Single Precision Logarithm and Exponential Architectures for Hard Floating-Point Enabled FPGAs

Martin Langhammer, Intel Programmable Solutions Group, UK
Bogdan Pasca, Intel Programmable Solutions Group, France

Abstract—In this article we present a novel method for implementing floating point (FP) elementary functions using the new FP single precision addition and multiplication features of the Arria 10 and Stratix 10 DSP Block architecture. Our application examples are $\log(x)$ and $\exp(x)$, two of the most commonly required functions for emerging datacenter and computing FPGA targets. We explain why the combination of new FPGA technology, and at the same time, a massive increase in computing performance requirement, fuels the need for this work. We show a comprehensive error analysis, and discuss various implementation trade-offs that demonstrate that the hard FP (HFP) Blocks, in conjunction with the traditional flexibility and connectivity of the FPGA, can provide a robust and high performance solution. The architectures presented in this work meet OpenCL accuracy requirements. Our methods map extensively to embedded structures, and therefore result in significant reduction in logic resources and routing stress compared to current methods. The methods allow leveraging the routing architectures introduced in the Stratix 10 device which results in high-function performance.

Index Terms—natural logarithm, exponential, floating-point, single-precision, FPGA, FP DSP Block, Arria 10, Stratix 10

1 Introduction

The motivation for this work is twofold. The HFP DSP FPGA features enable new levels of utility, especially when combined with the flexibility in connectivity; at the same time, new FPGA applications such as those used in datacenters require very different functional capability and density than traditional FPGA use models. The combination of greatly enhanced FPGA architectures, combined with a vital need for higher performance density of new applications, creates a strong incentive for this work.

Many of the new datacenter uses, such as web search and data analytics, require $\log(x)$ and $\exp(x)$. A recent publication by Microsoft Research [1] stated that the most commonly used functions were $\log(x)$, $\exp(x)$ and $\text{div}(x,y)$. Scientific computing, including SPICE electronic simulations [2], and recent work at the CERN Large Hadron Collider [3], [4], [5] indicate that elementary functions are used extensively, and simulation time is strongly related to their performance. Therefore, one use case for $\exp(x)$ and $\log(x)$ when accelerating these applications involves using many parallel and latency-sensitive instances.

The benefits of this work are many-fold: firstly, the reduced soft logic requirements will allow for increased functional density (absolute number of functions); secondly, the mapping of the arithmetic portions of the functions to almost entirely embedded features will reduce soft logic placement issues, thereby increasing the computational density (the ability for a large number of functions to be fit at a certain clock frequency); thirdly, the DSP-based critical path allows for reaching high-frequencies in new Stratix 10 devices for generally lower latencies.

GPU performance is often specified in terms of GFLOPs, or now TFLOPs, of multiply-accumulates (MACs). GPUs often have dedicated function accelerator blocks [6]; CPUs occasionally have function accelerators, but most elementary functions are typically implemented using proprietary Intel MKL libraries [7] or open source libraries. Work by Markstein [8] detailed targeting modern CPUs, and earlier works by Cody and Waite [9] showed generic algorithm construction. The advantage of an FPGA over a processor type architecture is that if we can find a method to efficiently map a required number of elementary functions for a particular application to uniformly distributed FP features - in this case the HFP DSP Blocks - the sustained performance can approach the peak performance of the device.

Elementary function design for contemporary FPGAs is a complex task because of 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 vary by changing the approximation polynomial degree.

In contrast, microprocessor implementations of elementary functions execute more of the computation using floating-point (FP) arithmetic. FP 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 (this may extend to degree 20 or 30), including rational polynomial approximations, Taylor expansions, digit-recurrence methods, and quadratic convergence methods. As an added complication, the arithmetic used internally is often higher precision than the target precision of the function; for instance double precision elementary functions use double-double and triple-double arithmetic [10]. This experience is of great concern for FPGA, as multi-precision use would require either multiple cycle arithmetic, which would disrupt the pipeline advantage of a hardware implementation, or need a very expensive multiple operator construction. One of our goals is to find an effective method that works completely in the native input/output precision of the function.

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 potential sharing difficult. Within branches, the high-degree polynomial evaluation requires 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 low-degree polynomials are used to provide an equivalent approximation quality [11]. The coefficients for these low-degree polynomials are stored in memory blocks, available in thousands on modern FPGA devices. Other bit-level manipulation techniques, costly in microprocessors but virtually free in FPGAs, can be used during the range reduction and reconstruction stages.

This work is organized as follows. Section 2 provides a background on the FP representation, the Arria 10 and Stratix 10 FPGA architectures and the accuracy context of this work, given by the OpenCL standard. Section 3 reviews other published $\log(x)$ and $\exp(x)$ works to give a context to the improvements made in this work. Section 4 describes algorithms for mapping $\log(x)$ into hardware, describing an improved range reduction method. Sections 4.1 and 4.1.2 present the implementation approach together with a comprehensive error analysis. Section 5 introduces the exponential function and gives basic implementation strategies. Further sections 5.1 and 5.2 present a two level argument reduction and discuss the implementation strategies for modern FPGA devices. A discussion of the synthesis results is provided in Section 6.

2 Background

2.1 Floating-Point

Let x be the FP 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 FP arithmetic [12] uses a normalized mantissa with $M \in [1,2)$, except for the subnormal range. 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 FP number. The number of bits used to represent the exponent and fraction give the different FP formats of standard. In this work we focus on the single-precision (binary32) format, having an 8-bit exponent and 23-bit fraction. When dealing with FP arithmetic a useful tool when managing errors is the notion of *ulp*, which is defined by Harrison [13] as the distance between two adjacent FP numbers.

2.2 Arria 10 and Stratix 10 FPGAs

The base platform for our work is the Arria 10 FPGA with HFP DSP Blocks supporting IEEE-754 single precision [14]. This work also extends to the new Stratix 10 device, having HFP features similar to those present in the Arria 10 FPGA. The relevant hardware features for this work are:

- the basic logic-element in Arria 10 and Stratix 10 devices is the ALM (Adaptive Logic Module). ALMs are grouped by 10 in a LAB (Logic Array Block). The ALMs within a LAB have extensive interconnect capabilities, but the LAB has a reduced number of Input/Output connections.
- both Arria 10 and Stratix 10 contain M20K memory blocks which can be configured in 512x40-bits or 1024x20-bits modes;

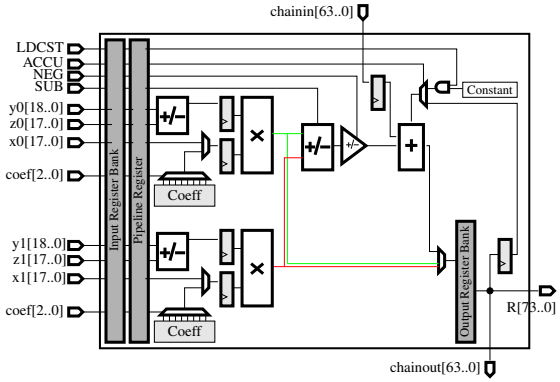


Figure 1. Arria 10 DSP Block in dual 19x18-bit fixed-point multiplier mode

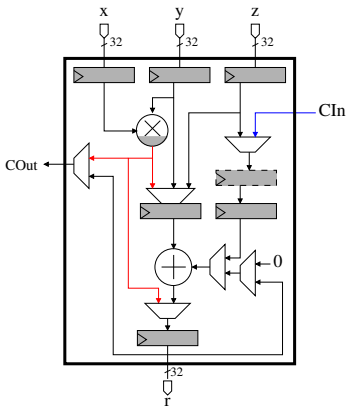


Figure 2. DSP Block in FP configuration

- DSP Blocks in both devices can be configured in fixed-point mode to execute one 27x27-bit multiplication, 2 independent 18x19 multiplications (Figure 1) or one sum-of-two 18x19-bit multiplications;
- the same DSP Blocks which can be configured in FP mode to execute one single-precision addition, multiplication, accumulation, one multiply-add operation or one multiply-accumulate operation (Figure 2).

The configurable latency is not detailed in this block diagram. Up to four pipeline stages are available in FP mode: an input stage, a FP multiplier pipeline stage, a FP adder input stage, and an output stage. Each pipeline stage can be optionally bypassed. For maximum multiply-add performance, all four stages are employed. In the case of a standalone FP multiplier, up to three stages can be used. Likewise, three or four stages can be used for a standalone FP adder. Although only two registers directly connect to it, using the DSP Block input stage will remove the additional combinatorial path from the input of the DSP Block to where the FP adder input register is physically located inside the block.

The DSP Block is designed to approximately match the maximum performance of the FPGA. However, it is known that the system level performance of a typical large design may be considerably lower than this. For example, although a 20nm planar FPGA may have a maximum performance in the 500MHz range, it is not uncommon for complex implementations to achieve only half of that.

The configurable latency of the DSP Block will allow us to

adjust the latency and performance of the DSP Block downwards to match the current design performance. As our goal is to map, as much as possible, this function to only embedded features, we are able to gain the benefit of latency reduction through a planned performance drop, while not affecting the way that multiple functions will fit into the device.

Performance drop in chip-filling designs often happens even with correct functional pipelining. A few reasons are: the lack of registers in the proximity of a short path between logic levels (the pipeline register is further away from each logic level, than the logic levels are one from another), far placement of adjacent logic, and the IO Gap crossing.

In order to address this global issue and enhance system-level performance, Stratix 10 introduces a novel routing architecture, HyperFlex, which embeds pipeline registers in the routing fabric. Using a re-timing algorithm, pipeline registers which otherwise could be placed in sub-optimal positions possibly causing critical path failures, can now be placed on the routing paths directly. The new challenge for Stratix 10 devices is adding a sufficient number of pipeline registers, especially since target frequencies are now much higher.

2.3 OpenCL Conformance

Recently, OpenCL has become a supported design flow for the FPGA industry [15], [16]. The promise of efficient implementation with a heavily abstracted design entry may make FPGAs accessible to programmers, who are not typically well versed in hardware design. OpenCL conformance requires a large, clearly defined set of vectors (approximately 4×10^9 individual tests) for each function. Each function has a maximum error allowed, which in the case of single precision $\log(x)$ and $\exp(x)$ is $3ulp$. For an embedded context, including FPGAs, an extra $1ulp$ of error is permitted. Correct rounding is not required for elementary functions, including $\log(x)$ and $\exp(x)$. Additionally, for single-precision the OpenCL standard allows flushing subnormals to zero. Consequently, the architectures described in this article also flush subnormals to zero on input and output.

3 Previous Work

The problem of FP $\log(x)$ has been well studied for FPGA targets, although most of the published methods are not suitable for the new generation of computing applications, because of area, performance, latency, and accuracy.

The performance limitations of software (SW) elementary function have been recognized; in comparison to hardware, table sizes for the exact portion of the functional decomposition are usually small (for cache management reasons), necessitating many term power series for the approximations portions of the calculation. The motivation for a fast single precision FP $\log(x)$ was described in [17], which stated that an improvement was needed over the software library example of a 94 cycle optimized $\log(x)$ [18] (GNU *glibc* requires 196 cycles for the same function [19]). A speedup of 6-8 times was realized, but with the tradeoff of large tables, and an decrease of accuracy by one to two decimal digits in the returned result. This algorithm was mapped to a Virtex II by Stamatakis [20]. Although the area is competitive (621 LUTs, 932 registers, and 3 DSP48E Blocks), the accuracy is much lower than our target OpenCL compliance.

In [21], Bruce, et.al. describe a library of FP functions, including $\log(x)$. Range reduction for $\log(x)$ was performed using division, which is very expensive, with a long latency, compared to the methods described in this work. The latency of the function was not reported.

Dinechin and Detrey showed a FP logarithm in [22]. Their approach did not use a range reduction for the mantissa processing, but instead approximated the contribution of the mantissa directly using a Higher-Order Table-Based Method HOTBM [23] polynomial evaluation. A combinatorial version of their design required 830 slices (each Virtex II slice contains two 4LUTs and two registers) and 9 18x18 multipliers, with an estimated 10% area increase to support pipelined operation, which they estimated to be able to achieve 100MHz.

A recent design by Xilinx [24], uses a CORDIC datapath [25], with multiplier-based post-processing. Single precision accuracy is claimed but not proven; a figure is shown where the maximum error is 1 *ulp* over a 2K sample set. The core is large compared to the other works: a Virtex7 -3 speed grade implementation requires 4.9K LUTs and 8 DSP48Es, with a very long latency (64), and modest performance (332MHz for a single instance). In our experience, the CORDIC method, with deep pipelines of subsequent carry chains, does not allow for a high functional density, as the relative placement of the LUTs will be constrained by the carry chains. While a single instance may exhibit good performance, multiple instances will show the effects of routing stress and congestion.

There are numerous works targeting the exponential function. Early works adapted software algorithms, [26] but map poorly on FPGAs. Even with the support of HFP DSP resources, a rough resource estimation of [26] yields in ≈ 17 DSPs, tables and DSPs for the division and additional tables for the implementation, that is w/o counting any logic required for synchronization.

A single-precision implementation of the exponential in the context of log-normal random-number generators is presented in [27]. The architecture targets fixed-point arithmetic, and computes the term e^y, $y \in (-\log 2/2, -\log 2/2)$ by splitting $y = y1 + y2$ and computing both exponentials by tabulation. Few details are given in terms of bit-widths for $y1$ and $y2$, and accuracy is not discussed.

The current state-of-the art for openly available and academically reviewed FPGA implementations of the exponential is probably [28], with the implementation available in the open source tool FloPoCo [29]. The work builds on [27], but employs a different implementation for computing e^y, based on a piecewise polynomial approximation. Section 6 compares and contrasts results obtained from FloPoCo for both the exponential and natural logarithm, against our proposed architectures. Although neither Arria 10 nor Stratix 10 devices are supported in FloPoCo 2.5.0, we have tried our best to provide fair comparisons by using a Stratix V target (-target=StratixV) which architecturally is close to an Arria 10 target. Since pipelining fidelity will be worst than if a native Arria 10 target would be available, we have tried to obtain results comparable in frequency by asking increasingly higher frequencies from the automatic pipelining generation feature.

4 Natural Logarithm

Computing the natural logarithm for x starts with applying the logarithm properties on the FP 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:

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

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 FP 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 restating the formula with a single branch.

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

The cancellation condition now 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:

$$\log(x) = E\log(2) + \log(m) \tag{1}$$

where

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

and

$$m = \begin{cases} 1.f & 1.f < \sqrt{2}, \\ \frac{1.f}{2} & 1.f \geq \sqrt{2} \end{cases} \tag{3}$$

4.1 Implementation

The first term in Equation 1 ($E\log(2)$) can be obtained directly in FP by tabulation. The 8-bit exponent is used to address a table (in the case of an Arria 10 device, a single M20K configured in 256x40-bit mode) which stores all the possible values of the product. By construction, this term will have a $1/2ulp$ error bound. Alternatively, if memory blocks are scarce, a floating-point constant multiplier [30] can be used.

Figure 3 depicts the values of the second term in Equation 1, $\log(m)$ for $m \in \left[\frac{\sqrt{2}}{2}, \sqrt{2}\right)$ (or $\approx [0.7, 1.41)$ in decimal). Calculating over this small, gently increasing monotonic section, as a consequence of the rewrite in Equation 1 eliminates the chance of cancellation errors. The value of $\log(m)$ may be computed using a Taylor series expanded in 1, for $\log(1+y)$ with y sufficiently close to 0, or m is sufficiently close to 1.

The Taylor expansion used has the form:

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

Implementation accuracy depends on the approximating polynomial degree: the higher the polynomial degree, the better the accuracy. Additionally, the approximation interval size also impacts accuracy: the smaller the interval, the higher the accuracy. An efficient FPGA implementation uses a truncated Taylor series with a low number of terms, in order to keep DSP resource usage low. Consequently, in order to obtain a sufficiently high accuracy, the approximation interval size needs to be narrowed.

For a given input and output precision p (for single precision $p = 1+24$) and a small interval $|y| < 2^{-y_{\max}}$, the higher order terms

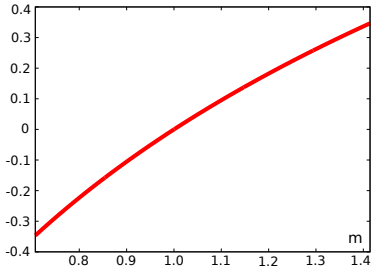


Figure 3. $\log(m)$ Approximation Range

with contributions smaller than the *ulp* of the maximum magnitude term can be dropped. In other words in the expansion

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

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 $m \in \left[\frac{\sqrt{2}}{2}, \sqrt{2}\right)$ translates into $y \in [-0.3, 0.41]$. Due to this wide range of y, the truncated Taylor series with sufficient accuracy would require 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 high number of FP operations to the branching required by the range reduction.

4.1.1 Range Reduction

Range reduction requires a division, but as we have seen [21] this is not efficient for hardware, in terms of both latency and area:

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

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

Ideally, we will try to replace the division with a multiplicative inverse. 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 here as $r_{m_{\text{top}}}$ can easily be computed via tabulation. Next, the value of y is simply obtained:

$$y = m \cdot r_{m_{\text{top}}} - 1. \tag{4}$$

The term $\log(m)$ is therefore computed as:

$$\log(m) = \log(1+y) - \log(r_{m_{\text{top}}}). \tag{5}$$

4.1.2 Accuracy for mapping to HFP resources

When computing y we can either use FP or fixed-point arithmetic. Properly scaled and bounded, the mantissa portion calculation in fixed-point arithmetic can achieve single precision accuracy, but the final summation requires significant FPGA resources due to alignment shifters.

Our logarithm implementation computes final result as the sum of three FP terms, $\widetilde{A}$, $\widetilde{B}$ and $\widetilde{C}$:

$$\log(x) = \underbrace{E\log(2)}_{\widetilde{A}} + \underbrace{(\log(1+y)}_{\widetilde{B}} \underbrace{- \log(r_{m_{\text{top}}}))}_{\widetilde{C}} \tag{6}$$

Let us assume that the three terms are computed in FP 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*.

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

We use the notation in [31], and do not distinguish between δs.

$$\begin{aligned} R &= (A(1+\delta)+(B(1+\delta)-C(1+\delta))(1+\delta))(1+\delta) \\ &= A(1+\delta)^2+B(1+\delta)^3-C(1+\delta)^3 \\ &= A(1+\theta_2)+B(1+\theta_3)-C(1+\theta_3) \\ &\leq (1+\theta_3)(A+B-C) \end{aligned}$$

Here $|\theta_n| \leq \frac{nu}{1-nu}$ with $u = ulp/2$. This formula suggests that if $A+B-C$ will have a magnitude close to $\max(A,B,-C)$ then the relative error in the final result is slightly worst than $1.5ulp$ (θ_3).

This is a simplistic error bound, and the actual value can be worst if $\max(A,B,-C) \gg A+B-C$. We present next a detailed analysis of the possible cases:

- if $A = 0$ then the accuracy is returned by the term $B-C$. We distinguish two sub-cases, depending on m:
  - for $m \in [1-2^{-9}, 1+2^{-9}]$, no range-reduction is required; y is exact and the final accuracy depends on the accuracy of B.
  - for $m \in [\sqrt{2}/2, 1-2^{-9}] \cup [1+2^{-9}, \sqrt{2}]$, range-reduction is required. y is calculated using Equation 4. For $m \geq 1+2^{-9}$ no cancellation can occur between B and C. This can be seen in Figure 4. The relative error can be written as $R = (B-C)(1+\theta_2)$ making the final accuracy be slightly lower than $1ulp$. For $m \leq 1-2^{-9}$ a 1 bit cancellation can occur. The final relative error is double the relative error of C, which is accurate to 1/2 *ulp* since it is tabulated.
- if $A \neq 0$ then A will be the dominant term. A 1-bit cancellation can happen when $E = -1$ and $m > 1$, or $E = 1$ and $m < 1$. In this cases the final accuracy is two times lower than the accuracy of A which is computed accurately to 1/2 *ulp*.

4.1.3 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 $1.f > \sqrt{2}$ (Equation 3), which is also our branch condition. Therefore, the table address will hold the branch bit and the top 9 bits of f (the implied leading 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 again guaranteed to be within $1/2ulp$.

4.1.4 Computing y

The accuracy of $\log(1+y)$ will directly be influenced by the accuracy of y. If no range reduction is necessary ($1-2^{-9} < m < 1+2^{-9}$ – m close to 1), the cancellation during subtraction ($y = m-1$) will ensure that y is exact. Range-reduction is necessary for m outside this interval, and we must ensure that y is computed with sufficient accuracy.

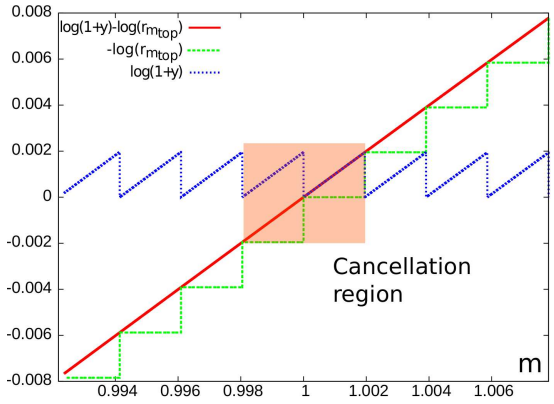


Figure 4. $\log(x)$ Cancellation Boundaries for $B-C$ in red. Contribution of B is depicted in blue and C in green.

When computing $y = m \cdot r_{m_{\text{top}}} - 1$ in FP our operands will be FP numbers. The product $m \cdot r_{m_{\text{top}}}$ is bound by construction to have the form `1.0000000001X...L`. Here L is used here to denote the LSB of the $m \cdot r_{m_{\text{top}}}$ mantissa, and the weight of the round-off error is 1/2 the weight of L. The value for y is obtained by subtracting 1 and normalizing: $y = 1.XX...L00000000 2^{-10}$. Post subtraction, the round-off error from $m \cdot r_{m_{\text{top}}}$ will be amplified by the cancellation size, and will have a magnitude of 2^{-23+10} (the position of L is now 10 positions higher in the mantissa). More formally:

$$\begin{aligned} y &= m \cdot r_{m_{\text{top}}}(1+\delta_r) - 1 \\ &= m \cdot r_{m_{\text{top}}} - 1 + m \cdot r_{m_{\text{top}}} \cdot \delta_r \end{aligned}$$

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:

$$y = m \cdot r_{m_{\text{top}}} - 1 + m \cdot r_{m_{\text{top}}} \cdot 2^{-p-1}$$

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

$$\begin{aligned} y &\approx (m \cdot r_{m_{\text{top}}} - 1)(1 + \frac{m \cdot r_{m_{\text{top}}} \cdot 2^{-p-1}}{m \cdot r_{m_{\text{top}}} - 1}) \\ &\approx (m \cdot r_{m_{\text{top}}} - 1)(1 + \frac{m \cdot r_{m_{\text{top}}} \cdot 2^{-p-1} 2^{\psi}}{2^{\psi}(m \cdot r_{m_{\text{top}}} - 1)}) \\ &\approx (m \cdot r_{m_{\text{top}}} - 1)(1 + 2^{-p-1+\psi}) \end{aligned}$$

This 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 resulting when computing y provided that no rounding error was actually performed on $m \cdot r_{m_{\text{top}}}$, 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 equation. Because of 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.

$$\log(1+x) = x - x^2/2 + x^3/3 - x^4/4 + ...$$
$$\log(1+x(1+\delta_x)) = x(1+\delta_x) - x^2(1+\delta_x)^2/2 + ...$$

This error will be the final error when both terms A and C are zero in our equation. This only happens when $m \in [1-2^9, 1+2^{-9}]$, in which case no range-reduction is necessary. The 2^{ψ} *ulp* error also

$$
\begin{aligned}
m \cdot r_{m_{\text{top}}} &= \texttt{1.000000000XXXXYYYYYYYYYYYYY} \\
j &= \texttt{1.000000000XXXX} \quad + \\
i &= \texttt{1YYYYYYYYYYYYY} - \\
k &= \texttt{1}
\end{aligned}
$$

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

occurs for inputs roughly outside this interval, and is shown 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 $r_{m_{\text{top}}}$ is not sufficient having this ψ-bit cancellation. Storing more accuracy in $r_{m_{\text{top}}}$ is impossible using the single precision format (which was chosen to match the HFP DSP Blocks). The requested error for $r_{m_{\text{top}}}$ needs to be smaller than $2^{-p-1-\psi}$ provided that we can produce the $m \cdot r_{m_{\text{top}}}$ product without rounding errors.

4.1.5 Improved accuracy range reduction

We store $r_{m_{\text{top}}}$ in fixed-point on a precision $1+p+\psi$ bits. The multiplication $m \cdot r_{m_{\text{top}}}$ 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 $m \cdot r_{m_{\text{top}}} - 1$ 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 FP.

We now present a novel way of performing this calculation efficiently using the available HFP DSP Blocks; the purpose of this stage is to obtain an accurate FP value y which can then feed into the Taylor polynomial evaluation stage.

We accomplish the translation between the fixed-point product $m \cdot r_{m_{\text{top}}}$, via the difference $m \cdot r_{m_{\text{top}}} - 1$ into FP by exploiting the format of this product: leading '1' followed by a number of zeros followed by information bits.

```
m*rmtop = 1.000000000XXXXYYYYYYYYYYYYY
```

The idea is to represent the fixed-point product $m \cdot r_{m_{\text{top}}}$ as a sum of FP numbers having a small overlap: $m \cdot r_{m_{\text{top}}} = j + i - k$. Once accomplished, the difference $m \cdot r_{m_{\text{top}}} - 1$ can be performed using FP arithmetic only.

A very efficient way of obtaining FP values from the fixed-point product $m \cdot r_{m_{\text{top}}}$ 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 FP value.

Figure 5 shows the fixed-point alignments of the mantissas of the 3 FP values created against the fixed-point product $m \cdot r_{m_{\text{top}}}$.

The full computation of y further requires subtracting the FP value '1'. The required order of operations is:

$$
\begin{aligned}
y &= m \cdot r_{m_{\text{top}}} - 1 \\
&= (j + i - k) - 1 \\
&= (j-1) - k + i
\end{aligned} \tag{7}
$$

One of the 3 FP operations in Equation 7 can be saved by restructuring the calculation as follows:

$$
y = (j - (1+k)) + i \tag{8}
$$

and observing that the weight difference between '1' and k, which are both constants, allows packing them into one single FP mantissa, without actually preforming a FP 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 so to match the characteristics of the embedded memory blocks.

4.2 Architecture

The architecture of the single precision natural logarithm is depicted in Figure 6. For simplicity, the FP input x is split into its 3 components, the sign, the exponent and the fraction. The branch condition ($1.f > \sqrt{2}$) is replaced by $1.f > 1.5$ for simplicity, which only uses 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 2).

The m argument (Equation 3) is calculated by the central multiplexer by selecting between 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 with an extended precision. A fixed-point multiplier then allows us to obtain ($mr_{m_{\text{top}}}$). Next, the FP subtracter and adder perform Equation 8 in order to obtain y. Data is fed to these units through a network of multiplexers which allow combining 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 6 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 pairs are mapped to individual FP DSP Blocks. The final value for $\log(m)$ is obtained by subtracting $\log(r_{m_{\text{top}}})$ in FP from the Horner evaluation output. The final result is obtained by summing $E\log(2)$ to this value.

By inspection, the Taylor series maps to three chained HFP DSP Blocks, with the FP adder in the last block also adding in $\log(r_{m_{\text{top}}})$. Three additional HFP DSP Blocks configured in FP adder-only mode are used in the circuit in order to prepare the input argument for the Taylor series, and to implement the final summation. Two more DSP Blocks, configured in fixed-point, implement a 36x34-bit fixed-point multiplier to provide the multiplicative inverse function required by the range reduction operation. Synthesis results for the natural logarithm functions are presented in Section 6.

5 Exponential

The floating-point exponential function is defined on a limited input range. In single-precision the maximum input value for which

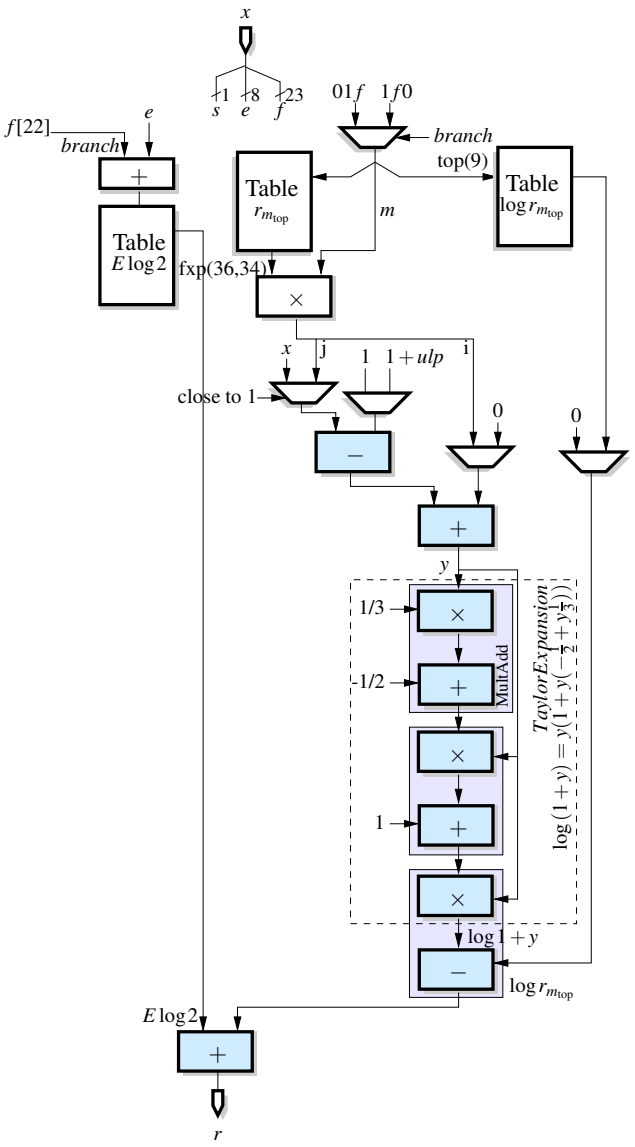


Figure 6. Architecture for the single-precision natural logarithm

the exponential has a representable output is $\lceil \log(2^{127}(2-2^{-23})) \rceil$ or approximately 88.72 in decimal, and the minimum one is $\lfloor \log(2^{-126}) \rfloor$ or -87.33 in decimal. Therefore, the input range for the exponential is defined on the interval $[-87.33, 88.72]$. Smaller values outside this interval result in an underflow (return 0), while larger values outside the interval will overflow (return $+\infty$).

The exponential of a floating-point value $x = (-1)^s 2^e 1.f$ is computed using a rewrite of x that has the following property:

$$x = E log(2) + y \tag{9}$$

E is an integer and $y \in (-\log(2)/2, +\log(2)/2)$. The rewrite in Equation 9 reduces the range on which the exponential is computed to the range of y, and allows for a straightforward implementation:

$$\begin{aligned} e^x &= e^{E\log(2)+y} \\ &= e^{\log(2^E)} e^y \\ &= 2^E e^y \end{aligned} \tag{10}$$

The values E and e^y in Equation 10 are close to the exponent and mantissa of the result. The range of $y \in (-\log(2)/2, +\log(2)/2)$, implies that e^y belongs to $(1/\sqrt{2}, \sqrt{2})$. The result mantissa needs to be normalized when $e^y < 1$, which involves a potential 1 position shift and exponent update (Equation 11).

$$e^x = \begin{cases} 2^E e^y & \text{if } y \in [0, +\log(2)/2) \\ 2^{E-1} 2e^y & \text{if } y \in (-\log(2)/2, 0) \end{cases} \tag{11}$$

The integer E is defined as the value for which the product $E\log(2)$ is closest to x. Therefore E is found by dividing x by $\log(2)$ and then rounding this value to the nearest integer value

$$E = \left\lfloor \frac{x}{\log(2)} \right\rceil. \tag{12}$$

Having found E, y is obtained using Equation 13, and by construction $y \in (-\log(2)/2, \log(2)/2)$

$$y = x - E\log(2). \tag{13}$$

5.1 Mapping E and y compute to the FPGA

5.1.1 Known FPGA implementation techniques

Computing y in Equation 13 requires computing E and the intermediary value K defined in Equation 14.

$$K = E\log(2). \tag{14}$$

The reduced range of x for single precision (approximately $[-87.33, 88.72]$) allows casting x into fixed-point x_{fxp}. A fixed-point constant-multiplier by the constant $C_{1/\log(2)} = 1/\log(2)$ (best implemented using the KCM technique [32]) is used to obtain $\widetilde{E} = x_{fxp} \cdot C_{1/\log(2)}$. The value E is obtained by rounding the fixed-point value $\widetilde{E}$ to the nearest integer, which requires an adder.

$$E = \lfloor \widetilde{E} \rceil$$

The intermediary variable K in Equation 14 is computed using another constant multiplier by $C_{\log(2)} = \log(2)$. Finally, y is found by performing a fixed-point subtraction (Equation 13).

The difference y may be the result of a large cancellation when the two terms have close values. In order to avoid a massive loss in accuracy caused the cancellation, term K needs to be computed on roughly twice the precision of x.

These implementation techniques are generic and lead to costly implementations. From a fixed-point perspective these can be improved, and some of these optimizations are presented in the next section. From a general mapping perspective, these methods have been developed having fixed-point FPGAs in mind, and make no use of FP arithmetic. Consequently, we will introduce the required adaptations for taking advantage of the HFP features, all in an optimized context.

5.1.2 Proposed FPGA-specific optimizations

A number of possible optimizations will be presented in this section. Some of these are included in the architectures benchmarked in the Results section. The correspondence between optimizations and benchmarked architectures will be highlighted as required.

Computing E requires a constant multiplier. The area and latency of this constant-multiplier depends on the input size (x_{fxp}) which for single-precision is 33 bits (size given by the magnitude of x_{fxp} and the fraction bits kept for when $x_{fxp} \cdot C_{1/\log(2)}$ can be different than 0). One solution to reduce the size and latency of the constant multiplier is to reduce the precision of its input. By using 8 magnitude bits (including the sign) + 1 fractional bit of the fixed-point x_{fxp} that we denote by x_{fxpRed}, the area and latency of the

constant multiplier is considerably reduced. The reduced latency allows starting the computation of K (Equation 14) sooner, and overall reduces path-balancing register requirements. Note that in general x_{fxp} may still required for computing y (Equation 13). However, in none of our proposed architectures x_{fxp} actually needs computing.

Using a less accurate x_{fxpRed} implies that the new value:

$$\widetilde{E'} = x_{fxpRed} \cdot C_{1/\log(2)}$$

will be less accurate than $\widetilde{E}$. Since x_{fxpRed} is obtained from x_{fxp} by truncating at position 2^{-1} (include 1 fractional bit) $|x_{fxp} - x_{fxpRed}| < 2^{-1}$ and therefore $|\widetilde{E} - \widetilde{E'}| < C_{1/\log(2)} 2^{-1} < 0.722$. With $E = \lfloor \widetilde{E} \rceil$ and $E' = \lfloor \widetilde{E'} \rceil$, the difference $|E - E'| \leq 1$ with $E, E' \in \mathbb{Z}$.

By reducing the precision in x_{fxpRed} the newly obtained E' is no longer the integer value for which the $E'\log(2)$ is the closest to x, but can be one unit off: $E' \in \{E-1, E, E+1\}$. Consequently, $y' = X - E'\log(2)$ belongs to the wider interval $[-3\log(2)/2, 3\log(2)/2]$. For this input domain the function $e^{y'}$ will produce a result in the interval $(1/2\sqrt{2}, 2\sqrt{2})$ or $\approx (0.35, 2.83)$. The wider output interval for $e^{y'}$ requires computing $e^{y'}$ with one more bit of accuracy and a slightly more elaborate normalization stage, but which still fits in one table lookup (LUT6):

$$e^x = \begin{cases} 2^{E'+1}\frac{e^{y'}}{2} & \text{if } e^{y'} \in [2, 2\sqrt{2}) \\ 2^{E'} e^{y'} & \text{if } e^{y'} \in [1, 2) \\ 2^{E-1} 2 e^{y} & \text{if } e^{y'} \in [1/2, 1) \\ 2^{E-2} 4 e^{y} & \text{if } e^{y'} \in (1/2\sqrt{2}, 1/2) \end{cases} \quad (15)$$

The function which computes E' inputs the 9 bits of x_{fxpRed}: 1 (sign) + 7 (magnitude) + 1 (fraction), and outputs 9 bits. The function that computes $K' = E'\log(2)$ inputs the 9 bits of E'. Since computing K' ultimately depends only on the initial 9 bits of x_{fxpRed}, the two computations can be fused in order to reduce latency. Obtaining K' using table-based approach significantly reduces latency of computing y', required for unlocking the more costly calculation $e^{y'}$. The value of E', still required for the final value of the exponent, may be obtained either using a constant multiplier or a similar table. Nonetheless, the computation of E' is now outside the critical path. The exponential architectures Arch_1 and Arch_2 in Table 1 make use of this architectural optimization.

One calculation which can still be improved in terms of latency is obtaining the fixed-point x_{fxpRed} by first obtaining x_{fxp}. The 33-bit wide barrel shifter requires multiple levels, thus having a long latency. One alternative which reduces latency at the expense of resources is using a smaller barrel shifter which only inputs the bits of x which can appear in x_{fxpRed}. The exponential architectures Arch_1 and Arch_2 in Table 1 make use of a small barrel shifter of obtaining the reduced-precision x_{fxpRed}. The difference between these architectures is highlighted in Section 5.2.2.

Another alternative is to obtain x_{fxpRed} via tabulation. The format of x_{fxpRed} includes 7 magnitude bits + 1 fractional bit, and the sign bit (8+1 bits in total). In order to obtain these bits via tabulation we need to input the following information: the top 7 bits fraction bits of x, the sign bit, exponent bits that describe the valid alignments (explained below). Figure 7 shows all the valid alignments of x in fixed-point that are used for x_{fxpRed}. In red we highlight the implicit one, and the light blue we show the top 7 fraction bits of x in all valid alignments. In pink we highlight the exponent bits which input the table. The exponent overflow case is captured separately, so a total of 1+3+7=11 bits are required for driving the x_{fxpRed} table.

																		Exp	Exponent (bin, biased)
0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	1	1	0	-1	0 1 1 1 1 1 1 0
0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	1	0	1	0	1 1 1 1 1 1 1 1
0	0	0	0	0	0	0	1	1	0	1	0	1	1	1	0	1	0	0	1 0 0 0 0 0 0 0
0	0	0	0	0	0	1	1	0	1	0	1	1	1	0	1	0	0	0	1 0 0 0 0 0 0 1
0	0	0	0	0	1	1	0	1	1	1	1	1	0	1	0	0	0	0	1 0 0 0 0 0 1 0
0	0	0	0	1	1	0	1	1	0	1	1	0	1	0	0	0	0	0	1 0 0 0 0 0 1 1
0	0	0	1	1	0	1	1	0	1	1	0	1	0	0	0	0	0	0	1 0 0 0 0 1 0 0
0	0	1	1	0	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0 0 0 0 0 1 0 1
0	1	1	0	1	1	0	1	1	1	Overflow or underflow								> 6	

Figure 7. Allowed alignments for x_{fxpRed}

The tabulation of x_{fxpRed} maps into one M20K block in $2^{11} \times 10$-bit configuration. The 3 exponent bits will only allow shifting in valid positions, and will never shift out the input. When the exponent of $x_{exp} \leq -2$, $E' = 0 \rightarrow K' = 0$ and therefore $y' = x_{fxp}$. This is accomplished by selecting x_{fxp} instead of y' in computing $e^{y'}$ (or equivalently masking K'). We have not applied this optimization alone to the exponential architecture as we did not believe that the 2 cycle memory block would significantly improve latency over a small barrel shifter.

Since x_{fxpRed} is the input to the stage computing K' also by tabulation we can fuse the two tables together such that $K' = func(x_{fxpRed})$. Fusing the tables will reduce the latency to the latency of an M20K, at the expense of approximately doubling the memory count. The exponential architecture Arch_3 in Table 1 highlights the resource tradeoffs of this architectural choice.

The value y' can be computed in fixed-point:

$$y'_{fxp} = x_{fxp} - K'. \quad (16)$$

The fixed-point difference may be subject to a cancellation, when both terms have close values. This requires computing K' on a higher precision, so that post subtraction, the number of significant bits in the difference is sufficient to compute $e^{y'}$ sufficiently accurate.

As opposed to current state-of-the-art implementations using a fixed-point y', our goal is to obtain y' in FP, and make use of the HFP blocks in evaluating $e^{y'}$. Obtaining y' in FP can be done casting the fixed-point value y' to FP. This requires a leading digit counter, absolute value calculation, left shifter, and possibly rounding and is in general costly both in terms of resources and latency. We have not explored this implementation further as resource requirements seemed significant.

Alternatively we focus on obtaining K'_{fp} directly in FP, and use HFP-supported arithmetic in order to obtain y'_{fp}. Similar to obtaining y'_{fxp}, a cancellation during the subtraction can lead to a massive accuracy loss. Therefore, K'_{fp} needs to be computed with more accuracy, and will be stored as an unevaluated sum of non-overlapping FP numbers:

$$K'_{fpUneval} = K'_{fpHigh} + K'_{fpLow}.$$

Having the higher accuracy $K'_{fpUneval}$, y' is obtained with the following operations:

$$\begin{aligned} y' &= x - K'_{fpUneval} \\ &= x - (K'_{fpHigh} + K'_{fpLow}) \\ &= (x - K'_{fpHigh}) - K'_{fpLow} \end{aligned} \quad (17)$$

Whenever the exponent of x, $x_{exp} \leq -2$, $K'_{fpUneval}$ needs to be set to 0. This is done by masking the exponent and fraction bits to reset the encoding of K'_{fpHigh} and K'_{fpLow} to 0. Since the embedded floating-point arithmetic in the DSP Block flushes subnormals on

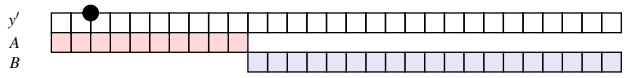


Figure 8. The argument reduction exemplified on fixed-point y'

input and output to 0, we can restrict to masking-off only the exponent field of the two values.

All 3 exponential architectures in Table 1 use a FP $K'_{fpUneval}$, and obtain y' as described in Equation 17. The Arch_1 and Arch_2 architectures index the $K'_{fpUneval}$ table with x_{fxpRed} whereas Arch_3 indexes the table with an 11-bit string composed of (x_{sign}, $x_{exp}[2:0]$, $x_{frac}[22:14]$).

The next section describes the basic range-reduction technique for computing $e^{y'}$ with y' stored in FP.

5.2 Argument reduction for $e^{y'}$

5.2.1 The basics

The exponential $e^{y'}$ with an output in $(1/2\sqrt{2}, 2\sqrt{2})$ is still too complex to compute directly. The following multiplicative argument reduction is preformed:

$$e^{y'} = e^{A+B} = e^A e^B. \tag{18}$$

The goal of this argument reduction is to get B close to 0 so that the a truncated Taylor series is sufficiently accurate to compute e^B and for e^A to be computed via tabulation.

The Taylor series for e^x, expanded in $x = 0$ is:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}. \tag{19}$$

With a truncated Taylor series up to $n = 2$ (3 terms), and $x \in [0, 2^{-8})$ the approximation error is:

$$\left| e^x - \sum_{n=0}^{2} \frac{x^n}{n!} \right| < 1.011 \cdot 2^{-27} \tag{20}$$

which is sufficiently accurate for a single-precision implementation.

5.2.2 Proposed argument-reduction implementation

As previously stated, y' may either be computed in fixed-point or floating-point. If y' is computed in fixed-point, then obtaining A reduces to fetching the most significant 10 bits (2 integer bits, sign included, and 8 fractional bits) of y'. B is also obtained by selecting bits having weights $\leq 2^{-9}$. The selection of A and B for a fixed-point y' is depicted in Figure 8. The Horner datapath may then use B for a fixed-point implementation, or requires normalizing B before feeding B_{fp} into a floating-point Horner evaluator.

Since our proposed architecture computes y' in FP, then A may be obtained either using an optimized shifter architecture, or by tabulation. If the architecture for getting A is shifter-based, then a 9-bit right shifter is sufficient. The magnitude of A is that of y', so 1 integer bit is sufficient to represent the largest value in y'. An exponent value of $y'_{exp} = -9$ will result in $A = 0$ (all bits are shifted out), with $e^A = 1$. The 1+9=10 bits used for A will then input a table that returns e^A directly in floating point. This architecture requires 2 M20Ks in configuration $1024x20$-bits. Exponential architectures Arch_1 and Arch_3 in Table 1 make use of this small right shifter for obtaining A.

								EXP	EXP(3:0)			
1	1	1	1	1	1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	0	-1	0	0	0	0
1	1	1	1	1	1	0	0	-2	0	0	0	1
1	1	1	1	1	0	0	0	-3	0	0	1	0
1	1	1	1	0	0	0	0	-4	0	0	1	1
1	1	1	0	0	0	0	0	-5	0	1	0	0
1	1	0	0	0	0	0	0	-6	0	1	0	1
1	0	0	0	0	0	0	0	-7	0	1	1	0
0	0	0	0	0	0	0	0	-8	0	1	1	1

Figure 9. The contents of the mask table used to obtain A in floating-point by masking the top 8 fraction bits of y'_{fp}

Table e^A may also be obtained directly by tabulation, by feeding the relevant exponent and fraction bits of y'. Unfortunately, this architectural choice leads to a large table size (address is 12-bits) and since e^A is computed outside the critical path, no latency saving is observed. The exponential architecture Arch_2, which is otherwise similar to Arch_1, shows the impact of using this tabulation.

The value A can easily be obtained in FP by masking certain bits of the mantissa of y'_{fp}. The LSB 4 bits of the exponent of y'_{fp}, y'_{exp} are sufficient for indexing the mask table. The mask table is 8-bits wide, and applies to the top fraction bits of y'. The contents of the mask table are presented in Figure 9.

When the exponent of y' is less than -8 then exponent bits are reset, therefore setting A to 0 which makes $B = y'_{fp}$. Having A in FP, B is obtained in FP by means of a subtraction:

$$B = y'_{fp} - A.$$

All 3 exponential architectures find B by first finding A in FP using a mask table.

Computing e^B is done using a degree 2 Taylor polynomial evaluated using the Horner's scheme.

$$e^B = 1 + B\left(1 + \frac{B}{2}\right)$$

When evaluating the Horner polynomial in FP, two multiply-add blocks, each mapping to a single HFP DSP Block are used. The total latency is 8 cycles for high performance implementation (4 cycles per block).

Once e^B is obtained, $e^{y'} = e^A \cdot e^B$ is implemented using a floating-point multiplier.

A final normalization stage is required before returning the result since $e^{y'}$ is $\in (1/2\sqrt{2}, 2\sqrt{2})$. The normalization cases are explained in Equation 15 and are implemented via trivial multiplexing.

The generic, simplified architecture of the exponential function is presented in Figure 10. The presented architecture obtains $K'_{fpUneval}$ directly using a lookup with the address from 11-bits of x (Arch_3). Alternative architectures (Arch_1 and Arch_2) use a small-size barrel shifter for x_{fpRed}. The depicted architecture also uses the generic box *Find A*. The two possibilities include using a small fixed-point shifter (Arch_1 and Arch_3), or using a table lookup driven by some bits of y' (Arch_2). As it will be presented in the Results section, the small shifter-based architecture provides a better trade-off in terms of resources.

6 Results and Discussion

Table 1 presents synthesis results for both the natural logarithm and the exponential, targeting two HFP-enabled devices: Arria 10

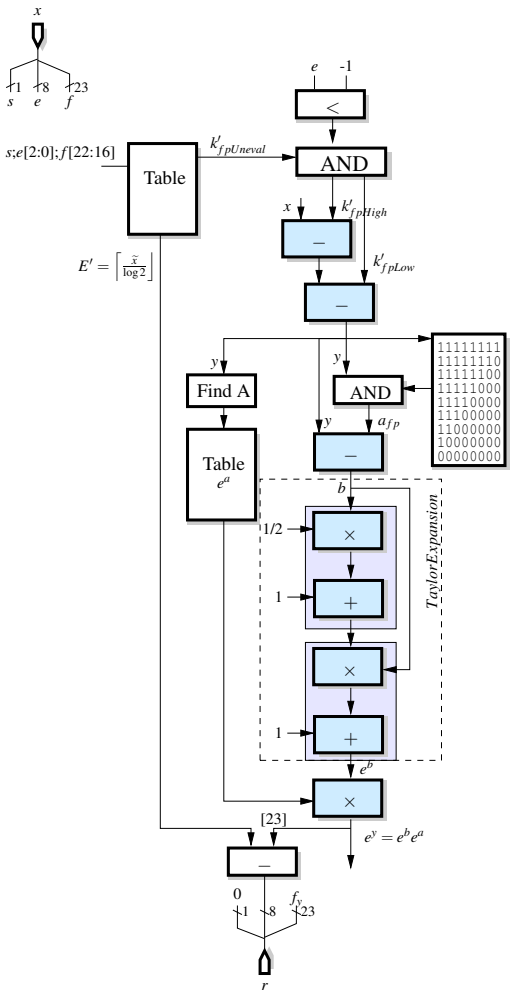


Figure 10. Architecture for the single-precision exponential

and Stratix 10[1]. On Arria 10 we show results for the fastest core device (core speedgrade 1). The results given in Table 1 target the highest achievable performance. Lower latency and logic utilization can be obtained if targeting a lower frequency.

LAB usage is of particular interest for this work. LABs are tiles of logic which regroup 10 ALMs. LABs give a better fitting density scale than counting individual ALMs since some ALMs can end-up being hard to use (only certain configurations left available due to pin count constraints) or inefficient to use (isolated from adjacent logic, usable but would break timing). Although we won't only focus on LAB count, we believe that future benchmarks should include LAB count as it gives a better core performance indicator in full designs.

For completeness we have compared our proposed cores against state-of-the art implementations available in the DSP Builder Advanced tool [33], and the open-source core generator FloPoCo. We have used Quartus Prime Pro version 16.1, and FloPoCo version 2.5.0. For the DSP Builder Advanced tool, we have generated the cores targeting the Arria 10 and Stratix 10 cores, and focused on high performance. Unfortunately FloPoCo is missing an Arria 10 target. We have used a Stratix V target, and the generated RTL was synthesized for an Arria 10, fastest core speedgrade[2]. Therefore, it is likely that the FloPoCo results are not optimal as the deeper pipeline of the DSP blocks in Arria 10 as opposed to Stratix V causes performance bottlenecks around DSP Blocks.

For the natural logarithm on Arria 10, our proposed architecture consumes both fewer ALMs, fewer LABs, but more DSPs. The latency of the DSP Builder Advanced implementation is shorter, for a similar output frequency. Compared to FloPoCo's $\log(x)$ (which we were unable to push beyond $\approx$ 225MHz due to DSP-block related critical paths) our proposed implementation consumes roughly half the ALMs and LABs at a minor increase in DSP and M20K count.

The maximum core performance of the Stratix 10 device is higher than that of Arria 10. Embedded Stratix 10 features like the DSP and memory blocks are designed to match the maximum performance. Additional innovations such as the HyperFlex architecture allow achieving maximum core performance with adequate pipelining of general-purpose logic. Therefore, for Stratix 10 benchmarking we have focused on high frequency architectures, ensured the HyperFlex routing architecture is used for all synthesized cores and tried to adequately pipeline logic.

We have compared our proposed core against the DSP Builder Advanced $\log(x)$ implementation. We have observed that both cores are capable of reaching very high frequencies: 724MHz for the proposed core, and 682MHz for the DSP Builder Advanced logarithm which confirms that our Stratix 10 setup behaves as expected. The latency of our proposed core is 9 cycles lower than the DSP Builder Advanced $\log(x)$ while also improving: LAB count – reduced to roughly half – and ALM count at the expense of some DSP Blocks. The synthesis results for $\log(x)$ on the Stratix 10 confirm the hypothesis that high-frequency instances will have both lower latency and lower resource requirements than existing state-of-the-art implementations.

For the exponential on Arria 10 we give the results for the architecture variations discussed throughout the paper. The first (Arch_1) uses a reduced-width fixed-point shifter for obtaining x_{fxpRed}, uses another reduced-width fixed-point shifter for obtaining A, and uses a table for fetching e^A. The second architecture (Arch_2) uses one table for obtaining e^A, and inputs top bits from the fraction of A together with some exponent bits. The goal of this architectural change is to reduce latency, and therefore reduce synchronization overhead which costs logic. However, as it can be observed from the left of Figure 10 depicting the architecture of the exponential function, computing e^A is outside the critical path. The large table has an overhead in M20K blocks, and does little to reduce logic. The third alternative (Arch_3) uses one table for obtaining $K'_{fpUneval}$, and the same architecture for computing e^A as the first architecture. Since computing $K'_{fpUneval}$ is on the critical path (see Figure 10), a reduction of 3 cycles in latency, combined with a 32 ALM reduction (5 LABs) is traded for extra 6 M20Ks. Choosing between the first and the third proposed architecture is therefore left to the user, and the choice will be application dependent.

For Stratix 10 synthesis results are only given for the first architecture, using Hyper-Flex registers, which is the default flow. Compared to the DSP Builder architectures, the latency of our proposed cores is longer on Arria 10, and shorter on Stratix 10 devices. Our proposed core always reduces ALM count,

1. Stratix 10 results are preliminary for both tools and silicon

2. Command-line used: ./flopoco -target=StratixV -frequency=400 FPExp 8 23

Table 1
Natural Logarithm and Exponential resources on Arria 10, fastest core speedgrade (1) and preliminary results of Stratix 10. For the exponential, the synthesis for [28] were generated using FloPoCo 2.5.0 targeting 400,500,600 and 700MHz respectively using the Stratix V target. PPA stands for architectures using a Piecewise-Polynomial Approximation technique.

Func.	Reference	Target	Architecture	Lat.	Freq.	ALMs	LAB	DSPs	FPDSP	M20K
Log	Proposed	Arria 10	Full Performance	25	482	302	45	3	6	3
	State-of-the art PPA [33]		PPA	20	481	418	72	3	0	3
	FloPoCo Log [34]		Iterative, unrolled (400MHz*)	20	222	607	98	7	0	2
			Iterative, unrolled (500MHz*)	29	223	681	126	7	0	2
			Iterative, unrolled (700MHz*)	39	225	856	175	7	0	2
	Proposed	Stratix 10	Full Performance w/o HyperFlex Registers	33	545	526	75	3	6	3
			High-performance w HyperFlex Registers	34	724	552	85	3	6	3
	State-of-the art PPA [33]		PPA with HyperFlex Registers	43	682	888	160	3	0	3
Exp	Proposed	Arria 10	Arch_1 shifter x_{fxpRed}, shifter A, table e^A	34	483	296	49	-	6	3
			Arch_2 shifter x_{fxpRed}, one table for e^A	34	483	295	51	-	6	6
			Arch_3 table for $K'_{fpUneval}$, shifter A, table e^A	31	483	264	44	-	6	9
	State-of-the art PPA [33]		PPA (MLABs)	19	452	748	112	2	0	0
			PPA (M20K)	22	411	632	128	2	0	3
	FloPoCo PPA [28]		PPA (400MHz*)	16	213	473	50	1	0	1
			PPA (500MHz*)	18	197	496	74	1	0	2
			PPA (600MHz*)	23	197	504	82	1	0	2
			PPA (700MHz*)	29	358	532	93	1	0	2
	Proposed	Stratix 10	Arch_1 w HyperFlex Registers 1	36	589	397	59	-	6	3
			Arch_1 w HyperFlex Registers 2	40	642	435	61	-	6	3
			Arch_1 w HyperFlex Registers 3	44	658	457	63	-	6	3
	State-of-the art PPA [33]		PPA (M20K)	41	561	969	147	2	0	3

sometimes by more than twice. The LAB count is improved from 128 in the case of a DSP Builder Advanced core targeting M20Ks, to 49 LABs. The differences also hold on Stratix 10, but frequencies are generally higher. Compared to the FloPoCo exponential implementation, which we struggled to get to 358MHz with deep pipelining, our architecture consumes roughly half the logic (ALMs, LABs) at the expense of some DSPs, while at the same time running at 230MHz+ more.

Overall, the proposed cores offer high-performance while generally reducing logic consumption, at the expense of DSP and M20K Blocks. With Stratix 10 devices requiring higher levels of pipelining, the proposed architectures based on HFP usage reduce overall latency while providing a higher-frequency implementation. The LAB usage, a metric that we believe better shows the behavior of a core in a full design confirms that our proposed architectures always outperform the current solutions.

7 Future Work

This required increase in performance also introduces the need for computational density benchmarking. One of the motivations for this work was to be able to fit more functions, with higher $F_{\max}$, together with other sizable designs implemented in both soft logic, as well as the multiply-accumulate and vector reductions supported by the HFP DSP features. Full device benchmarks, to give confidence in expected peak to sustained throughput ratios not only for the traditional measurement of MAC GFLOPs - but also for a new benchmark considering the saved overhead of elementary function implementation - should be considered.

Another area of interest is the scaling of this FP functional mapping to non-native precisions, such as single extended or even double precision. Multi-operator implementation of higher precision arithmetic are used in software, but would be expensive to unroll into a hardware environment. Analogous to the combination of bit level and word level techniques used in this work, the ideal would be to find a way of efficiently using single precision HFP, along with the flexibility of soft logic, to support other precisions.

8 Conclusions

In this article we have presented architectures for the single-precision natural logarithm and exponential, targeting FPGA architectures with hardware support for FP addition and multiplication, such as the Arria 10 and Stratix 10 FPGAs. For both functions, the proposed implementations make efficient use of an entire mix of resources: the DSP Block in fixed-point arithmetic mode, memory blocks, logic, and most significantly, the DSP Block in FP arithmetic mode. The accuracy of both function implementations is studied using the worst case cancellation values, and is accurate to $3ulp$, the bound required for OpenCL conformance [35], with subnormals flushed to zero o input and output.

Although timing values for the Stratix 10 device are preliminary, the critical paths of these architectures show that high frequencies, of over 700MHz, can be expected for upcoming devices.

References

[1] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," in *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 36–43. [Online]. Available: http://dx.doi.org/10.1109/.21

[2] N. Kapre and A. DeHon, "Accelerating SPICE Model-Evaluation using FPGAs," *IEEE Symposium on Field-Programmable Custom Computing Machines*, vol. 0, pp. 37–44, 2009. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2009.14

[3] V. Innocente, "Floating point in experimental HEP data processing," in *2nd CERN Openlab/INTEL Workshop on Numerical Computing*, 2012. [Online]. Available: http://indico.cern.ch/conferenceDisplay.py?confId=202688

[4] D. Piparo, "The VDT mathematical library," in *2nd CERN Openlab/INTEL Workshop on Numerical Computing*, 2012. [Online]. Available: http://indico.cern.ch/conferenceDisplay.py?confId=202688

[5] J. Apostolakis, A. Buckley, A. Dotti, and Z. Marshall, "Final report of the ATLAS detector simulation performance assessment group," CERN/PH/SFT, CERN-LCGAPP-2010-01, 2010. [Online]. Available: http://sftweb.cern.ch/AAdocuments

[6] NVIDIA Corporation, *CUDA C PROGRAMMING GUIDE*. NVIDIA Corporation, 2015.

[7] Intel, "Math kernel library," http://developer.intel.com/software/products/mkl/.

[8] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*, ser. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[9] W. J. Cody and W. Waite, *Software manual for the elementary functions*, ser. Prentice-Hall series in computational mathematics. Englewood Cliffs, N. J. Prentice-Hall, 1980. [Online]. Available: http://opac.inria.fr/record=b1081296

[10] 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.

[11] F. de Dinechin, M. Joldes, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *International Conference on Application-specific Systems, Architectures and Processors*. France Rennes: IEEE, Jul 2010.

[12] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–58, 29 2008.

[13] J. Harrison, "A machine-checked theory of floating point arithmetic," in *Theorem Proving in Higher Order Logics*, 1999, pp. 113–130.

[14] *Arria10 Device Overview*, 2014, http://www.altera.com/literature/hb/arria-10/a10_overview.pdf.

[15] "Altera OpenCL SDK," http://www.altera.co.uk/products/software/opencl/opencl-index.html.

[16] "SDAccel development environment – development environment for OpenCL, C and C++," http://www.xilinx.com/products/design-tools/sdx/sdaccel.html.

[17] O. Vinyals and G. Friedland, "A hardware-independent fast logarithm approximation with adjustable accuracy," in *Multimedia, 2008. ISM 2008. Tenth IEEE International Symposium on*, Dec 2008, pp. 61–65.

[18] A. D. Central, *AMD Core Math Library (ACML)*, Advanced Micro Devices, Inc., 2011, see http://developer.amd.com/libraries/acml/pages/default.aspx.

[19] J. Detrey and F. de Dinechin, "Parameterized floating-point logarithm and exponential functions for FPGAs," *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing*, vol. 31, no. 8, pp. 537–545, 2007.

[20] N. Alachiotis and A. Stamatakis, "Efficient floating-point logarithm unit for fpgas," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1–8.

[21] R. Bruce, M. Devlin, and S. Marshall, "An elementary transcendental function core library for reconfigurable computing," 2008.

[22] J. Detrey and F. de Dinechin, "A parameterizable floating-point logarithm operator for FPGAs," in *39th Asilomar Conference on Signals, Systems & Computers*. Pacific Grove, CA, USA: IEEE Signal Processing Society, Nov. 2005, pp. 1186–1190.

[23] ——, "Table-based polynomials for fast hardware function evaluation," in *Application-Specific Systems, Architectures, and Processors (ASAP'05)*. Samos, Greece: IEEE Computer Society, Jul. 2005, pp. 328–333. [Online]. Available: http://perso.ens-lyon.fr/jeremie.detrey/publications/pub/DetDin2005_asap.pdf

[24] N. Dhume and R. Srinivasakannan, "Parameterizable CORDIC-based floating-point library operations," 2012. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp552-cordic-floating-point-operations.pdf

[25] J. E. Volder, "The CORDIC trigonometric computing technique," *Electronic Computers, IRE Transactions on*, vol. EC-8, no. 3, pp. 330–334, Sept 1959.

[26] C. Doss and R. L. Riley, Jr., "FPGA-based implementation of a robust IEEE-754 exponential unit," in *Field-Programmable Custom Computing Machines*. IEEE, 2004, pp. 229–238.

[27] P. Echeverría, D. Thomas, M. López-Vallejo, and W. Luk, "An FPGA run-time parameterisable log-normal random number generator," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, 2008, vol. 4943, pp. 221–232, http://dx.doi.org/10.1007/978-3-540-78610-8_22.

[28] F. de Dinechin and B. Pasca, "Floating-point exponential functions for DSP-enabled FPGAs," in *Field-Programmable Technologies*. IEEE, 2010.

[29] ——, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design and Test*, 2011.

[30] N. Brisebarre, F. de Dinechin, and J.-M. Muller, "Integer and floating-point constant multipliers for FPGAs," *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, vol. 0, pp. 239–244, 2008.

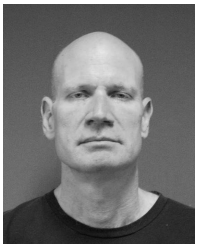
[31] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM, Philadelphia, PA, 2002.

[32] K. Chapman, "Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner)," *EDN magazine*, May 1994.

[33] "DSP Builder Advanced Blockset," http://www.altera.com/technology/dsp/advanced-blockset/dsp-advanced-blockset.html.

[34] F. de Dinechin, "A flexible floating-point logarithm for reconfigurable computers," Tech. Rep., 2010. [Online]. Available: http://prunel.ccsd.cnrs.fr/ensl-00506122/

[35] 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



Martin Langhammer received the BASc (Electrical Engineering) from the University of Toronto in 1988. He is currently a principal engineer with Intel, previously with Altera since 1999. He works in a number of different areas, such as FPGA architecture, arithmetic datapath design, FEC implementation, and compiler design. He serves on a number of technical program committees for FPGA, computing, and arithmetic conferences. He is a member of both the IEEE and the ACM.

Bogdan Pasca received a Ph.D. from ENS-Lyon in 2011. He is currently with Intel Programmable Solutions Group (PSG), previously with Altera since 2011. He was a major contributer in the FloPoCo project and is currently the main developer and maintainer of the arithmetic cores used throughout the Intel PSG products. His research interest include floating-point arithmetic and efficient mapping of arithmetic structures to FPGAs.