Single Precision Natural Logarithm Architecture for Hard Floating-Point and DSP-Enabled FPGAs

Martin Langhammer, Bogdan Pasca

Single Precision Natural Logarithm Architecture for Hard Floating-Point and DSP-Enabled FPGAs

Martin Langhammer, Bogdan Pasca
Altera European Technology Centre, UK

***Abstract*—Elementary function design has recently been added yet another level of flexibility with the integration of single precision addition and multiplication into the Arria10 DSP block architecture. Implementation techniques developed having floating-point operations support in mind are only available for microprocessors and lead to slow and high-cost implementation when naively ported to FPGA architecture. In this article we show how the new features can be used in conjunction with the existing resources in the design of the natural logarithm elementary function. Compared to traditional FPGA implementations we use Taylor expansion based techniques which enable the use of floating-point adders and multipliers available in DSP blocks for the polynomial evaluation. We show the various tradeoff points of the architecture together with expansion-based techniques for increasing the internal precision in order to produce OpenCL conforming operators. The presented architecture proposes a possible resource tradeoff with significantly lower logic reduction on Arria10 devices.**

I. Introduction

Elementary function design for contemporary FPGAs is a complex task due to the multitude of implementation tradeoffs exposed by the architecture features: logic, memory blocks and fixed-point DSP blocks. For instance digit-recurrence methods make extensive use of logic resources while polynomial approximation techniques use memory and DSP blocks; the ratio between memory and DSP blocks can varied by changing the approximation polynomial degree. The most recent Arria10 FPGA devices add another architectural feature which can be used when implementing floating-point elementary functions: direct silicon support for floating-point addition and multiplication.

Microprocessor implementations of elementary functions already make use of floating-point arithmetic. Floating-point units support fast execution of addition and multiplication, and use these basic operations in software routines for implementing more complex elementary functions. For most functions the software routine contains multiple branches of execution, depending on the range of the input. Within these branches various techniques for approximating the function are used: some branches use high degree polynomial evaluation (may go up to degree 20 or 30), rational polynomial approximations, Taylor expansions, digit-recurrence methods, quadratic convergence methods etc. The arithmetic used internally is higher precision, for instance double precision elementary functions use double-double and triple-double implementations [1].

Porting these implementations directly to the FPGA architecture by means of high-level synthesis tools is inefficient. The disjoint nature of execution branches potentially allows for resource sharing, however the different algorithms used within the branches makes sharing difficult. Within branches, the high degree polynomial evaluations require a significant number of additions and multiplications for high-throughput implementations. Efficient FPGA implementations need to make use the entire mix of FPGA features. For instance, the high-degree polynomial evaluations may be replaced by piecewise polynomial approximations where numerous low-degree polynomials are used to provide an equivalent approximation quality. The coefficients for these low-degree polynomials would be stored in the memory blocks, available in thousands on modern FPGA devices. Other bit-fiddling techniques, costly in microprocessors but virtually free in FPGAs, can be used during the range reduction and reconstruction stages.

In this article we show how the new floating-point resources available in the DSP block may be used to further improve the FPGA-specific implementation of the single-precision natural logarithm.

II. Background

A. Floating-Point

Let x be the floating point input such that:

$$x = (-1)^s 2^e M$$

where s denotes the sign of the number – with values $\in \{0,1\}$, e denotes the exponent and M denotes the mantissa. The IEEE-754 standard for floating-point arithmetic [2] uses a normalized mantissa with $m \in [1,2)$. Since the leading bit of the binary mantissa representation will be a constant one, it is omitted in the representation. Consequently we use the following notation:

$$x = (-1)^s 2^e 1.f$$

where f denotes the fraction of the floating-point number. The number of bits used to represent the exponent and fraction give the different floating-point formats presented by standard – Table I. When dealing with floating-point arithmetic a useful tool when managing errors is the the notion of *ulp*, which is, as defined by Harrison [3] the distance between two adjacent floating-point numbers.

In this article we focus on an natural logarithm implementation targeting single-precision.

Table I
FORMATS DEFINED BY THE IEEE-754 STANDARD FOR FLOATING-POINT ARITHMETIC: WE – NUMBER OF BITS FOR THE EXPONENT; WF – NUMBER OF BITS FOR THE FRACTION.

Format (wE, wF)	Name (IEEE-754)	IEEE-754-2008
(5,10)	half precision	binary16
(8,23)	single precision	binary32
(11,52)	double precision	binary64
(15,112)	quadruple precision	binary128

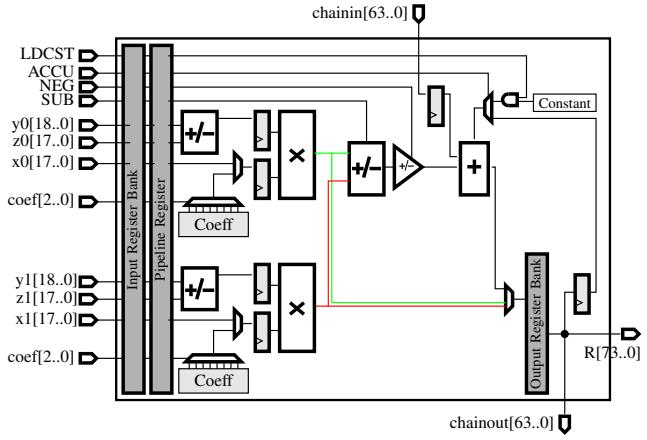

Figure 1. Arria10 DSP block in dual 19x18-bit fixed-point multiplier mode

B. FPGA

The deployment target for our implementation is a modern FPGA architecture having silicon support for basic single-precision floating-point arithmetic – the Arria10 FPGA [4]. The relevant Arria10 features for this work are:

- memory blocks M20K which can be configured in 512x40-bits or 1024x20-bits;
- DSP blocks which can be configured in fixed-point mode to execute one 27x27-bit multiplication, 2 independent 18x19 multiplications (Figure 1) and one sum-of-two 18x19-bit multiplications;
- the same DSP blocks can be configured in floating-point mode to execute one single-precision: addition, multiplication, accumulation, one multiply-add operation or one multiply-accumulate operation (Figure 2).

III. NATURAL LOGARITHM TECHNIQUES

Computing the natural logarithm for x starts with applying the logarithm properties on the floating-point representation of x:

$$\begin{aligned}\log(x) =& \log((-1)^s 2^e 1.f)\\ =& e\log(2)+\log((-1)^s 1.f)\end{aligned}$$

The natural logarithm is only defined on positive inputs, so the above formula can be simplified:

$$\log(x) = \begin{cases} e\log(2)+\log(1.f) & s=0,\\ NaN & s=1 \end{cases}$$

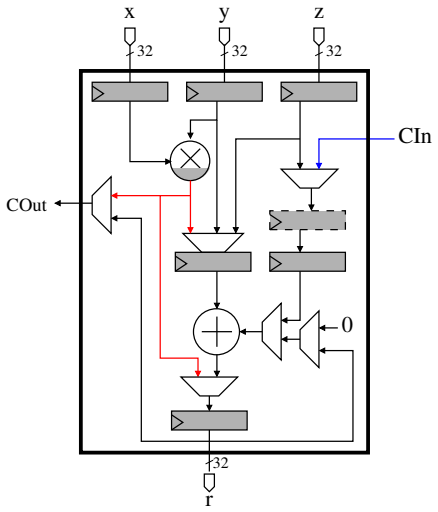

Figure 2. DSP block in floating-point configuration

We will focus in the following on the first branch of this function. The sum $e\log(2)+\log(1.f)$ may be the result of a massive cancellation when $e=-1$ and $1.f\to 2$. In order to return the sufficient number of meaningful result bits for a floating-point result the fixed-point precision of the calculation, together with the accuracy of both terms needs to significantly increase. Alternatively, if the massive cancellation is prevented then there will be no loss of accuracy and hence no need for an extended internal precision. This is accomplished by means of the following rewrite:

$$\log(x) = \begin{cases} e\log(2)+\log(1.f) & 1.f<\sqrt{2},\\ (e+1)\log(2)+\log(\frac{1.f}{2}) & 1.f\geq\sqrt{2} \end{cases}$$

Using this rewrite the previous cancellation triggers the second branch. When $1.f$ becomes greater than $\sqrt{2}$ and $e=-1$ the first term becomes zero and all the accuracy is then returned by the second term.

The following notation is used:

$$\log(x) = E\log(2)+\log(m)$$

where

$$E = \begin{cases} e & 1.f<\sqrt{2},\\ e+1 & 1.f\geq\sqrt{2} \end{cases} \quad (1)$$

and

$$m = \begin{cases} 1.f & 1.f<\sqrt{2},\\ \frac{1.f}{2} & 1.f\geq\sqrt{2} \end{cases} \quad (2)$$

IV. IMPLEMENTATION

In this section we discuss the various implementation opportunities for the natural logarithm on modern FPGAs.

A. Computing $\log(m)$

The second term $\log(m)$ with $m\in[\frac{\sqrt{2}}{2},\sqrt{2})$ (or $\approx[0.7,1.41)$ in decimal) is depicted in Figure 3. The value of $\log(m)$ may be computed using a Taylor series for $\log(1+y)$ where m

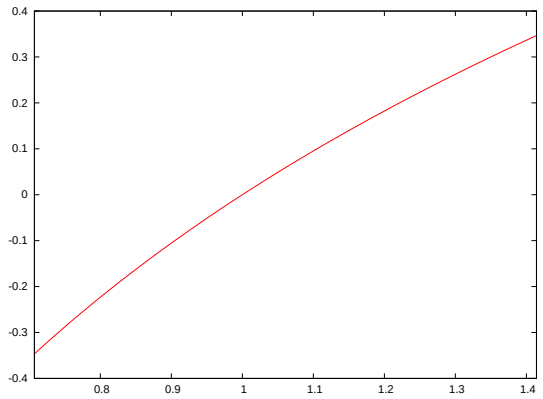

Figure 3. $\log(m)$, $m \in [\frac{\sqrt{2}}{2}, \sqrt{2})$

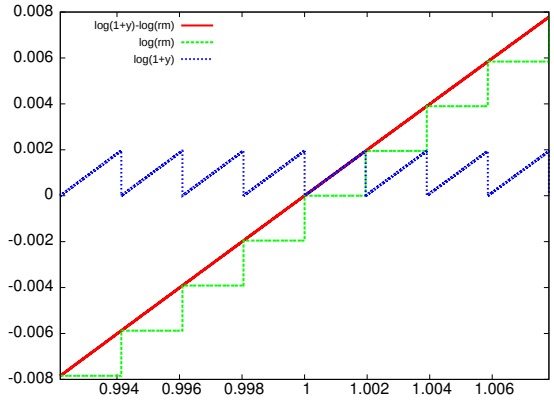

Figure 4. $\log(1+y) - \log(r_m)$ in red; $\log(1+y)$ in blue; $-\log(r_m)$ in green

is sufficiently close to 1 (y sufficiently small). The Taylor expansion used has the form:

$$\log(1+y) = y - \frac{y^2}{2} + \frac{y^3}{3} - \frac{y^4}{4} + ...$$

Truncated Taylor series (Taylor expansions where only a finite number of terms are kept) are used in practice for computing an approximation of the result. The accuracy of the approximation depends on the number of terms dropped; in Taylor expansions the higher order terms have increasingly lower contributions to the final result.

For a given input and output precision p (for single precision $p = 1+24$) and a small interval for y ($|y| < 2^{-y_{\max}}$) the higher order terms with contributions smaller than the *ulp* of the maximum magnitude term can be dropped. In other words in the expansion below,

$$\log(1+y) = y(1 - \frac{y}{2} + \frac{y^2}{3} - \frac{y^3}{4} + ...$$

as long as the values of terms such as $\frac{y^3}{4}$ are smaller than 2^{-p-2} these values will not contribute to the final result, and can therefore be truncated.

The current range of $m \in [\frac{\sqrt{2}}{2}, \sqrt{2})$ translates into $y \in [-0.3, 0.41]$. Due to this wide range of y, the truncated Taylor series with sufficient accuracy would have tens of terms (same order as the required precision) which leads to a costly implementation for a high-throughput FPGA architecture. A general technique used is to reduce the range of the input so that the computation is less costly. The final result is obtained after a reconstruction stage and is based on the computation on the reduced argument. Conversely, a microprocessor-based implementation would prefer the a high number of floating-point operations to the branching required by the range reduction.

B. Range Reduction

The range reduction used is a multiplicative one, which allows us to take advantage of the properties of the logarithm:

$$\log(a/b) = \log(a) - \log(b)$$

where a is desired to have he form $1+y$.

Ideally, we want to represent $m = (1+y)/r_{m_{\text{top}}}$ where $r_{m_{\text{top}}}$ is easy to compute and y is small, say $|y| \leq 2^{-9}$. For this we first choose m_{top} as the most significant 9 bits of m; the inverse of m_{top} denoted $r_{m_{\text{top}}}$ can easily be computed via tabulation. Next, the value of y is simply obtained:

$$y = m r_{m_{\text{top}}} - 1.$$

V. Implementation Accuracy

Our logarithm implementation computes final result as the sum of three terms:

$$\log(x) = E\log(2) + (\log(1+y) - \log(r_m)) \qquad (3)$$

which we denote for simplicity by terms $\widetilde{A}$, $\widetilde{B}$ and $\widetilde{C}$. Let us assume that the three terms are computed in floating-point as accurately as possible, that is to say, each having a maximum error of half *ulp*. We use the following notation where the $\widetilde{A}$ variables represent values post rounding, non tilde variables are mathematical values and δ represents the value of a half *ulp*.

$$\widetilde{A} = A(1+\delta_A)$$
$$\widetilde{B} = B(1+\delta_B)$$
$$\widetilde{C} = C(1+\delta_C)$$

and:

$$\begin{aligned} R &= (\widetilde{A} + (\widetilde{B} - \widetilde{C})(1+\delta_d))(1+\delta_s) \\ &= (\widetilde{A} + (B(1+\delta_B) - C(1+\delta_C))(1+\delta_d))(1+\delta_s) \\ &= (\widetilde{A} + (B - C + B\delta_B - C\delta_C))(1+\delta_d))(1+\delta_s) \\ &\leq (\widetilde{A} + (B-C)(1 + \frac{B\delta_B - C\delta_C}{B-C})(1+\delta_d))(1+\delta_s). \end{aligned}$$

If $B - C << \max(B, C)$ the error present in computing B or C (whichever is larger) will be amplified by this amount.

Figure 4 plots the size of the cancellation between the B and C terms. As expected, the cancellation occurs when m is close to 1, either approaching from below or from above

within a 2^{-9} region. This case is handled separately (in this case there is no need for range reduction) and allows for the B-C computation to provably have no major cancellation when a range reduction stage is required.

Now provided that the 3 terms $\widetilde{A}$, $\widetilde{B}$ and $\widetilde{C}$ are computed accurately to 1/2 *ulp*, the formula will allow us to compute a natural logarithm within 1.5 *ulp*.

VI. Implementation discussion

A. Computing term $\widetilde{A} = E\log(2)$

The product $E\log(2)$ can be computed using either a constant multiplier or a pure tabulation. For a constant multiplier implementation, due to the binary representation of the constant ($\log(2) =$`0.10110001011100100001011`) which does not seem to present any obvious regularity or simplicity the KCM constant multiplication algorithm is preferred [5].

The KCM based implementation requires rewriting E as a sum:

$$E = 2^4 E_{\text{high}} + E_{\text{low}}$$

with the product:

$$E\log(2) = 2^4 E_{\text{high}}\log(2) + E_{\text{low}}\log(2).$$

The two terms are obtained by tabulation, and one integer addition is then used to obtain the final result in fixed-point format. A floating-point output can be obtained by normalising this value (left shift of at most 8 positions is required).

A floating-point output can be obtained alternatively using a tabulation which returns the result directly in floating-point. The exponent is used to address a table (M20K configured in 256x40-bit mode) which stores all the possible values of the product. This solution would uses 1 M20K and no ALMs.

Both solutions allow for computing the term with a 1/2 *ulp* error bound.

B. Computing term $\widetilde{C} = \log(r_{m_{top}})$

When tabulating the natural logarithm of the inverse $r_{m_{\text{top}}}$ we make use of the property that $m < 1$ only when $1.f > \sqrt{2}$ which is also our branch condition. Therefore, the table address will hold the branch bit and the top 9 bits of f (hidden one is not used but accounted for). This allows us to double the accuracy of the inversion when $m < 1$ as there is one extra bit of information used in the input. Since this result is obtained through tabulation, the accuracy is guaranteed to be within 1/2*ulp*.

C. Computing term $\widetilde{B} = \log(1+y)$

We target computing y directly in floating-point. The term m is exact and carries no error into the calculation. The second term r_m is not exact, and carries into the calculation a maximum error of half *ulp*. The product mr_m will approach 1 by construction, and therefore the final subraction will not introduce any additional error, but will amplify the existing error by the cancellation size.

$$\begin{aligned} y &= mr_m(1+\delta_r) - 1 \\ &= mr_m - 1 + mr_m\delta_r \end{aligned}$$

Consider we are working in precision p with an unbounded exponent range, then $\delta_r \leq 2^{-p-1}$. In the above formula we inject the maximum error value and therefore have:

$$y = mr_m - 1 + mr_m 2^{-p-1}$$

Consider a cancellation size ψ, then the error in the y term is:

$$\begin{aligned} y &\approx (mr_m - 1)(1 + \frac{mr_m 2^{-p-1}}{mr_m - 1}) \\ &\approx (mr_m - 1)(1 + \frac{mr_m 2^{-p-1} 2^{\psi}}{2^{\psi}(mr_m - 1)}) \\ &\approx (mr_m - 1)(1 + 2^{-p-1+\psi}) \end{aligned}$$

This actually shows that given a ψ-bit cancellation -which we will get by multiplying m by a roughly ψ-bit accurate approximation of its inverse - the error resulted when computing y provided that no rounding error was actually performed on mr_m is roughly 2^{ψ} *ulp*. The value y is then used to compute a truncated Taylor expansion for $\log(1+y)$, which is term B in our result equation. Due to the linearity of the function on the interval close to 1, the 2^{ψ} *ulp* error in the input will roughly translate to a 2^{ψ} *ulp* error in the output.

$$\begin{aligned} \log(1+x) &= x - x^2/2 + x^3/3 - x^4/4 + ... \\ \log(1+x(1+\delta_x)) &= x(1+\delta_x) - x^2(1+\delta_x)^2/2 + ... \end{aligned}$$

This error will be the final error when both terms A and C will be zero in our equation. This will be the case then $E = 0$ for term A and $r_m = 1.000000000$. Fortunately this case is captured separately by the no range-reduction branch (triggered when m is within an interval of size 2^{-9} away from 1).

The 2^{ψ} *ulp* error is present nonetheless for inputs slightly larger than $1 + 2^{-9}$ and is visible in Figure 4 when terms $\widetilde{B}$ and $\widetilde{C}$ have roughly the same magnitude and $\widetilde{A} = 0$.

Therefore, the accuracy of $1 + p$-bits stored in r_m is not sufficient having this ψ bit cancellation. Storing more accuracy in r_m is impossible using the single precision format (that was chosen to match the hardware support). The requested error for r_p needs to be smaller than $2^{-p-1-\psi}$ provided that we can produce the mr_m product without rounding errors.

D. Improved accuracy range reduction

We store r_m in fixed-point on a precision $1 + p + \psi$ bits. The multiplication mr_m is a fixed point multiplication with the result very close to 1; m is multiplied by a $1 + p + \psi$ bit accurate inverse computed using only the leading ψ bits of m. The result will have a deterministic position for the leading one, but after the subtraction $mr_m - 1$ there is no guarantee where the leading one is found. Hence, a classical leading zero counter and left shifter (normalization stage) is needed for converting this value into floating-point.

In this paper we present a novel way of performing this calculation efficiently using the available hard floating-point DSPs. The main goal of this stage is to obtain an accurate

```
mr = 1.000000000XXXXYYYYYYYYYYYYY
 j = 1.000000000XXXX             +
 i =               1YYYYYYYYYYYYY-
 k =               1
```

Figure 5. Fixed-point product to floating-point decomposition showing i, j and k mantissa alignments

floating-point value y which can then feed into the Taylor polynomial evaluation stage.

We accomplish the translation between the fixed-point product mr_m, via the difference $mr_m - 1$ into floating-point by exploiting the format of this product: leading '1' followed by a number of zeros followed by information bits.

```
mr = 1.000000000XXXXYYYYYYYYYYYYY
```

The idea is to represent the fixed-point product mr_m as a sum of floating-point numbers having a small overlap: $mr_m = j + i - k$. Once accomplished, the difference $mr_m - 1$ can be performed using floating-point arithmetic only.

A very efficient way of obtaining floating-point values from the fixed-point product mr_m is to:

- take j as the most significant 24-bits of the product;
- inject a '1' having a weight equal to the LSB position of j and take i as the 24 bits starting with this injected '1' and the next 23 bits of the product;
- since an artificial '1' was inserted it needs to be subtracted using k.

The leading '1' injected into i is used in order to avoid the leading-zero counting for obtaining i and obtain directly a normalized floating-point value.

Figure 5 shows the fixed-point alignments of the mantissas of the 3 floating-point values created against the fixed-point product mr_m.

However, the full computation of y further requires subtracting the floating-point value '1'. The required order of operations is:

$$\begin{aligned} y &= mr_m - 1 \\ &= (j + i - k) - 1 \\ &= (j - 1) - k + i \end{aligned} \tag{4}$$

One of the 3 floating-point operations in Equation 4 can be saved by restructuring the calculation as follows:

$$y = (j - (1 + k)) + i \tag{5}$$

and observing that the weight difference between '1' and k, which are both constants, allows packing them into one single floating-point mantissa, without actually preforming a floating-point addition.

The extra accuracy required for y before feeding into the polynomial evaluation stage is obtained after the addition of the i term. The typical values for ψ are 8-11 bits, meaning that 8-11 bits of m are used to address a table which outputs a $1 + p + \psi$ precision value. These values are selected such

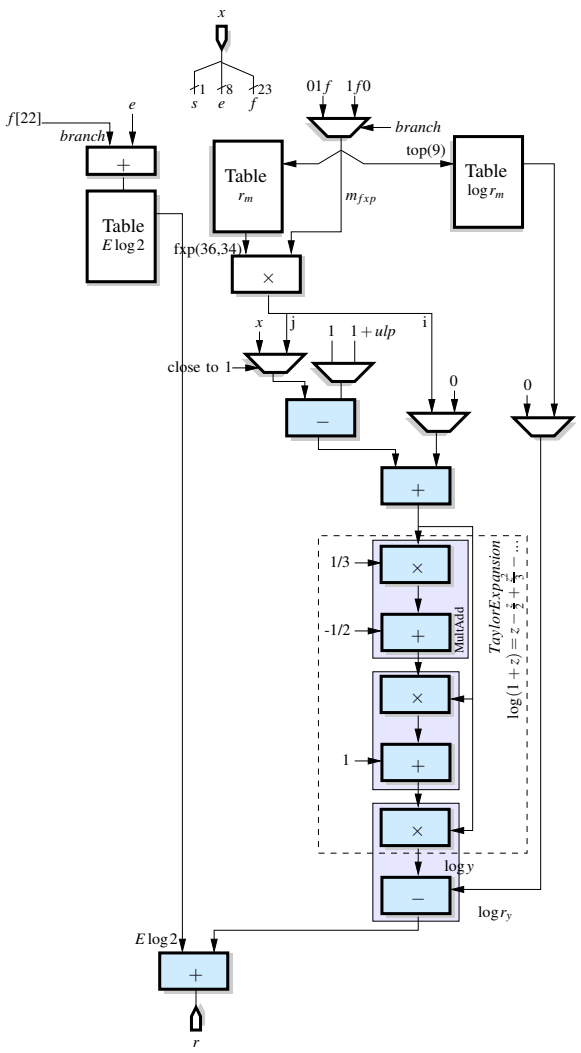

Figure 6. Architecture for the single-precision natural logarithm

to match the characteristics of the embedded memory blocks M20k.

As previously stated, the value of the product mr_m will come close to 1, most of the time having a value larger than 1. Since $m \geq m_t$ where m_t represent the leading ψ bits of m, then $m \times 1/m_t$ will be larger than 1 except when $m = m_t$. In this exceptional case the value of $\log(1+y) - \log(r_m)$ will be returned by the tabulated $\log(r_m)$ term.

VII. Architecture

The architecture of the single precision natural logarithm is depicted in Figure 6. For simplicity, the floating-point input x is split into its 3 components, the sign, the exponent and the fraction. The branch condition ($1.f > \sqrt{2}$) is replaced for simplicity by $1.f > 1.5$. This reduces to using the MSB of the fraction as the branch condition. The $\widetilde{A}$ term is obtained on the left from a table indexed by E (see Equation 1).

The m argument (Equation 2) is calculated by the central multiplexer by selecting between the f and $f \gg 1$ according

to the branch condition. Using the leading 10-bits of m (m_{top}) the fixed-point inverse ($r_{m_{\text{top}}}$) is computed using a table on an extended precision. A fixed-point multiplier then allows us to obtain ($mr_{m_{\text{top}}}$). Next, the floating-point subtracter and adder implement Equation 4 in order to obtain y. Data is fed to these units through a network of multiplexers which allow integrating the case when $|m| \leq 1+2^{-9}$ within the same datapath. When *close* is high, the subtracter inputs x and the constant 1, while the adder will simply add zero to this result; additionally, the third term in Equation 3 will also be set to zero, since no range reduction is necessary in this case.

The truncated Taylor series is computed using a Horner scheme in order to minimize resources. Sequences of a multiply/add are mapped to a single floating-point DSP block. The final value for $\log(m)$ is obtained by subtracting $\log(r_{m_{\text{top}}})$ in floating-point from the Horner evaluation output. The final result is obtained by summing $E\log(2)$ to this value.

VIII. Results

Table II shows the synthesis results for a single-precision natural logarithm implementation on an Arria10 device. The results are obtained for an architecture which uses a table to obtain $E\log(2)$, as opposed to a constant multiplier and normalization unit. This solution was preferred in order to reduce the logic footprint of the architecture at the expense of a memory block. We present various flavours of the same architecture by varying the pipeline depth of the used floating-point operators: adder/subtracter (a) and multiply-add (ma). A default value of 4 cycles for both operations is used for attaining maximum performance for a total latency of 29 cycles; reducing the latency of the adder to 3 cycles reduces the total latency to 26 cycles at the expense of a lower frequency; subsequent reduction in the latencies of these basic operators reduce overall latency at the expense of a lower frequency.

For comparison purposes the table also presents the resource requirement of a state-of-the-art natural logarithm implementation based on piecewise polynomial approximation (PA) available in Altera DSP Builder [6], and with an iterative implementation [7] available in the open source FloPoCo tool [8].

The proposed unit provides an alternative solution by trading in ALMs for embedded embedded resources (DSPs and M20Ks) compared to the piecewise polynomial approximation implementation. The trade-off is likely to be beneficial since new devices are now featuring many thousands of such embedded blocks. Additionally, the lower logic footprint results in fewer routing wires which relaxes the routing difficulty and may potentially lead to increased frequencies in chip-filling designs.

The frequency of our proposed unit varies with the latency of the used floating-point operators and during compilation Quartus-II warns about these timing numbers being preliminary. It is expected however that as the tools mature the proposed unit will show similar high frequencies in stand-alone IP benchmarking mode, and will better perform in chip-filling design.

Table II
Natural logarithm resources on A10, speedgrade I1. Iterative is reported for Virtex4 (ALMs number represents Slices, DSP number represents DSP48s and M20K number represents RAMB16.

Arch.	Lat.	Freq.	ALMs	LAB	DSPs	FPDSP	M20K
ours (a4 ma4)	29	457	262	57	2	6	3
ours (a3 ma4)	26	326	257	49	2	6	3
ours (a2 ma4)	23	223	263	50	2	6	3
ours (a2 ma3)	20	231	248	50	2	6	3
ours (a1 ma2)	14	187	210	43	2	6	3
PA	21	489	571	79	3	0	3
iterative	17	250	601[1]		5[2]	0	3[3]

Finally, a closer investigation into the resource breakdown of our proposed architecture shows that the vast majority of non-DSP and block-memory resources goes into synchronization paths: either registers or MLAB-based delay chains. It may therefore be beneficial to have a better awareness of these values when designing the architecture in order to reduce the number of parallel paths.

The lower latency implementations, since running at less than half the maximum frequency of the DSP block may use time-division multiplexing techniques for pumping data through these blocks at twice the clock speed. This is expected to reduce the DSP usage by 2 blocks.

Finally, the OpenCL standard also allows for "native" elementary function implementations which have much lower accuracy requirements. Our implementation could use a "simplified" range reduction which only uses 1 DSP and our Taylor polynomial may also be more aggressively truncated.

IX. Conclusion

In this article we have presented an architecture for the single-precision natural logarithm targeted at FPGA architectures with hardware support for floating-point addition and multiplication, such as the Arria10 FPGA. The presented implementation makes efficient use of the entire mix of resources including the DSP block in fixed-point arithmetic mode, memory blocks, logic but also the floating-point addition and multiplication. The accuracy of the proposed implementation is studied using the worst case cancellation values, and is shown to be accurate to 3*ulp*, which is the bound required for OpenCL conformance [9]. Compared to state-of-the art implementations targeting FPGAs with no silicon floating-point support, the proposed implementation offers a different resource tradeoff, using less logic but more embedded bocks. Although timing values for the Arria10 device are preliminary and the current values are lower than expected, the critical paths of this architecture show that high frequencies (+400MHz) can be expected for this core at comparable latencies. High-level design tools will need to make efficient use of the available architectures, in order to balance resource consumption.

References

[1] 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.
[2] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–58, 29 2008.
[3] J. Harrison, "A machine-checked theory of floating point arithmetic," in *Theorem Proving in Higher Order Logics*, 1999, pp. 113–130.
[4] *Arria10 Device Overview*, 2014, http://www.altera.com/literature/hb/arria-10/a10_overview.pdf.
[5] K. Chapman, "Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner)," *EDN magazine*, May 1994.
[6] "DSP Builder – Advanced blockset with timing-driven Simulink synthesis," 2011, http://www.altera.com/products/software/products/dsp/adsp-builder.html.
[7] F. de Dinechin, "A flexible floating-point logarithm for reconfigurable computers," Tech. Rep., 2010. [Online]. Available: http://prunel.ccsd.cnrs.fr/ensl-00506122/
[8] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design and Test*, 2011.
[9] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 8 December 2008. [Online]. Available: http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf