HAL
open science

An FPGA architecture for solving the Table Maker's Dilemma

Florent de Dinechin, Jean-Michel Muller, Bogdan Pasca, Alexandru Plesco

To cite this version:

Florent de Dinechin, Jean-Michel Muller, Bogdan Pasca, Alexandru Plesco. An FPGA architecture for solving the Table Maker's Dilemma. Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on, Sep 2011, Santa Monica, United States. pp.187-194, 10.1109/ASAP.2011.6043267 . ensl-00640063

HAL Id: ensl-00640063

https://ens-lyon.hal.science/ensl-00640063v1

Submitted on 14 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An FPGA architecture for solving the Table Maker's Dilemma

F. de Dinechin, J.-M. Muller, B. Pasca, and A. Plesco
CNRS, ENS Lyon, INRIA, UCBL
Laboratoire LIP, ENS Lyon, 46 allée d'Italie
69364 Lyon Cedex 07, FRANCE
FirstName.LastName@ens-lyon.fr

***Abstract*—Solving the Table Maker's Dilemma, for a given function and a given target floating-point format, requires testing the value of the function, with high precision, at a very large number of consecutive values. We give an algorithm that allows for performing such computations on a very regular architecture, and present an FPGA implementation of that algorithm.**

***Keywords*-table maker's dilemma; floating-point arithmetic; correct rounding; elementary functions; FPGA;**

I. THE TABLE MAKER'S DILEMMA

The IEEE 754-2008 standard for Floating-Point arithmetic [1] recommends (yet does not dictate) that some elementary functions should be correctly rounded. That is, given a rounding function $\circ$, (e.g., round to nearest even, or round to $\pm\infty$), when evaluating function f at the floating-point number x, the system should always return $\circ(f(x))$.

Building a fast correctly rounded library for some target floating-point (FP) format requires preliminarily solving a problem called the *table maker's dilemma*. This requires very large computations which may use environments and formats totally different from the target environment and format. In this paper, we suggest performing these computations on an FPGA.

For the sake of simplicity, in this paper, we assume that the target rounding function is the IEEE 754 standard double-precision round to nearest even. On the target environment, to compute $f(x)$ in a given format, where x is a FP number, we must first compute an approximation to $f(x)$ with a given accuracy [2], which we round to the nearest FP number in the considered target format. The problem is the following: find what the accuracy of the approximation must be to make sure that the obtained result be always equal to the "exact" $f(x)$ rounded to the nearest FP number. To solve that problem we have to locate, for each considered target floating-point format and for each considered function f, the *hardest to round* (HR) points. Defining a midpoint as the exact middle of two consecutive floating-point numbers, an HR point is an input floating-point numbers x such that $f(x)$ is very close to a midpoint, without being exactly a midpoint.

This work is partly supported by the TaMaDi project of the french *Agence Nationale de la Recherche* (ANR). The donation of a DK-DEV-4GX530N board by the Altera University Program is also gratefully acknowledged.

Assuming (after some re-normalization) that x and $f(x)$ are between 1 and 2, that the target floating-point format is a binary format of precision p, we need to find the largest possible value of m such that there exist a FP number x that satisfies:

- $f(x)$ is not exactly equal to a midpoint;
- the binary representation of $f(x)$ has the form

$$\overbrace{\underbrace{1.xxxxx\cdots xxx}_{p \text{ bits}}1000000\cdots 000000}^{m \text{ bits}}xxx\cdots$$

or

$$\overbrace{\underbrace{1.xxxxx\cdots xxx}_{p \text{ bits}}0111111\cdots 111111}^{m \text{ bits}}xxx\cdots\ ;$$

Two different algorithms have been suggested for dealing with this problem. The first one, first presented in Lefèvre PhD dissertation [3], [4], allowed Lefèvre and Muller to publish the first tables of HR points for the most common functions in double-precision/binary64 FP arithmetic [5]. We will call it the L-algorithm. The second one, the SLZ algorithm, was introduced by Stehlé, Lefèvre and Zimmermann [6], [7]. It was used to find the HR points for the exponential function in decimal64 arithmetic [8]. SLZ has a better asymptotic complexity than the L-algorithm, however, when the target format is the double precision/binary64 format of the IEEE 754 standard, they require similar computational times: weeks of computation for all input exponents (hours of computation for one input exponent), using massive parallelism.

The problem with these algorithms does not only lie in this huge computation time. In both cases we have a very complex algorithm, implemented in a very complex program, that runs for weeks and just outputs one result: what confidence can we have in that result? The HR points are the main weak point of a library such as CRLibm [9]: each function of that library comes with a theorem of the form "if the HR points have been rightly computed then the function always outputs a correctly rounded result". Hence, our major goal here is to design a very simple, very regular

algorithm (therefore suited for FPGA implementation): if it outputs the same results as the L-algorithm, then this will give much confidence in these results.

The method we are going to suggest here has an even worse asymptotic complexity than the L- and SLZ-algorithms. And yet, due to its simplicity, the hidden constant in the complexity term is so small that, still with double precision/binary64 as a target, our method will require similar times on a single FPGA.

Let us now present our method. Defining $u = 2^{1-p}$, we will compute the values of $f(1) \bmod 2^{1-p}$, $f(1+u) \bmod 2^{1-p}$, $f(1+2u) \bmod 2^{1-p}$, $f(1+3u) \bmod 2^{1-p}$, $\ldots$, $f(2) \bmod 2^{1-p}$, with a given, predetermined, accuracy $2^{-\mu}$ (with μ larger than p and—for probabilistic reasons [2]—less than $2p$). Each time we find a value $f(1+ku) \bmod 2^{1-p}$ extremely close to 2^{-p} (i.e., whose leading bits are of the form $011111111\cdots$ or $10000000\cdots$), we output the value of k for further testing using multiple-precision software. The major difficulty here is that there are 2^{p-1} values $f(1+ku) \bmod 2^{1-p}$ and p is fairly large (a typical value is 53 for the double precision/binary64 format). Therefore, the computation of these values must be done very quickly.

To do this, we will approximate f by some polynomial P (with an accuracy of approximation significantly better than $2^{-\mu}$), and compute the successive values $P(1+ku) \bmod 2^{1-p}$ using a modulo 2^{1-p} adaptation of the well-known *tabulated differences method* [10].

II. The tabulated differences method

Let P be a polynomial of degree n and x_0 a real. We define $x_k = x_0 + ku$ for $k > 0$, and we wish to compute the successive values $P(x_0)$, $P(x_1)$, $P(x_2)$, $P(x_3)$, $\ldots$.

The tabulated differences method is based on the fact that if we define the following "discrete derivatives":

- $P^{(1)}(x) = P(x+u) - P(x)$;
- $P^{(2)}(x) = P^{(1)}(x+u) - P^{(1)}(x)$;
- $\ldots$
- $P^{(n)}(x) = P^{(n-1)}(x+u) - P^{(n-1)}(x)$;

then $P^{(n)}$ is a constant C. This leads to the following algorithm.

Initialization: compute $P(x_0)$, $P(x_1)$, $P(x_2)$, $\ldots$, $P(x_n)$, and deduce from these values all the possible partial discrete derivatives, of which we keep the initial vector at point x_0: $P^{(n)}(x_0) = C$, $P^{(n-1)}(x_1)$, $P^{(n-2)}(x_2)$, $\ldots$, $P^{(2)}(x_{n-2})$, $P^{(1)}(x_{n-1})$, $P(x_n)$.

Iteration: The following recurrence computes the value of this vector at point $x_{k+1} = x_k + u$ out of the vector at point x_k.

$$\left\{\begin{array}{lcl} P^{(n-1)}(x_{k+2}) & = & P^{(n-1)}(x_{k+1}) + C \\ P^{(n-2)}(x_{k+3}) & = & P^{(n-2)}(x_{k+2}) + P^{(n-1)}(x_{k+2}) \\ \vdots & \vdots & \vdots \\ P^{(2)}(x_{k+n-1}) & = & P^{(2)}(x_{k+n-2}) + P^{(3)}(x_{k+n-2}) \\ P^{(1)}(x_{k+n}) & = & P^{(1)}(x_{k+n-1}) + P^{(2)}(x_{k+n-1}) \\ P(x_{k+n+1}) & = & P(x_{k+n}) + P^{(1)}(x_{k+n}) \end{array}\right. \tag{1}$$

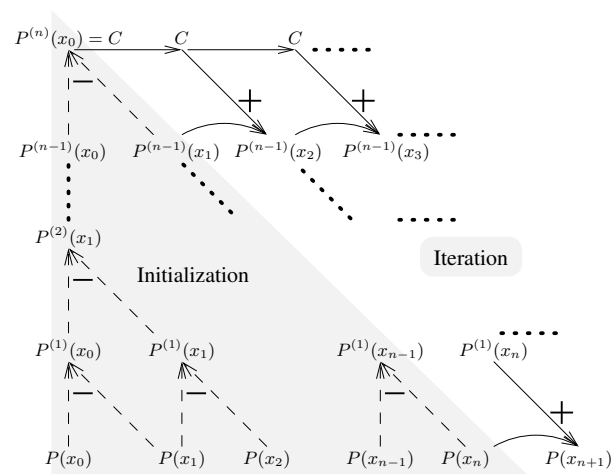

Figure 1. The tabulated difference method

The computations in (1) are done *modulo* 2^{1-p}: they are simple fixed-point additions, with the bits of weight $\geq 2^{1-p}$ being ignored. Notice that the n additions in (1) are straightforwardly pipelined, hence, once the initialization is done *computing a new value $P(x_k)$ takes the time of one addition*.

It should also be noted that the problem is embarrassingly parallel: Although the iteration itself is intrinsically sequential, the full domain of an elementary function in double-precision may be split into arbitrarily many sub-domains, and we may perform initialization/iteration processes in parallel for each sub-domain. We indeed aim at processing hundreds of sub-intervals in parallel within a single FPGA.

Besides, the initial values $P(x_0)$, $P(x_0+u)$, $P(x_0+2u)$, $\ldots$, $P(x_0+nu)$ cannot be computed exactly in practice. They are correct within some rounding error, and we will see in the following that when performing (1), these errors accumulate quite quickly. After some value of k, say $k_{\max}$ this accumulated error becomes unacceptable, and we have to invoke the initialization process again, with x_0 replaced by $x_0 + k_{\max}u$. In other words, the size of a sub-interval is dictated by an error analysis that will be the subject of Section III.

The initialization process requires $n+1$ polynomial evaluations, and $n(n+1)/2$ subtractions. There are two possible ways of performing it:

1) on the FPGA itself, or
2) in software on the host computer.

In both cases we must choose $k_{\max}$ so that the initialization time is totally overlapped by the iterations (1).

The initialization process first involves evaluating the polynomial in $n+1$ points using a classical multiplication-based scheme (typically Horner's). Modern FPGAs contain up to several thousand small multipliers that could be used for this purpose, but designing an architecture for this initialization would add a lot to the FPGA design effort. In the sequel of the paper, we therefore choose the simpler second approach. It also has the advantage of exploiting the computing power of the host processor. However, there is a price to pay: due to limited bandwidth between the host and the FPGA, iteration (1) must run for a much longer $k_{\max}$. We will see in next section that this entails a significantly wider data-path, hence more resource consumption for the iteration hardware, possibly cancelling the benefits of saving the initialization hardware. This question remains to study quantitatively.

Let us now formalize the dependency between $k_{\max}$ and the datapath width.

III. Error analysis

Let us bound the difference between the value computed $F(x_k)$ and the true value of the function $f(x_k)$. We may first decompose this error as follows:

$$F(x_k) - f(x_k) \;=\; (F(x_k) - P(x_k)) + (P(x_k) - f(x_k))$$

The second term is the approximation error, and the Remez approximation algorithm will allow us to keep it as small as needed. Let us focus on the first term, the rounding error.

Let $\delta(i)(x_0+ku)$ be the overall rounding error in the initial evaluation of $P^{(i)}(x_0+ku)$. Notice that the additions in (1) are performed in fixed-point, ignoring the outgoing carries: they do not induce any error, yet they propagate the initial rounding errors on the $P^{(i)}(x_0)$. Let ϵ be a bound on the errors on $P^{(n)}(x_0) = C$, $P^{(n-1)}(x_0+u)$, $P^{(n-2)}(x_0+2u)$, …, $P^{(1)}(x_0+(n-1)u)$. We easily find

$$\delta^{(n-1)}(x_0+ku) \le \delta^{(n-1)}(x_0+(k-1)u) + \epsilon,$$

so that

$$\delta^{(n-1)}(x_0+ku) \le (k+1)\epsilon.$$

From that, we deduce

$$\begin{aligned}\delta^{(n-2)}(x_0+ku) \;&\le\; \delta^{(n-2)}(x_0+(k-1)u)\\ &\quad +\delta^{(n-1)}(x_0+(k-1)u)\\ &\le\; \delta^{(n-2)}(x_0+(k-1)u) + k\epsilon,\end{aligned}$$

so that

$$\delta^{(n-2)}(x_0+ku) \le (1+2+3+\cdots+k)\epsilon = \frac{k(k+1)}{2}\epsilon.$$

Similarly,

$$\begin{aligned}&\delta^{(n-3)}(x_0+ku)\\ &\le \left(1+\tfrac{2(2+1)}{2}+\tfrac{3(3+1)}{2}+\cdots+\tfrac{(k-1)k}{2}\right)\epsilon\\ &= \tfrac{(k-1)(k)(k+1)}{6}\epsilon.\end{aligned}$$

An elementary induction shows that the bound on the error of the computed value of $P(x_0+ku)$ satisfies

$$\delta^{(0)}(x_0+ku) \le \frac{(k-n+2)\cdots(k-1)(k)(k+1)}{n!}\epsilon. \quad (2)$$

IV. A case study: the exponential function

All parameters in the method are function-dependent, so we cannot give a general performance result (although it should not vary much with the function). Hence, we give here some figures related to the exponential function on the input interval $[1,2)$. We take $f(x) = \frac{1}{2}e^x$ to normalise the output to $[1,2)$.

We first split the input interval into 2^m sub-intervals, each of size 2^{-m}, and we will compute one approximation polynomial on each sub interval, using the Remez algorithm. The trade-off here is between the degree of the obtained polynomials (the smaller, the better for the subsequent evaluation) and the number of Remez polynomial to compute. A good choice here is $m = 15$: on each of the $2^{15} = 32768$ sub-intervals, a polynomial of degree 4 approximates $f(x)$ with an accuracy better than 2^{-90}, and the Sollya tool is able to compute all these polynomial and formally validate their accuracy [11] in about 3 hours.

Now we must choose $k_{\max}$, the length of an evaluation run between reinitializations. Having decided that reinitializations are performed on the host processor, we now have to take into account the limits on 1/ computing power of the host processor and 2/ data bandwidth between processor and FPGA. A larger $k_{\max}$ means fewer initializations, but larger data-path, hence slower operation and less parallelism. Note also that if we have P parallel iteration cores, the host must serve them all.

Our current trade-off is to take $k_{\max} = 2^{20}$. Equation (2) with degree $n = 4$ tells us that the error is smaller than $2^{76}\epsilon$. For our target accuracy of 2^{-85} modulo 2^{52}, we need to have $85 - 52 = 33$ valid bits at the end of the computation. A datapath width of $33 + 76 = 109$ bits ensures this accuracy. Assuming $M = 2^8$ parallel iterations on the FPGA, the host must be able to compute one initialization every $2^{20}/2^8 = 4096$ FPGA cycles. FPGA cycles are typically 10 times slower than processor cycles, so the host has roughly 40,000 cycles to compute each initialization. Efficient multiple-precision libraries such as GMP and MPFR make this possible. Host-FPGA bandwidth, in this scenario, is not a problem.

V. Our design

Field-programmable gate arrays (FPGAs) are a natural technology for implementing this type of algorithm:

- for a given set of input parameters we need to perform a big, one-off computation. Once completed we can reconfigure the FPGA for a different set of input parameters.
- the implemented method is based on binary additions for which FPGAs are very efficient.

Nevertheless, implementing such a complex, multi-parametrized design using a hardware description language (HDL) is a tedious and error-prone task. To overcome this, we used the FloPoCo framework [12]. We implemented in FloPoCo an architecture generator that inputs the many parameters and generates an FPGA-specific implementation of the circuit in the form of a portable, human-readable VHDL file.

A. Functional model

1) TaMaDi Core: The core component of our design is the polynomial evaluator based on the tabulated differences method. The architecture of this component is depicted in Figure 2. Its main entities are the $n-1$ adders chained together which are used to evaluate the vector of discrete derivatives.

Each computation starts with the component receiving a '1' value on the `Initialize` line together with a unique interval identifier on the `Interval` bus. During the next $N+1$ clock cycles, the values of the initialization vector $C = P^{(n)}(x_0)$, ..., $P(x_0+nu)$ are received in sequence on the `DataIn` bus, and fill the pipeline. A counter is used to keep track of k in evaluating $P(x_0+ku)$. The output of the nth adder feeds a pattern detector unit, implemented as wide ANDs. A value of '1' at the output of the pattern detector signals that the value present on the output `Counter` bus, together with the interval identifier, points to one HR case. The component raises the `Ready` line to '1' when it has finished the allowed number of iterations and needs a new reinitialization.

The architecture of the TaMaDi Core (Core) is perfectly suited to FPGA hardware. Adders benefit from the fast carry-chains which allow the simple ripple carry adder (RCA) scheme to be implemented efficiently. The wide AND of the pattern detector may also take advantage of these fast carry-chains.

Table I presents area and timing post place-and-route results of the TaMaDi Core on modern FPGAs from Xilinx [13], [14] and Altera [15], [16] for the exponential function example presented in Section IV. The area of one Core occupies a very small fraction of these FPGA. The largest StratixV from Altera(5SGXAB) having 1052K LUTs and 1588K REGs can, in theory accommodate over 1500 Cores while the largest Virtex6 from Xilinx(XC6VLX760) having 758K LUTs and 1516K REGs can accommodate roughly 1000 Cores, if one also considers the interfaces overhead.

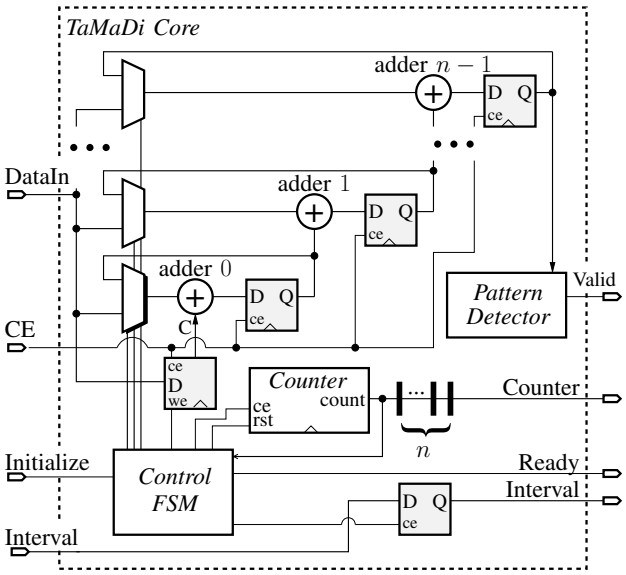

Figure 2. Polynomial Evaluator based on the tabulated differences method

Table I
POST PLACE-AND-ROUTE RESULTS OF THE TAMADI CORE PE

Datapath width	Degree N	FPGA	Frequency	Area LUTs	Area REGs
120	4	StratixIV	237 MHz	584	750
		StratixV	359 MHz	585	750
		Virtex5	262 MHz	640	646
		Virtex6	332 MHz	640	646

2) TaMaDi Cluster: Multiple TaMaDi Cores may be assembled in a larger component named the TaMaDi Cluster, whose architecture is depicted in Figure 3.

The presented system has several parameters:

- M the number of TaMaDi Cores in the system. The maximum value of this parameter depends on the size of one Core and the size of the FPGA. The practical value for this parameter also depends on the bandwidth between FPGA and the host system, processing element datapath width and degree and the reinitialization interval.
- size of the input and output FIFOs, also depends on bandwidth, M and processing element characteristics.
- size of processing element output FIFO. Their dimension can be as small as one element. However, for good performance their size depends on the probability of finding HR cases in that interval and the output bandwidth.

The TaMaDi Cluster is connected to the host system (or the next hierarchical level) my means of two FIFOs. Data is fed by the host system to ClusterInFIFO while this FIFO is not full. Each element on this FIFO has the structure depicted below:

The ClusterInFIFO element contains the necessary infor-

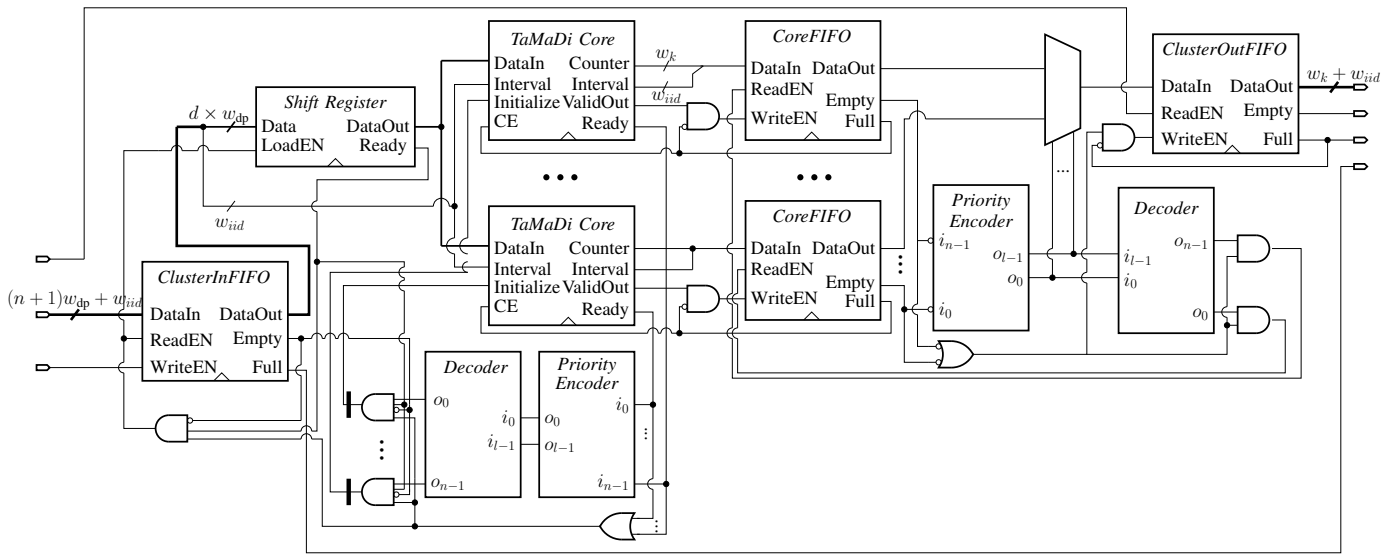

Figure 3. Overview of the TaMaDi Cluster architecture

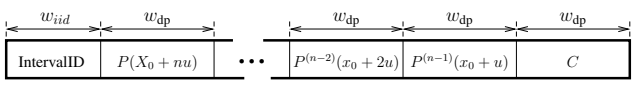

Figure 4. Structure of one element in the ClusterInFIFO

mation to bootstrap one processing element. Once a TaMaDi Core is ready to process new information (signaled '1' on the corresponding output `Ready` port) the input FIFO is popped one element. The uppermost w_{iid} bits of information containing the interval ID are fed to the processing element together with a value on '1' on the corresponding `Initialize` input pin. The lowermost $(N+1)p_w$ bits are loaded into a $N+1$-level shift-register in order be serialized in chunks of p_w bits. During the next $N+1$ clock cycles, the shift-register feeds the TaMaDi Core as the pipeline starts.

When the TaMaDi Cores signals the detection of a HR case, the information concerning this case (counter value and interval identifier, totaling $w_k + w_{\text{iid}}$ bits) is pushed into the corresponding Core output FIFO.

The data from the CoreFIFOs is then placed in the ClusterOutFIFO whenever this FIFO is not full. Simple priority encoders on both inputs and outputs manage the access to the Cluster input and output FIFOs.

3) TaMaDi System: The TaMaDi Cluster has low resource count and fast clock speeds for modest number of Cores (up to 32). Unfortunately, this multiplexed data dispatch architecture scales badly due to several issues: (1) size of the multiplexers and priority encoder-decoder circuitry (2) the long lines between the dispatcher (shift-register in our implementation) and the computing cores (TaMaDi Cores).

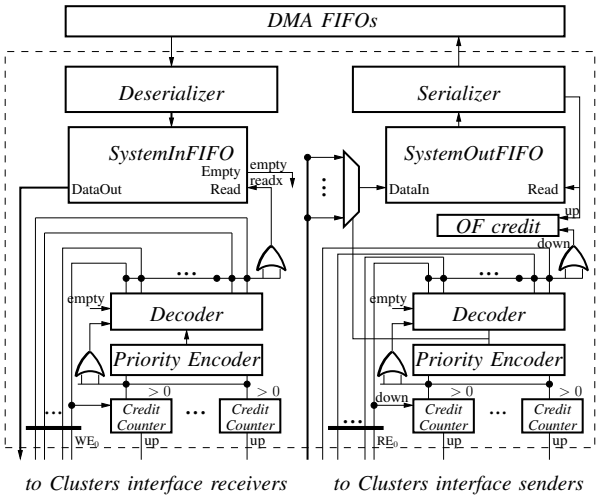

Figure 5. Global system dispatcher interface

By grouping multiple TaMaDi Cores into a small number of Clusters (where the size of the Cluster remains reasonably small say 16), we could potentially use the same dispatching architecture at a macro-level.

However, when filling up a large FPGA chip, a new problem arises: connections between the dispatcher and the Clusters become very long, introducing large delays. To overcome this, we have used a different, credit-based, dispatcher, depicted in Figure 5. It allows us to pipeline the communication lines to Clusters, thus breaking the long delays into several shorter ones. To keep track of data in flight on these long lines, a sender has a credits counter, initialized to the size of the receiver FIFO. The sender sends

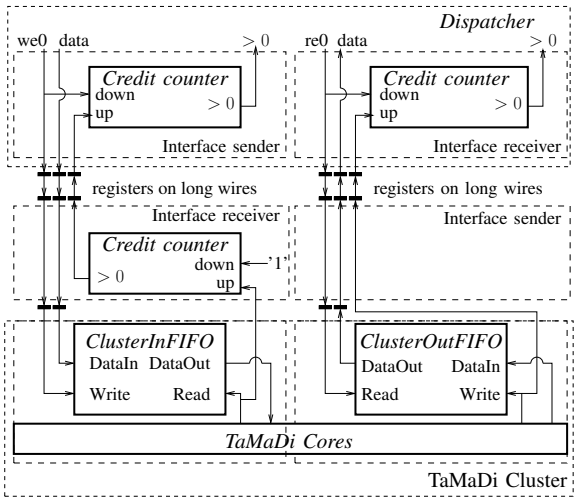

Figure 6. Credit based pipelined communication interface for one Cluster

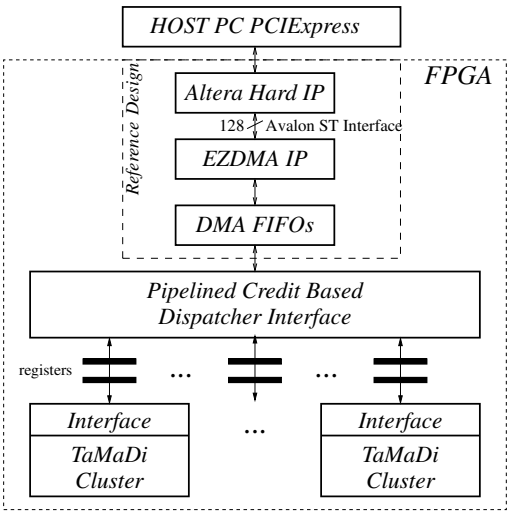

Figure 7. Global system architecture

only when its credit counter is positive, decrements it on each send, and blocks when is is zero. The receiver sends a credit back to the sender at each read from its FIFO.

A detailed view of this sender/receiver interface is presented in Figure 6. The receiver part of the interface is tightly coupled to the TaMaDi Cluster.

A global dispatcher circuit manages the communication with the host. It has two functionalities (1) dispatching reinitialization data to the TaMaDi Cluster (Figure 5 left) and (2) retrieval of HR cases from these modules (Figure 5 right).

4) Full Prototype: For prototyping purposes we use the Stratix IV GX development board featuring a Stratix IV GX EP4SGX530KH40C2 FPGA. The board communicates with a host PC by means of a PCI Express 2.0 8x interface that can provide up to 3.4 GB/s full-duplex.

The Altera PCI Express hard IP together with the PLDA EZDMA2 IP [17] offered in the PLDA reference design ensure a simple FIFO interface for our pipelined credit-based Dispatcher Interface (Figure 7). The PLDA host driver offers a high-level API interface for feeding and retrieval of information from the DMA FIFOs by means of multiple DMA channels (2 in our case).

B. Bandwidth requirement

In this section we will compute the bandwidth requirement of the entire TaMaDi System. First, we need to compute the bandwidth of TaMaDi Cluster depicted in Figure 3.

We use the following notations:

f circuit frequency
K the number of TaMaDi Clusters
M number of TaMaDi Cores within a Cluster
N approximation polynomial degree
w_{dp} the datapath width on the TaMaDi Core
$k_{\max}$ the number of iterations between re-initializations
w_k $= \lceil \log_2(k_{\max}) \rceil$, width in bits of the iteration counters.
η the maximum number of intervals to be processed by the system
w_{iid} $= \lceil \log_2(\eta) \rceil$, width in bits of interval identifiers
ξ the probability of finding one HR case

The input bandwidth for one TaMaDi Core:

$$B^{\text{in}}_{\text{Core}} = ((N+1)w_{\text{dp}} + w_{\text{iid}}) \frac{f}{k_{\max} + N + 1} \quad (3)$$

and the output bandwidth of one Core is:

$$B^{\text{out}}_{\text{Core}} = \xi(\lceil log_2(k_{\max}) \rceil + w_{\text{iid}}) \frac{f}{k_{\max} + N + 1} \quad (4)$$

The Core bandwidth is $B_{\text{Core}} = B^{\text{in}}_{\text{Core}} + B^{\text{out}}_{\text{Core}}$. Considering that a TaMaDi Cluster has M such Cores, the total bandwidth requirement for a Cluster is $B_{\text{Cluster}} = M \cdot B_{\text{Core}}$. A TaMaDi System is composed out of K Clusters, therefore requiring a bandwidth equal to: $B_{\text{System}} = K \cdot B_{\text{Cluster}} = K \cdot M \cdot B_{\text{Core}}$.

Table II presents the dependency between the parameters of a Core, its area and the required bandwidth for keeping it busy at 100MHz. A larger bandwidth requirement leads to more pressure on the I/Os but a smaller Core size, which allows fitting more in one single FPGA.

For a system comprising of 100 TaMaDi Cores, each requiring a bandwidth of 5.42 MBit/s the bandwidth requirement is approximatively 0.5 Gb/s which seems to be reasonably within our available bandwidth potential. Nevertheless, this configuration would require us generating more than 57 TBytes of reinitialization data (some of which can indeed be generated on-the-fly) compared to a more manageable 1.9 TBytes required for a $w_{\text{dp}} = 120$.

For a 200-Core system $B_{System} = 3.4 Mbits/s$, which can easily be provided by FPGA platforms connected

Table II
DEPENDENCY BETWEEN TAMADI CORE PARAMETERS, ITS AREA AND THE NECESSARY BANDWIDTH/CORE FOR A STRATIXIV. SIMILAR RESULTS HOLD FOR OTHER FPGAS

Parameters				Area		Bandwidth
N	w_{dp}	$k_{\max}$	w_{iid}	LUTs	REGs	(Mbit/s)
4	120	2^{20}	32	584	750	0.017
4	81	8,192	39	400	531	5.42

through to the host system through the PCIE bus, Ethernet interface and even USB2.0.

C. Performance estimation

We are currently using the Altera StratixIV development kit based on an EP4SGX530KH40C2 FPGA to prototype our system. This gives us an environment for estimating the performance and scalability of our architecture. However, as presented in this section the amount of computation needed for an elementary function, on one exponent value is as the order of tens of hours. We therefore envision mapping this architecture on even larger FPGAs, and even multi-FPGA based systems.

In this section we provide a performance estimation for the case of the exponential function for double precision $p = 52$ (we consider one input exponent). Considering the 2^{20} iterations until having to reinitialize the Core, the total number of intervals to process is about $4.3 \cdot 10^9$.

Table III shows the dependency between system frequency, number of Cores and task completion.

On our current FPGA prototyping system we conservatively estimate to be able to pack 200 PE which yields an realistic execution time of approximatively 50H.

D. Reality Check

We have tested the real performance of different configurations of our proposed systems on two FPGAs, our prototyping StratixIV development-kit and a large StratixV FPGA. The purpose of these tests was to show the *performance* of our solutions and to determine the degree of *scalability* of the proposed architectures (both the simple-dispatcher and the credit-based dispatcher solution). The results are shown in Table IV

First, the results obtained validate that, once the number of Cores exceeds a certain threshold (32 for StratixIV), the credit-based dispatch offers a more attractive solution. It is expected that, as the number of Cores scales up, the credit-based dispatch will continue to function at frequencies over 150 MHz, as the critical path of this system is in the adders of the TaMaDi Core.

Table III
PERFORMANCE ESTIMATES FOR DOUBLE-PRECISION EXPONENTIAL (ONE INPUT EXPONENT)

		Frequency		
		100	150	200
Cores	100	125h	83.4h	62.5h
	200	62.5h	41.7h	31.3h
	400	31.2h	20.9h	15.7h
	800	15.7h	10.4h	7.9h

Secondly, the results obtained on the StratixV FPGA prove that the increased capacity linearly improves the task completion time.

However, when reading this table one should consider that, as the size of the FPGA increases linearly, the time needed to compile the project on the FPGA increases at best polynomially. In other words, if for the 16 cores StratixIV design, compilation took some tens of minutes on a fast server, the StratixV designs took tens of hours to compile.

A solution to improve the compile/execution time ratio is to use a multi-FPGA based system, comprising of multiple similar FPGAs, such as the multi-FPGA prototyping board DN7020K10 from Dini Group [18], comprising 16 Altera StratixIV FPGAs. The TaMaDi System would be compiled once, then replicated on these FPGAs. For simplicity one FPGA will also contain a dispatcher interface and will be connected to the host system. We estimate that one such system would complete the execution of one exponent in less than 2 hours.

All in all, depending on the available FPGA, the order of magnitude of the time required to process one input exponent is between a few hours and two days. Although the double precision/binary64 format has 2046 possible exponents, we do not need to perform such a calculation for every exponent: the exponential of a number larger than 710 is an overflow, and if $|u| \leq 2^{-54}$, then e^u correctly rounded to that format is equal to 1. The most up-to-date implementation of the L-algorithm takes 45 hours to process one input exponent on a fairly recent FP core (AMD Opteron 2.19 GHz). Since these algorithm are *very* different and are run on very different machines, we suggest using both of them, which allows one to get much confidence in the obtained results.

CONCLUSION

We have suggested an algorithm and an FPGA architecture that make it possible to find hardest-to-round points for elementary functions in double precision. This requires huge computations, but they are done once for all, and allow one to design efficient libraries or hardware for elementary function evaluation. The achieved performance is slightly better than the one obtained using Lefèvre's L-algorithm but the real gain is not there: it lies in the fact that if, with a completely different method that runs on a completely different hardware, we obtain the same results, this gives much confidence in these results.

REFERENCES

[1] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008, available at http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

Table IV
POST PLACE-AND-ROUTE RESULTS OF THE TAMADI SYSTEM. THE CORE PARAMETERS ARE: $w_{\text{DP}} = 120$ BITS AND $N = 4$

FPGA	Cores	Freq.	Area			Completion
			LUTs	REGs	M9K	Time
StratixIV (EP4SGX530KH40C2) *simple-dispatch*	16	196 MHz	10,614 (2%)	14,725 (3%)	17	398.9h
	32	174 MHz	20,021 (4%)	26,250 (6%)	17	224.7h
	64	154 MHz	48,416 (11%)	61,234 (14%)	148	126.9h
	128	111 MHz	95,428 (22%)	118,944 (28%)	276	88.05h
	256	97 MHz	189,298 (45%)	234,432 (55%)	532	50.4h
StratixIV (EP4SGX530KH40C2) *credit-based dispatch*	16 (2x8)	198 MHz	13,159 (3%)	21,586 (5%)	53	394.9h
	32 (4x8)	193 MHz	25,014 (6%)	39,402 (9%)	87	202.6h
	64 (8x8)	168 MHz	59,213 (14%)	89,051 (21%)	308	116.4h
	128 (16x8)	168 MHz	96,534 (22%)	156,370 (36%)	592	58.2h
	256 (32x8)	127 MHz	232,649 (54%)	348,335 (82%)	1172	38.5h

[2] J.-M. Muller, *Elementary Functions, Algorithms and Implementation*, 2nd ed. Birkhäuser Boston, MA, 2006.

[3] V. Lefèvre, "Moyens arithmétiques pour un calcul fiable," Ph.D. dissertation, École Normale Supérieure de Lyon, Lyon, France, 2000.

[4] ——, "New results on the distance between a segment and $\mathbb{Z}^2$. Application to the exact rounding," in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*. IEEE Computer Society Press, Los Alamitos, CA, Jun. 2005, pp. 68–75.

[5] V. Lefèvre and J.-M. Muller, "Worst cases for correct rounding of the elementary functions in double precision," in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, N. Burgess and L. Ciminiera, Eds., Vail, CO, Jun. 2001.

[6] D. Stehlé, V. Lefèvre, and P. Zimmermann, "Worst cases and lattice reduction," in *Proceedings of the 16th Symposium on Computer Arithmetic (ARITH'16)*. IEEE Computer Society Press, 2003, pp. 142–147.

[7] ——, "Searching worst cases of a one-variable function," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 340–346, Mar. 2005.

[8] V. Lefèvre, D. Stehlé, and P. Zimmermann, "Worst cases for the exponential function in the IEEE 754r decimal64 format," in *Reliable Implementation of Real Number Algorithms: Theory and Practice*, ser. Lecture Notes in Computer Sciences, vol. 5045. Springer, Berlin, 2008, pp. 114–126.

[9] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C. Q. Lauter, and J.-M. Muller, "CR-LIBM, a library of correctly-rounded elementary functions in double-precision," LIP Laboratory, Arenaire team, Available at https://lipforge.ens-lyon.fr/frs/download.php/99/crlibm-0.18beta1.pdf, Tech. Rep., Dec. 2006.

[10] D. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley, Reading, MA, 1998, vol. 2.

[11] S. Chevillard, J. Harrison, M. Joldes, and C. Lauter, "Efficient and accurate computation of upper bounds of approximation errors," *Theoretical Computer Science*, vol. 412, no. 16, pp. 1523 – 1543, 2011.

[12] "FloPoCo, Floating-Point Core generator," http://flopoco.gforge.inria.fr/.

[13] *Virtex-5 FPGA User Guide*, Xilinx, 2009, http://www.xilinx.com/support/documentation/user_guides/ug190.pdf.

[14] *Virtex-6 FPGA User Guide*, Xilinx, 2011, www.xilinx.com/support/documentation/virtex-6.htm.

[15] *Stratix IV Device Handbook*, Altera, 2011, http://www.altera.com/literature/hb/stratix-iv/stratix4_handbook.pdf.

[16] *Stratix V Device Handbook*, Altera, 2011, http://www.altera.com/literature/hb/stratix-v/stratix5_handbook.pdf.

[17] "PLDA EZDMA2 DMA for PCI Express Hard IP," http://www.plda.com.

[18] "DN7020K10 'Uncle of Monster' Altera Stratix IV ASIC Prototyping Engine," http://www.dinigroup.com.