domains over traditional Von Neumann or Harvard stored-program architectures. The logic densities of FPGAs are increasing at an exponential rate and over recent years they have been used to implement floating-point functionality [1],[2]. Exponential increases in logic density have led to crises in design productivity in reconfigurable computing. Designing for early FPGAs essentially consisted in hand-placing logic resources. Increased logic densities led to the trend of designers switching to electronic design automation (EDA) design tools. These tools compiled and synthesized hardware descriptions languages (HDLs) to create a physical design in terms of the FPGA's resources. VHDL and Verilog are the best-known examples of such languages. As the logic densities of FPGAs have increased yet further, even the HDL/EDA approach has become very demanding on users. There is now scope for tools that allow designers to focus on the structure of their algorithm, tools that abstract away the production of HDL code that describes circuit behaviour. Several tools now exist that allow designers to program FPGAs using high-level language syntaxes such as C, C++ and FORTRAN [3]. Graphical or schematic-based programming environments also exist to develop algorithms for implementation on FPGAs. In order to obtain both the productivity benefits of high-level languages and the high-performance and low resource use of HDL techniques, core libraries are required. Core libraries permit the integration of functional units into high-level languages in a manner that is simple for the user. The OpenFPGA CORELIB effort is looking to create a standard for core libraries suitable for use in a range of high-level FPGA languages.

Research has shown FPGAs to be suitable for the implementation of floating-point elementary functions [5-14]. This paper looks at the development of a math library for Nallatech's high-level tool DIME-C. This library will conform to the CORELIB standard as and when it solidifies.

II. High-Level FPGA Languages

Using high-level languages to program FPGAs has two distinct motivations. One is to allow people with little or no experience of HDLs to transition to FPGA programming. The other is to make existing hardware engineers more productive and cost-efficient

There is now an active market for high-level languages (HLLs) that target FPGAs. We shall restrict our discussion to those that fit roughly into the category of *C-to-VHDL* compilers. C is the high-level syntax for the majority of FPGA HLL tools as C is one of the few languages that is familiar to both software and hardware engineers alike. The following tools can be

Manuscript received May 16th, 2007. This work was sponsored by Nallatech Ltd. and was carried out in conjunction with the Institute of Communications and Signal Processing at the University of Strathclyde. Additional funding was provided by the UK Engineering and Physical Science Research Council in conjunction with the Institute for System-Level Integration.

Robin Bruce is with the Institute for System-Level Integration, Alba Centre, Alba Campus, Livingston, EH54 7EG, UK, (phone: +44(0)117-377-7144; fax: +44(0)117-904-0099; e-mail: robin.bruce@sli-institute.ac.uk).

Dr Malachy Devlin is with Nallatech Ltd., Boolean House, 1 Napier Park, Glasgow, UK, G68 0BH (e-mail: m.devlin@nallatech.com).

Professor Stephen Marshall is with the University of Strathclyde, 204 George Street, Glasgow, UK G1 1XW (e-mail: s.marshall@eee.strath.ac.uk).

considered C-to-VHDL compilers: Nallatech's DIME-C, Impulse's Impulse-C [4], Mitrionics' Mitrion-C, SRC's Carte, Mentor Graphics' Catapult-C, Celoxica's Handel-C and Los Alamos' Trident compiler. ANSI standards, produced in 1989 and 1999 define the C programming language. The C-to-VHDL languages that target FPGAs do not adhere to any standards. They all use a version of the C syntax that differs from the ANSI standard. Some languages, such as DIME-C, are faithful to a subset of the ANSI C standard. Others, such as Impulse-C and Handel-C are supersets of ANSI C, containing proprietary additions to the standard. Some syntaxes differ significantly from C. Mitrion C for example uses a custom syntax that is developed with FPGA-style parallelism in mind.

All FPGA HLL tools require users to structure their algorithm in a manner that will result in optimal hardware structures. Designers must prepare the algorithm for a two-level compilation process. Firstly, the FPGA HLL compiler will create a behavioural description of the algorithm, specifying in exact detail how logical operations will occur in relation to clock cycles. This description will typically exist in some form of HDL. This behavioural description will then be passed to a second compilation process where abstract logical operations are assigned to FPGA logic resources, routing and pin mappings. This two-step compilation process is usually fully automatic from the perspective of the FPGA HLL designer. This two-step compilation process is analogous to the software case where a high-level language is first compiled to assembler and then to a binary executable. In order to obtain the best performance, designers must pipeline and parallelise as much of their algorithm as possible. The limiting constraints are the nature of the algorithm, the resources available on the FPGA and the capabilities of tool and designer combined. FPGA Pipelines have no theoretical depth limit and a single design may have multiple parallel pipelines, with depths of hundreds or even thousands of cycles. This, combined with other parallelisation techniques such as logic replication and concurrent scheduling of independent operations, gives FPGAs their vast performance potential. The approach taken to generate parallel and pipelined structures is a distinguishing factor between FPGA HLLs. Some languages require users to explicitly tag sections of code to pipeline or schedule in parallel. Others, such as Trident or DIME-C, have the philosophy that the tool should attempt to parallelise and pipeline code wherever it sees the opportunity. DIME-C also has certain *pragmas* or *compiler directives* that allow users to specify large-scale logic duplication without adding any non-ANSI-compliant grammar to its syntax.

An important feature that distinguishes FPGA HLLs from lower-level HDLs is their capability for handling arithmetic that combines different datatypes. Users can mix integer and floating-point computation without having to consider the additional logical structures that are required. These include conversions of operators and results. Not all FPGA HLLs have the same capabilities in this regard. In DIME-C, Trident and SRC's Carte, support for floating-point computation is part of the language itself. Other languages only support floating point via libraries and this naturally affects how users mix datatypes.

Some FPGA high-level languages are *cycle accurate.* They require users to specify the clock cycles on which operations are scheduled. This means that users have a more powerful tool at their disposal but have less abstraction from the design process. Users of cycle accurate languages must still describe the circuit behaviour that their algorithm requires.

FPGA HLL users require a means of functionally testing their algorithms as they develop them in the HLL syntax. As the two-step hardware compilation process can take anything from minutes to hours, and because it can be difficult to debug an algorithm in hardware, debug is optimally done in software running on a microprocessor. Most tools offer a proprietary debug environment geared to accepting their variant of the C syntax. DIME-C code, being a subset of ANSI C standard code, can be compiled using a standard C compiler such as the GNU C Compiler (gcc).

III. Core Libraries and the OpenFPGA CORELIB effort

A. Core Libraries

Many of the FPGA HLL tools that are available offer the capability to integrate libraries of low-level cores into the language and instantiate them as function calls. SRC's Carte environment is one such tool. DIME-C also has this capability. The low-level cores are blocks of logic that are designed either using DIME-C itself or using a more traditional HDL design process. The cores are described in a library descriptor file that gives DIME-C all the information about the file that is necessary for implementing the logic block as part of a greater logical structure. The entire mechanism is designed in such a way as to make the instantiation of library cores appear to the user as indistinguishable from a C function call in software. The user adds in the library descriptor file to a project in the same manner that they would add in a statically linked library file in ANSI C. The function calling prototype is found in a *header* file, as in ANSI C.

Core Libraries allow users to enjoy the design productivity and abstraction from complexity that high-level languages offer while leveraging the high performance and low resource consumption that traditional development techniques offer. The cost for these clear advantages occurs in the design and verification of these cores themselves

B. The OpenFPGA CORELIB Effort

FPGA HLL tool developers are at present working together to form a standard for library cores that would allow them to be shared between FPGA HLL tools. This standard effort is being carried out as part of the CORELIB workgroup of the OpenFPGA organization [20]. This level of standardization

would enable the reconfigurable computing community to collaborate on developing libraries. These libraries would be suitable for use with a range of software tools and hardware platforms, thereby benefiting the entire community.

The specific goals of this effort include the following:

- To develop a standard for describing the interface and operating characteristics for cores that facilitates the integration of these cores within high-level programming language compilers and other FPGA design tools.
- To develop a standard for libraries of cores that facilitates the building, distribution and integration of cores across tools and hardware platforms.
- To encourage the use of these standards by compiler developers, core developers, universities, and other tool vendors.
- To create a synergistic environment where programmers, and core developers can create the most effective implementations of applications on FPGA-based systems.
- To promote the creation of general purpose and application specific core libraries for use across the available tools and platforms.

There are a range of organizations that exist presently with similar goals to the CORELIB group. These include the SPIRIT consortium, opencores.org, OCP-IP and the Virtual Socket Interface Alliance. Although all of these organizations have similar goals to the CORELIB group, none of them provide an answer to all of CORELIB's specific requirements, so there is a need for a new group.

In order to integrate IP cores into a high-level language, the cores must be accompanied by a descriptor file that can be read by all of the relevant high-level tools. This descriptor file describes the clocking, timing, control and data properties of IP cores. The approach CORELIB is pursuing is to create an extensible markup language (XML) specification for representing these details. This is to be based on the Spirit Consortium IP-XACT format. The current status of the CORELIB effort is covered in greater detail in [21],

C. *Off-Chip Interface Cores*

Another potential use for core library cores is to simplify the interactions with the hardware resources that reside *off-chip*. These resources could include, but are not limited to: communication network connections, SRAM banks, DRAM banks, still and video cameras, analog-to-digital & digital-to-analog converters, LVDS links and multi-gigabit transceivers. Typically one would need to develop *firmware* that interfaced to the off-chip resource. A user wishing to use that resource would then have to understand the user-facing interface to the firmware in order to leverage the resource. Instead the off-chip resource cores can be built to a standard such as CORELIB and integrated easily into high-level languages. This would permit the rapid development of complex processing systems using high-level languages. The I/O processing capabilities of FPGAs are arguably their greatest asset and this would permit users to leverage them without having to deal with the complex interactions that this usually entails. VITA 57 [22], the FPGA Mezzanine Card (FMC) standard is currently under development. VITA 57.1 defines a standard for small FPGA I/O modules that can be added into larger reconfigurable computing systems. The VITA 57.3 standard defines firmware drivers with standard interfaces that will facilitate the transport of I/O data. With the combination of FMC modules, CORELIB compatible firmware drivers, CORELIB processing cores and FPGA high-level languages one can envisage the rapid development of high-performance I/O processing systems. These systems would have an incredibly tight, low latency coupling to their input/output devices.

IV. Implementing a Low –Level Math Library For Use in High-Level Tools

A. *Floating-Point on FPGAs and Other Architectures*

FPGAs can implement a high number of GFLOPS (giga floating-point operations per second). They are also often more easily capable of implementing a high proportion of peak FLOPS in typical designs. This is due to their ability to directly implement dataflow architectures. FPGAs have generally been regarded as more viable as single-precision accelerators than they have been as double-precision accelerators.

FPGAs excel where they are used to implement operations for which no dedicated hardware units exist on competing architectures. In these cases microprocessors must carry out this operation via hardware operations that do exist. There is a class of floating-point operations that do not exist in hardware in modern devices. These are the *elementary transcendental* functions, e.g. *exp, log, cos, sin* etc. In programmable architectures these operations are carried out largely via the use of polynomial approximations, sometimes with the aid of look-up tables to reduce the degree of polynomial required. Unlike basic arithmetic operations, these functions cannot be pipelined on microprocessors, so the throughput for these functions can be relatively low when compared to FPGAs.

FPGAs on the other hand allow for the implementation of fully-pipelined custom units. Using a mix of the LUT, register, block RAM and fixed-point multiplier structures present on the FPGA, floating-point elementary function cores can be created. Often these functions can be implemented without actually requiring the instantiation of floating-point functional units. The architectures implemented in this project have thus far been interpolated table lookups, usually some derivatives of *Tang's method* [24]. Muller [8] provides a good overview of the issues involved in approximating elementary functions in floating-point, and this work is developed well for FPGAs by Detrey and de Dinechin [9-12].

B. Techniques of Core Creation

The table below shows the math functions that have been implemented thus far for the DIME-C compiler and Xilinx FPGAs:

Function	Description
expf	Natural exponential
logf	Natural logarithm
sinf	Sine
cosf	Cosine
tanf	Tangent
fabsf	Returns magnitude of input
frexpf	Splits input into fractional and exponent output
ldexpf	Creates output from fractional and exponent inputs
modff	Splits input into integer and fractional outputs
powf	Returns input x raised to the power of input y, i.e. x^y
sqrtf	Square Root
rand	Generates a random number in range [0,32767]

TABLE 1: FUNCTIONS IMPLEMENTED THUS FAR IN THE DIME-C MATH LIBRARY

More details on this math library than are found here can be found from [5] and [7].

Throughout the course of the ongoing project, a number of different approaches to implementing the cores have been taken. Initially the cores were implemented directly in a hardware description language (HDL), namely VHDL. Some of the later cores were implemented using DIME-C then exported to become library cores. Finally, system generator was investigated as a means of generating the cores. System Generator [23], a tool that leverages Matlab, Simulink and Xilinx tools, is a high-level graphical tool intended for DSP. Each of these three approaches had their strengths and weaknesses, which are detailed below. VHDL can be considered to be equivalent to Verilog, and DIME-C and System Generator could be expected to be broadly similar to other tools that fulfill the same roles.

1) VHDL

VHDL Core Creation Advantages:

- VHDL is a long-established industry standard with a strong development ecosystem and a clear future.
- Assuming an expert user, VHDL cores can generally offer the greatest performance, latency and area characteristics possible.
- Library Developers can design cores to use non-standard data types within the intermediate data calculations. For example, although a core may have floating-point inputs and outputs, all its intermediary calculations may be carried out using variable bitlength fixed-point arithmetic.
- Cores are *future-proof* in the sense that any technology for which VHDL synthesis exists can use them, perhaps even without any modification. This means that investment of time and money into developing VHDL cores is less risky and dependent on the success of a proprietary language.
- Library developers can acquire cores from third parties. End users could even source entire libraries from third parties. Using traditional HDLs as the base for core libraries allows for their use in different reconfigurable computing compilers.

VHDL Core Creation Disadvantages:

- Library core development is time consuming and error prone. The majority of library cores developed were fully pipelined and much time was lost dealing with mismatched signal delays.
- Requires electronic design skills not possessed by the majority of software engineers.
- VHDL source for complex pipelined cores can be difficult to interpret and modify

2) DIME-C

DIME-C Core Creation Advantages:

- Users are more productive, and the result is more reliable
- Simplest manner to create cores
- Easier to understand functionality and modify cores
- Process is more accessible in the sense that the user does not have to install a number of costly tools and get to grips with a potentially unreliable design flow linking several unrelated tools together.
- Library designers can quickly create functional cores from existing software routines.
- Generated library cores can be reused in other tools
- Generating fully-pipelined cores is not more difficult than generating non-pipelined cores

DIME-C Core Creation Disadvantages:

- Increased resource consumption and lower performance (i.e. higher latency and lower clock rate) than VHDL.
- Library Maintenance is dependent on DIME-C. Although application developers can use the library cores in standard HDL projects, they cannot easily modify them. The HDL produced by DIME-C is not easily readable by humans.
- Library developers can only use the Boolean type and the datatypes present in ANSI C, that is to say 8,

16, 32 and 64-bit integers and single and double-precision floating-point numbers.
- Less control over the hardware resources used and the temporal scheduling of operations.

3) System Generator

System Generator Core Creation Advantages:

- High Productivity (lower than DIME-C, but far higher than VHDL)
- Relatively low resource use (lower than DIME-C, higher than expert VHDL)
- Relatively high performance (higher than DIME-C, lower than expert VHDL)
- Generating a pipelined structure is not more difficult than generating a non-pipelined structure
- System generator suits the implementation of the *predicated dataflow* algorithms used for elementary function approximation

System Generator Core Creation Disadvantages:

- Cost of Tools. Users require to gain access to Simulink, Matlab, System Generator Toolbox and ISE,
- Maintenance of the tool chain, integrating Matlab, Simulink, ISE and System Generator and their various updates and patches, is left to the user.
- Some of the datatype assumptions and the lack of certain functional blocks mean that System Generator is not a perfect match for the implementation of elementary transcendental functions.

V. FPGA-Implemented Transcendental Functions

A. Single-Precision vs. Double Precision Implementations

The implementation of each transcendental function must be considered in isolation. The minimum resources and minimum latency possible for a fully-pipelined core will vary depending on the mathematical properties of the function being implemented. For example, the following basic mathematical properties, together with those of the floating-point representation format, simplify the implementation of the *exp, log, and sin* cores respectively:

$$\begin{aligned} & x = 1.f_m \ldots f_0 \times 2^{ept} \\ & x_{fix} = fix(x) = i_p \ldots i_0.f_q \ldots f_0 \\ & \exp(x) = \exp(x_{fix}) = \\ & \exp(i_n \ldots i_0) \times \exp(0.f_m \ldots f_0) \end{aligned} \quad (1)$$

f here is the fractional part of the floating-point input x, preceded by its leading 1. m corresponds to the precision of the input floating-point number. *ept,* an integer, is the unbiased exponent of the floating-point input. Following a floating-to-fixed-point conversion of the input i here is the integer part of the input number, and f is its fractional part, with p and q being their respective bitwidths. In the implementation of the above function one exponential term is a table lookup and the other is obtained from a polynomial approximation.

$$\begin{aligned} & x = 1.f_m \ldots f_0 \times 2^{p} \\ & \log(x) = \\ & p \times \log(2) + \log(1.f_m \ldots f_0) \end{aligned} \quad (2)$$

In the physical implementation on FPGA $p \times \log(2)$ is a table lookup and the other term is a polynomial approximation.

$$\begin{aligned} & x = x = 1.f_m \ldots f_0 \times 2^{ept} \\ & x_{fix} = fix(x) = i_p \ldots i_0.f_q \ldots f_0 = (a+b) \\ & \sin(x) \\ & = \sin(a+b), \\ & = \sin(a)\cos(b) \\ & + \cos(a)\sin(b) \end{aligned} \quad (3)$$

x and x_{fix} are as defined previously. x_{fix} is split into two parts: input portion a that indexes a look-up table and input portion b serves as input to a polynomial approximation.

For the implementation of single-precision functions, where inputs and results have 24 bits of fractional precision, it is usually sufficient to implement fixed-point arithmetic operations based around lower degree polynomial approximations (degree 2 or 3) with table sizes rarely more than 512 or 1024 entries. There is a tradeoff between polynomial degree and table lookup size. Switching from single-precision to double-precision function approximation significantly increases resource requirements. Higher degree polynomial approximations become necessary. For a doubling in approximation precision, the number of basic arithmetic units required typically tends to more than double. Furthermore, moving to higher-precision approximations will necessitate the use of higher-precision floating-point units, so as well as having more units, each one will consume a far greater amount of resource. As an example, the implementations of a single-precision and a double-precision logarithm core are compared:

a) Single-Precision Logarithm vs. Double-Precision Logarithm

The single precision logarithm used a similar, but simpler algorithm to the double-precision example. Only the double-precision algorithm is explained here in detail. In this example, all multiplications are carried out in slice logic and not in dedicated multipliers. This is to simplify comparison between the two architectures. Note: when dealing with relatively large precision arithmetic, fixed-point additions consume nearly negligible resources when compared to floating-point

additions. On Virtex-4 a 64-bit fixed-point addition consumes 32 slices, whereas a 64-bit floating-point addition consumes 701 slices. For this reason, only floating-point additions are considered as contributing significantly to the resource total of the algorithm below:

$$\begin{aligned} & x \in [2^{-1022}, 2^{1023}] \\ & x = 1.f_m \ldots f_0 \times 2^{ept} \\ & frac = 1.f_m \ldots f_0 \\ & \log(x) = (\log(2) \times ept) + \log(frac) \\ & ROM1(ept) = \log(2) \times ept \end{aligned} \quad (4)$$

ROM1, a look-up table, requires 2048 × 64 bits storage = 8 16kbit block BRAMs

In the equations below, *N* is the number of breakpoints into which the range of *frac* is subdivided. *frac* is the fractional part of the input, together with its implied leading one.

$$\begin{aligned} & c_k = 1 + \frac{k}{N}, where \left| frac - c_k \right| \le \frac{1}{2N} \\ & r = 2(frac - c_k)./(frac + c_k) \\ & r \in [0, \frac{1}{N}) \\ & \log(\frac{frac}{c_k}) = \log(\frac{1 + r/2}{1 - r/2}) \\ & \log(frac) = \log(c_k) + \log(\frac{frac}{c_k}) \\ & ROM2(k) = \log(c_k) \\ & p(r) = \log(\frac{1 + r/2}{1 - r/2}) \end{aligned} \quad (5)$$

Note: When targeting a particular precision, any increase or decrease in the look-up table size *N* should be balanced by a corresponding decrease or increase in the degree *n* of the interpolating polynomials. Equally changes in *n* should be mirrored by changes in *N*.

If polynomial approximation p(r) is of degree *n*, then the block RAM storage required for look-up table ROM2 is

$$BRAMS = \frac{N \times (floor(\frac{n}{2}) + 1)}{256} \quad (6)$$

n = 1 requires many thousands of BRAMs in order to be implemented. n = 3 seemed to be the optimum polynomial degree. Anything higher makes too little use of the abundant BRAMs and increases slice resources significantly. For double-precision 16 BRAMs are required for ROM2 and 8 for ROM1, making the resource estimate for the logarithm function as follows:

24 BRAMs
3 Double-Precision Additions (701 slices)
1 Double-Precision Division (3036 slices)
3 Double-Precision Multipliers (1238 slices)

Also required are a barrel shifter, a normalization unit, registers and logic to connect everything up meaningfully in a pipelined structure with error correction and control. This leads to an estimated slice total of 12,000 slices.

	Single	Double	Increase
Exponent Bits	8	11	**1.375**
Fractional Bits	24	53	**2.21**
Slice Consumption	1736	12000	**6.91**
BRAM Consumption	3	24	**8.00**

TABLE 2 – COMPARISON BETWEEN SINGLE AND DOUBLE PRECISION LOGARITHM

Implementing double-precision elementary functions requires significantly more resources than single precision. Part of the reason for this is the lack of *hard-wired* double-precision arithmetic units on the FPGA fabric. Even so, one would still expect a potential throughput from a high-end FPGA to be up to 1.5 GOPS for the double-precision case, assuming the units are clocked at 300MHz and 5 units are implemented on the FPGA. During development in this project a number of cores, such as the single-precision *exp* and *log*, were clocked in the 300 MHz range on Virtex-4 FPGAs, and it is expected that level of performance could be attained for the double precision cores, though this has yet to be proven in practice.

B. Faithful Rounding versus Correct Rounding

Elementary transcendental functions typically have a finite, or at the very least *countably infinite,* number of inputs for which they issue rational results. This leaves an *uncountably infinite* number of inputs for which the results are irrational or *transcendental*. This means that they cannot be explicitly expressed in any form of numerical notation. Therefore, the overwhelming majority of outputs of the functional units described here are approximations to a transcendental. The functional units described in this paper are intended to issue *faithfully rounded* results. This means that the output of the functional units is one of the two rational floating-point numbers that bound the true transcendental result. Faithfully rounded results have errors in the range [0.5,1] *ulps* (*units in the last place)*. Guaranteeing *correctly-rounded* results, where all results are the floating-point number closest to the underlying transcendental, is a challenge. It is known as the table-maker's dilemma. Muller describes it well in his book on

the subject [8]. To guarantee the 0.5 ulps accuracy of correct rounding, experiment has shown that around ~120 bits of approximation accuracy are typically necessary. This guarantees correct rounding for the worst case in double-precision floating point. For the microprocessor implementation of correct rounding, this is not a great issue. For the overwhelming majority of cases the microprocessor will be able to use a simpler algorithm. All that is required is a check for *difficult-to-round* cases, and a different algorithm to handle these cases when they arise. The additional time penalty is only paid on the occasions where a difficult-to-round case is detected. For the pipelined dataflow structure of the FPGA functional unit, all possible paths through the algorithm must be represented in hardware. Therefore, to create a fully-pipelined functional unit that guaranteed 0.5 ulp correctly rounded results, one would have to implement the worst case in hardware. This means using ~120-bit precision floating-point units, enormous look-up tables and approximation polynomials of exceptionally high degree. It would be difficult if not impossible to fit such a structure onto even the largest FPGAs available today. Correctly-rounded results guarantee monotonicity and are perhaps the only practical method of creating portable numerical algorithms. However, mitigation strategies can be envisaged for this problem. One such strategy would be the implementation of non-correctly rounded transcendental functions that guarantee preservation of monotonicity, [16] explains how one might implement such algorithms.

VI. Results

A. Implementation Results of the Math Library

All of the results shown in table 1 are taken from physical synthesis reports, except where '*' denotes logical synthesis only. The target device was a Xilinx Virtex-4 LX160 speed grade -10. The DSP48s are arithmetic units on Xilinx Virtex-4 FPGAs, used here as 18x18 multipliers. The RAMB16s are dual-ported 16 kbit RAMs present on the FPGA. The difference in the clock rate of the rand functions is determined by their internal architecture. *rand_ms_i* exactly matches the mingw32 rand() function. The iterative nature of this algorithm slows down the clock rate. To overcome this, six threads of this algorithm were implemented on a pipelined structure to create the *rand()* function, thereby increasing the clock rate.

Function	Slices	DSP48s	RAMB16s	Clock Rate (MHz)
tanf	10338	48	2	130.5
powf	3589	4	4	136.0
cosf	5389	24	1	168.9
sinf	3982	24	1	156.7
frexpf	26	0	0	191.2
ldexpf	44	0	0	134.5
expf	1405	0	1	150.1
logf	1736	0	3	133.5
rand_ms	150*	0	0	106.3
rand_ms_i	150*	0	0	90.4
rand	475	0	0	181.0

TABLE 3 – IMPLEMENTATION RESULTS OF MATH LIBRARY

Comparisons between software and hardware implementations of the sqrtf, expf, & logf functions are now made. It is difficult to get access to proprietary code that would be used on specific microprocessors. Instead code has been taken from one of the many libraries for which code is publicly available, one developed by Jesus Calvino-Fraga and released under the LGPL [25]. For the exponential and logarithm functions, care has been taken to ensure that these are actual single-precision approximations, not double-precision code that has been adapted to produce a single-precision result. This would otherwise unfairly skew the result in favour of the FPGA.

Although in the earlier stages of implementation *log* and *exp* cores clocking at around 300 MHz were developed, these were abandoned upon the discovery of precision errors. The next generation of cores was implemented focusing primarily on precision. It should be noted that it is possible for cores to run at a higher clock rate than the logic generated by the high-level compiler

There is some justification in choosing this public code, as it is portable C code. It would be reasonable to expect many of the proprietary libraries to be microprocessor specific. This would make porting between microprocessors difficult.

The FPGA routines compared here exist as portable VHDL, not specifically instantiating any resources, and only inferring those that one could reasonably expect to be present on all contemporary and future FPGAs. It is therefore considered fair in this case to compare them with portable C code.

B. Software Performance

In order to measure the software performance of these functions each of them will be measured in two ways.

1. Executed in isolation 2^{10} times, with the average execution time being recorded. (**SW 1**)
2. Executed as part of a greater function 2^{10} times, with the function's average contribution to the overall run time being measured. The greater function is to be a

function that is pipelined when implemented in DIME-C. (**SW 2**)

For our case the *greater function* will be based around the probability density function, into which each of the functions will be inserted. This function is chosen as being representative of the kind of functions present in high-performance computing applications. This means that expf SW2 will consist of an execution of the probability density function for a single set of inputs, shown below in eqn. (7), with an additional, meaningless exponential operation added in.

$$f(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2n}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad (7)$$

For the 'all_funcs' function, all of the functions under test were combined into a single "super function" that could be compared with an FPGA implementation. This simply consisted of each operation being executed in turn within a single C function call.

Additionally the full probability density function, PDF, is implemented both in software and in hardware.

C. Comparison Environment

Microprocessor & ANSI-C Compiler:

- 3.2 GHz Pentium D (dual-core) (90 nm process)
- Windows XP
- 2GB RAM
- gcc –O3

FPGA & DIME-C Compiler:

- Virtex-4 SX35-10 FPGA (90 nm process)
- All Logic Clocked at 100MHz

D. Comparison Results

The times shown below are the times taken for one execution of each function, taken as an average.

HW is the time taken for a single execution of each function on the FPGA, with fully pipelined operation.

SW 1 is the time taken for a single execution of the function in software, with the function operating in isolation.

Note that SW 2 only measures the contribution of that particular function's runtime to the total runtime of the larger function and not the time to run the entire function. Table 4 below shows that the FPGA-implemented functions have higher performance than the microprocessor. The performance of the microprocessor drops when the microprocessor is carrying out a mix of operations, as opposed to carrying out the same operation repeatedly.

The HW throughput is largely independent of the amount of logic implemented. This means that you can build up ever larger functions on the FPGA, while maintaining the same throughput. All closed-form mathematical expressions can be pipelined in this manner. The software throughput on the other hand will drop as the function builds up.

Function	HW Timing (us)	SW 1 Timing (us)	SW 2 Timing (us)	HW/SW 1 Speedup	HW/SW 2 Speedup
expf	0.01	0.213	0.215	**21.29**	**21.48**
logf	0.01	0.126	0.178	**12.60**	**17.77**
sqrtf	0.01	0.199	0.246	**19.92**	**24.61**
all_funcs	0.01	0.625	N/A	**62.50**	**N/A**
PDF	0.01	0.43	N/A	**43.00**	**N/A**

TABLE 4 – SOFTWARE / HARDWARE PERFORMANCE COMPARISONS

VII. CONCLUSIONS

The motivations for high-level languages that target FPGAs have been presented. The general properties and features of high-level languages for FPGAs have been outlined. Specific reference has been made to the features of the DIME-C language as well as to a number of other high-level tools. Core libraries for use in high-level languages have been defined, along with their advantages. The existence and progress of a standards body for core libraries, the OpenFPGA CORELIB workgroup, has been reported. Cores that interface to off-chip resources and their relevance to core libraries and high-level languages have been presented. Reference has been made to the VITA-57 standardization effort. The effort to produce a math library for DIME-C has been outlined. Three different means of producing cores, via VHDL, via DIME-C and via System Generator have been discussed. The relative merits of each design approach have also been given. The greater resource consumption of double-precision elementary functions on FPGAs versus single precision was shown. Reference has been made to the handicap FPGAs possess in not having hard-wired floating-point arithmetic units. The complications in implementing correctly-rounded elementary transcendental functions on FPGA have been explained. The implementation results of the present incarnation of the math library on Virtex-4 devices were presented. Finally, the

performance improvements of FPGA-based implementation of elementary functions over a microprocessor were shown.

REFERENCES

[1] Underwood K, Hemmert K, *Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance*, Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on (2004), pp. 219-228.

[2] 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.

[3] R. Wain, I. Bush, M. Guest, M. Deegan, I. Kozin, and C. Kitchen, *An overview of FPGAs and FPGA programming; Initial experiences at Daresbury*, Computational Science and Engineering Department, CCLRC Daresbury Laboratory, Daresbury, Warrington, Cheshire, November 2006.

[4] David Pellerin, and Scott Thibault, *Practical FPGA Programming in C*, Prentice Hall, Washington, April 2005.

[5] Bruce R, Chamberlain R, Devlin M, Marshall S, *Implementing algorithms on FPGAs using high-level languages and low-level libraries,* Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, Florida

[6] DU Lee, AA Gaffar, O Mencer, W Luk*, Optimizing Hardware Function Evaluation,* Computers, IEEE Transactions on, 2005

[7] Bruce R, Devlin M, Marshall S. *Lessons Learned Implementing the VSIPL API on Reconfigurable Computers.* Military and Aerospace Programmable Logic Devices International Conference, Washington DC, September 2006

[8] Muller, J, *Elementary Functions, Algorithms and Implementation,* 2nd Edition. Boston: Birkhauser, 2005

[9] Detrey J, de Dinechin F*, Parameterized floating-point logarithm and exponential functions for FPGAs*, Field-Programmable Technology, 2005. Proceedings.

[10] Detrey J, de Dinechin F, *High-performance hardware operators for polynomial evaluation,* International Journal of High Performance Systems Architecture, Volume 1, Number 1 / 2007, pp14 - 23

[11] Detrey J, de Dinechin F, and Pujol X, *Return of the hardware floating-point elementary function*. In 18th Symposium on Computer Arithmetic. IEEE Computer Society Press, June 2007

[12] Detrey J, de Dinechin F, *Second Order Function Approximation Using a Single Multiplication on FPGAs*, Lecture Notes in Computer Science, Volume 3203/2004, pp 221-230

[13] Nagayama S, Sasao T, Butler JT*, Programmable numerical function generators based on quadratic approximation: architecture and synthesis method,* Proceedings of the 2006 conference on Asia South Pacific design automation, pp378-383

[14] Doss, C.C. Riley, R.L., Jr. *FPGA-based implementation of a robust IEEE-754 exponential unit*, Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on, pp 229- 238

[15] Andraka R, *A Survey of CORDIC Algorithms for FPGA Based Computers*

[16] Ferguson, W.E., Jr. Brightman, T., *Accurate and monotone approximations of some transcendental functions* Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on, pp237-244

[17] Brian Holland, Mauricio Vacas, Vikas Aggarwal, Ryan DeVille, Ian Troxel, and Alan D. George. *Survey of C-based Application Mapping Tools for Reconfigurable Computing*. Military and Aerospace Programmable Logic Devices International Conference, Washington DC, September 2005

[18] Kindratenko V, *High-Performance Reconfigurable Computing Application Programming in C*

[19] Beauchamp MJ, Hauck S, Underwood KD, Hemmert SK, *Embedded Floating-Point Units in FPGAs*

[20] *OpenFPGA Working Groups,* www.openfpga.org

[21] Mike Wirthlin, Dan Poznanovic, Prasanna.Sundararajan, Alan Coppola, David Pellerin and other contributors*, OpenFPGA CoreLib Core Library Interoperability Effort,* RSSI 2007 submission

[22] VITA Standards Organization http://www.vita.com/vso-stds.html

[23] Xilinx System Generator for DSP Reference Guide, www.xilinx.com

[24] PTP Tang , *Table-driven implementation of the exponential function in IEEE floating-point arithmetic*, ACM Transactions on Mathematical Software (TOMS), 1989

[25] Jesus Calvino-Fraga, math.h implementation, http://tinyurl.com/2z6gbl