Efficient Floating-Point Implementation of the Probit Function on FPGAs

Mioara Joldeş, Bogdan Pasca

Efficient Floating-Point Implementation of the Probit Function on FPGAs

Mioara Joldes[1] and Bogdan Pasca[2]

[1]LAAS-CNRS, Toulouse, France, joldes@laas.fr
[2]Intel Corporation, France, bogdan.pasca@intel.com

Abstract

Non-uniform random number generators are key components in Monte Carlo simulations. The inverse cumulative distribution function (ICDF) technique provides a viable solution for generating random variables from various distributions. Thus, the ICDF of the standard normal distribution, or probit function for short, is of particular interest. The goal of this article is to revisit and improve a floating-point (FP) implementation of probit, from the perspective of modern hardware resources available on FPGAs. Beside reexamining the classical Wichura's algorithm, we propose: (1) a single-precision implementation using the embedded FP DSP Blocks available in recent FPGA families; (2) generic custom-precision architectures that scale up to double-precision. These present a user-selectable trade-off between tail accuracy and resource utilization. Our proposed cores outperform existing single-precision FPGA implementations in area, latency and accuracy, and also set benchmarks for new custom and double-precision FP implementations.

Keywords. Floating-point arithmetic, minimax approximation, FPGA, quantile, inverse error function.

1 Introduction

The hardware-based evaluation of elementary and special functions has recently received a lot of interest [16, Chap. 8], [18]. In this article we focus

on the hardware floating-point (FP) implementation of the probit function, which is the inverse cumulative distribution function for the standard Gaussian distribution, also called normal quantile. Specifically, let the standard normal cumulative distribution function be $\Phi : \mathbb{R} \to [0, 1]$,

$$\Phi(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{t} e^{-\frac{\alpha^2}{2}} \mathrm{d}\alpha. \tag{1}$$

The probit function f is defined as the inverse of Φ, with $f : [0, 1] \to \bar{\mathbb{R}}$, $f(x) = \Phi^{-1}(x)$, for $0 < x < 1$ and respectively $f(0) = -\infty$, $f(1) = +\infty$. Neither Φ, nor f have a closed-form in terms of elementary functions and usually they are expressed in terms of special functions, like the so-called error function erf, or its complementary erfc. For instance, one has:

$$f(x) = \sqrt{2}\,\mathrm{erf}^{-1}(2x - 1), \tag{2}$$

where

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-\alpha^2} \mathrm{d}\alpha. \tag{3}$$

Similarly to erf, erfc and their inverses, the probit function is more complex to implement than usual elementary functions, since range reduction techniques are not available and its asymptotic behavior (near 0 and respectively 1, see Figure 1) makes it more difficult to approximate by polynomials or rational fractions. Thus, the quality of implementation of probit is often assessed in terms of the maximum attainable standard deviation, which occurs at the smallest non-zero value in the input range: $\max_\sigma = |f(x_{\min})|$.

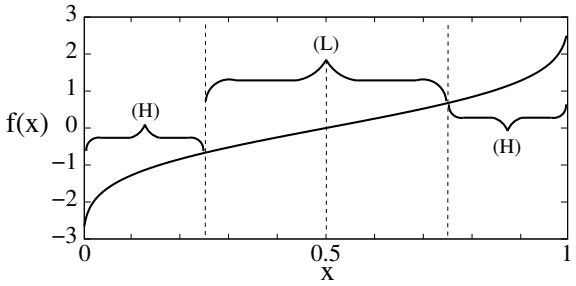


Figure 1: Probit Function

The main practical usage of the probit function lies in the Gaussian random number generation (GRNG). The so-called inversion method for

generating non-uniform random numbers is based on the fact that a quantile function monotonically maps uniform variates to variates of its corresponding distribution. The inversion method is thus considered as one of the best choices for random number generation. For the normal distribution, the lack of an analytical expression for the corresponding quantile function means that other methods may be preferred computationally. Several comprehensive studies already analyzed these choices and we refer the reader to [19, 15] and references therein.

While probit remains a viable alternative for GRNG (for instance it is currently the default method for sampling from a normal distribution in the statistical package R[1]), an efficient and accurate floating-point hardware implementation of this function is interesting in itself. The goal of this article is to revisit and improve such an implementation from the perspective of modern hardware resources available on FPGAs.

1.1 Related works

Among the software-based solutions, Wichura [20] proposed a three-subdomain rational approximation which suits single or double-precision computations and is implemented in statistics packages like R. Variations of this approach (see for instance [14] for a survey) are implemented in most numerical libraries, including Intel's Math Kernel Library (MKL), Boost's C++ Math Toolkit, and Nvidia's CUDA Math Library. Since modern FPGAs include Hardware-FP (HFP) DSP Blocks (which support single-precision multiply-add), it now makes sense to synthesize such a software-based algorithm to hardware.

Among hardware-based solutions, several works focus on fixed-point implementations. The thorough work of Lee et al. [4] proposes an architecture generation framework that can target arbitrary distributions. An accuracy-driven non-uniform segmentation scheme is used for splitting the input, and degree-2 polynomials (evaluated in fixed-point) approximate the function. For a 52-bit input, the output is on 16 bits, with last-bit-accuracy in terms of absolute error and $\max_\sigma = 8.2$. A fixed-point implementation differs from an FP one in the sense that both the inputs and outputs close to 0 hold very few bits of information. Since the approximation accuracy goal is different between fixed and FP implementation: absolute vs relative, the segmentation strategy also leads to different solutions.

[1] stat.ethz.ch/R-manual/R-devel/library/base/html/Random.html

To overcome these relative-accuracy shortcomings of fixed-point implementations, Echeverria and Lopez-Vallejo [9] adapt to hardware the software-based FP implementation from [11]. They use a more hardware-friendly segment-finding circuitry, to generate 256 subintervals and corresponding quintic FP coefficients Hermite polynomials. The claimed relative accuracy is $\approx 2^{-20}$ for a tail accuracy of $\max_\sigma = 6.23$.

Another FP implementation is presented by Schryver et al. [8]. It uses a hierarchical segmentation from [4] adapted to the FP format. Unlike [9], the inputs to the function are FP values in the interval $(0, 0.5)$ with an extra bit accounting for the symmetry. A degree 1 fixed-point piecewise-polynomial evaluator is used, but it remains unclear whether the output is in fixed or floating-point, since no normalization or exponent handling is presented or discussed.

1.2 Contributions and outline

With respect to previous works, this article presents:

– a family of single-precision (SP) architectures targeting modern HFP-based FPGAs. Generation-time architectural parameters are used for trading-off input range (affecting $\max_\sigma$) and resource utilization.

– concerning higher precision formats, a generic implementation strategy based on fused fixed-point piecewise polynomial approximation is proposed. It is applied for generating efficient architectures for three floating-point formats, including double precision.

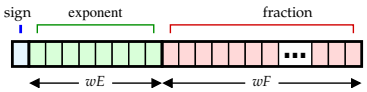
To this end, after recalling some basic notions in Sec. 2, we detail several approximation strategies for the probit function in Sec. 3: an analysis of Wichura's change of variable provides a more efficient segmentation method, which is then jointly used with coefficient-constrained minimax polynomial approximations. These are generated employing a high precision reliable golden reference implementation based on interval Newton's method. In Sec. 4, several custom input range segmentations are analyzed. Piecewise polynomials, together with a generic fused polynomial evaluator, discussed in Sec. 5, provide hardware implementations accurate to 3 ulps. The features of these cores are detailed in Section 6. Finally, the synthesis results are discussed and compared in Section 7.

This is an extended version of the homonymous article [12]. In the present work, besides a more thorough analysis of existing and proposed architectures, several custom input range segmentation schemes are overviewed, designed and implemented. With respect to [12], this allows for a much improved architecture for the single-precision implementation of the pro-

bit function, which either reduces by 35% the memory requirement for an equivalent tail accuracy, or increases the tail accuracy by 38% for a similar resource utilization.

2 Background

Let $x = (-1)^s 2^e M$ be the FP input, with sign s, exponent e and mantissa M. In this work we only focus on the regular range of the IEEE-754 [2] format, where the mantissa is normalized $M \in [1, 2)$. The corresponding IEEE-754 standard binary encoding for x is:



From left to right: the sign is encoded on 1 bit $\{0 \leftarrow \text{positive}, 1 \leftarrow \text{negative}\}$; the exponent holds on wE bits, and is stored as $e + \text{bias}$, with bias $= 2^{wE-1} - 1$; the fraction $F = M - 1$ is stored on the next wF bits. The IEEE-754 standard defines two compute-oriented formats: binary32 (single) having $wE = 8$, $wF = 23$ and binary64 (double) having $wE = 11$, $wF = 52$. Two other intermediary formats, namely $wE = 11$, $wF = 26$ and respectively $wE = 11$, $wF = 35$, are chosen to exemplify the proposed architectures. [2]

It is common to express the rounding errors of "nearly atomic" functions (arithmetic operations, elementary functions, etc.) in terms of ulp. For the purpose of this article, ulp(y) is defined as the distance between the closest two FP numbers straddling y [16, Def. 2.4]. In round to nearest, the error is 0.5 ulp. For this probit function implementation, we target an error budget of few ulp (say, 2 to 4, depending on the specifications).

The targeted hardware includes all Intel FPGA devices starting with the Arria 10/ Stratix 10 FPGA[1] onward, which have the following features relevant for this work. Firstly, their DSP Blocks that can be configured either in fixed or FP mode, to execute: (1) in fixed-point, one 27x27-bit multiplication, 2 independent 18x19 multiplications or one sum-of-two 18x19-bit multiplications; (2) in FP mode: one binary32 addition, multiplication, accumulation, or multiply-add. Furthermore, their basic logic-element is the ALM (Adaptive Logic Module). Finally, available M20K

[2]These correspond to formats with the same dynamic range as the binary64, but less precision, which can be useful for some applications. The fraction widths are chosen based on existing FP formats in the DSP Builder Advanced Blockset [3, Sec. 10.2].

memory blocks can be configured either in 512×40-bits or 1024×20-bits modes.

3 Approximations to probit function

For special functions (like erf, erfc, probit), where ad-hoc argument reduction techniques are not available and non-linear asymptotic behavior is present, a common FP implementation technique consists in dividing the input domain into several subdomains:
– when the behavior of the function is "sufficiently nice" for conventional polynomial or rational approximation to hold, that we denote by (L), see also Figure 1;
– "extremal subdomains", denoted by (H), where one has to cleverly use the asymptotic behavior of the function, together with polynomial or rational approximation.

The symmetry of the probit function:

$$f(1-x) = -f(x), \tag{4}$$

provides a first domain subdivision: one can then focus only on the interval $0 < x \leqslant 0.5$ or $0.5 \leqslant x < 1$.

From the FP perspective, the grid is finer on the range $0 < x \leqslant 0.5$, and hence, the implementation more challenging on this interval. In what follows, the interval $0 < x \leqslant 0.5$ is thus considered for the implementation and the higher range $0.5 \leqslant x < 1$ is obtained from equation (4). Note that for x in this range, the FP subtraction $1 - x$ is exact (always computed without rounding error) according to Sterbenz Lemma [16, Chap. 4].

When subdivision is performed, the coefficients of all polynomials (rational fractions) are tabulated and the hardware cost (multipliers, adders) amounts to the evaluation of the "worst-case" among the stored polynomials. At the same time, higher order approximations allow for better accuracy or for handling larger intervals. Therefore, a trade-off is to be found between the number of subdomains, approximation degree, accuracy provided and hardware resources. We analyze in what follows two types of approximations from this perspective.

3.1 Wichura's subdivision and change of variable

Wichura [20] and related methods [14] use rational approximations on the "sufficiently nice" (L) interval $[0 + b, 0.5]$, where $b > 0$ is a tail breakpoint.

The other (H) values in $(0, b)$, where the function is approaching the vertical asymptote (see Figure 1) are covered by at least one additional polynomial or rational approximation, which are in terms of a computationally expensive change of variable. This change of variable is related to the asymptotic behavior and can be obtained as follows.

For $w < 0$, one has

$$\Phi(w) \leqslant B(w), \text{ with } B(w) := -\frac{e^{-\frac{w^2}{2}}}{w\sqrt{2\pi}},$$

and

$$\lim_{w \to -\infty} \frac{B(w)}{\Phi(w)} = 1,$$

which states that for sufficiently large $|w|$, for negative w, $B(w)$ is a good approximation for $\Phi(w)$. Hence, to solve $\Phi(w) = x$, one can consider $B(w) \approx x$, take the natural logarithm $\log$ and obtain by rewriting:

$$\sqrt{w^2 + 2\log(-\sqrt{2\pi}w)} \approx \sqrt{-2\log x}. \tag{5}$$

Thus, for $w \ll 0$,

$$w \approx -\sqrt{-2\log x}. \tag{6}$$

Eq. (6) provides a means for computing an approximation for f near its asymptotics. As an example, Figure 2 shows the linearization effect of such a variable change: for $x \in [2^{-127}, 2^{-3}]$, the values of $f(x)$ are plotted on a $-\sqrt{-\log_2(x)}$ *linear scale*, as well as a degree 1 polynomial $P(y) := P\left(-\sqrt{-\log_2(x)}\right)$, with $P(y) = 0.8974609375 + 1.2421875y$. The maximum absolute approximation error between P and f is less than 0.105 on this range. To improve the approximation error, Wichura [20] considers a higher order approximation R:

$$f(x) \approx -R\left(\sqrt{-2\log x}\right), \tag{7}$$

for x sufficiently close to 0. From eq. (4), one has

$$f(x) = R\left(\sqrt{-2\log(1-x)}\right), \tag{8}$$

for x sufficiently close to 1. Wichura [20] tail breakpoint for performing the change of variable is $x < b = .075$. After the variable change, Wichura [20]

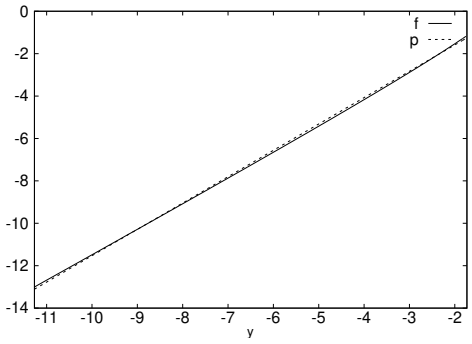


Figure 2: Plot of $f(x)$, on a *"$-\sqrt{-\log_2(x)}$"-linear scale*, for $x \in [2^{-127}, 2^{-3}]$; a linear approximation P with the Wichura's variable change: $P(y) := P\left(-\sqrt{-\log_2(x)}\right)$.

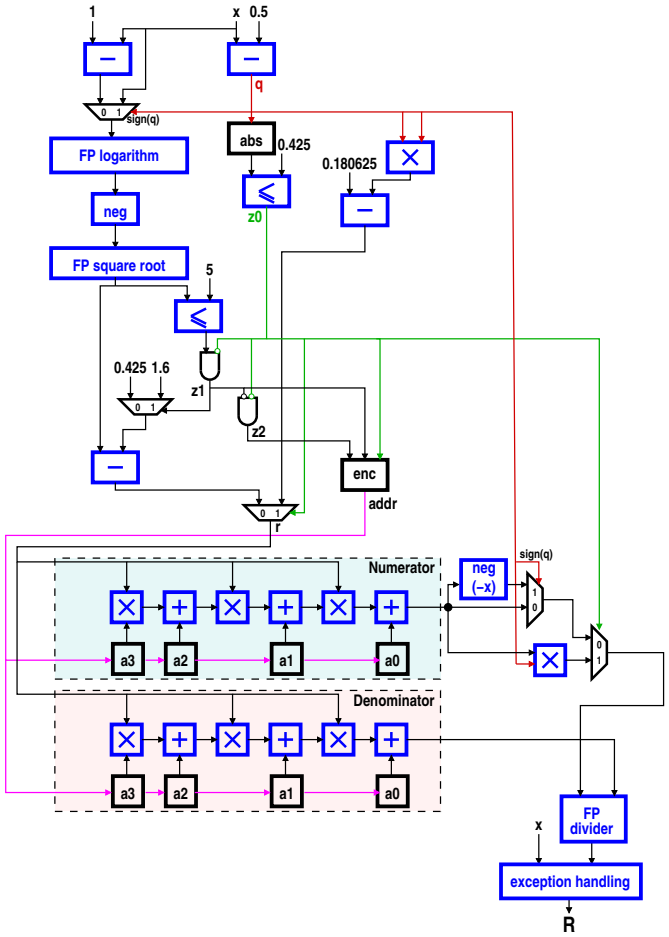


Figure 3: Hardware architecture of Wichura's algorithm.

uses two more higher order rational approximations, with a second breakpoint for very small inputs $\sqrt{-\log x} < 5$. The degree of these approximations is 3 (for both the numerator and the denominator) for single-precision and respectively 7 for double precision. A schematic view of the hardware implementation of this algorithm is given in Figure 3.

For a single-precision hardware implementation targeting HFP DSPs, the change of variable together with a division (necessary for rational approximation) is very costly: it accounts for more than 50% of the logic and DSP utilization. This can be seen in the first 4 rows of Table 6. For double-precision (last 4 rows of Table 6), the relative cost decreases, but still accounts for 35% of DSPs and latency.

Hence, our goal is to avoid the costly computation of the change of variable like $y = \sqrt{-\log_2 x}$, as well as the division. For that, we consider: in Section 4.2 a non-uniform segmentation of the input range of x, which would roughly translate to uniform segmentation for the range of y, on which piecewise polynomial approximations can be more easily performed (cf. Figure 2); furthermore, we employ minimax constrained-coefficients polynomial approximations, as detailed in what follows.

3.2 Polynomial Approximations

Previous approaches used either degree-2 Chebyshev approximations [13] or quintic Hermite interpolations [9]. We consider the minimax constrained-coefficients polynomial approximations provided by the Sollya software tool [5]. This open-source tool is the state-of-the-art for obtaining machine-tuned polynomial approximations and was already used in the implementation of other elementary and special functions [16, Chap. 10], [7, 18]. The following features are used:

– The `fpminimax` command inputs a function f, an interval I, a degree d and a list of constraints on the coefficients (e.g.constraints on FP formats, bitwidths). It returns the coefficients of a polynomial p of degree d, which minimizes the maximum of the (relative) approximation error $\varepsilon_{\text{approx}} := |f - p|/|f|$ on the given interval I, while satisfying the coefficients constraints.

– A certified upper-bound for the approximation error $\varepsilon_{\text{approx}}$ can be also obtained with Sollya; an upper bound for the floating-point or fixed-point evaluation error $\varepsilon_{\text{eval}}$ of the Horner scheme of p, can be obtained in a second step with the Gappa software [6].

Unfortunately, the probit function was not implemented in Sollya. However, Sollya provides arbitrary precision, as well as certified computa-

ALGORITHM 1: PROBITEVALNEWTON(x, $\mathcal{I}$, ε).

1: $T \leftarrow \mathrm{erfc}\left(\frac{w}{\sqrt{2}}\right) - 2 + 2x$
2: **while** `intervalDiam`$(\mathcal{I})/2 \leqslant \varepsilon$ **do**
3: $\quad w_0 \leftarrow$ `intervalMidpoint`$(\mathcal{I})$
4: $\quad \mathcal{T}' \leftarrow$ `evaluate`$(T', \mathcal{I})$
5: $\quad \mathcal{W} \leftarrow$ `evaluate`$(w_0 - T(w_0)/\mathcal{T}')$
6: $\quad \mathcal{I} \leftarrow$ `intervalIntersect`$(\mathcal{W}, \mathcal{I})$
7: **end while**
8: **return** `intervalMidpoint`$(\mathcal{I})$

tions (interval arithmetic) for erf (and erfc). Hence, we developed an arbitrary precision faithfully-rounded implementation[3] for the probit in Sollya based on inverting the erfc function, with interval Newton method [17], as follows.

Arbitrary precision implementation. The probit function is implemented in Sollya, based on solving for w the equation $\Phi(w) - x = 0$, which is equivalent to $T(w) = 0$ where T and its derivative with respect to w are:

$$T(w) := \mathrm{erfc}\left(\frac{w}{\sqrt{2}}\right) - 2 + 2x, \tag{9}$$

$$T'(w) = -\sqrt{(2/\pi)}\exp\left(-w^2/2\right). \tag{10}$$

These functions can be evaluated with arbitrary accuracy in Sollya and their range on a given interval $\mathcal{I}$ can be tightly enclosed, using the command `evaluate`$(T, \mathcal{I})$. This gives the interval Newton Algorithm 1, which computes an evaluation of the probit function at x, with required accuracy[4] ε, starting with an initial guess range $\mathcal{I}$, s.t. $f(x) \in \mathcal{I}$. The advantage with respect to the classical Newton method is that the algorithm is guaranteed to always converge, even if the initial guess range is very wide, or very small, provided arbitrary precision computations are available [17]. This algorithm was also easily coded in C based on the MPFR library [10] and the monotonicity properties of erfc and its derivative. Hence, it provides a very flexible accuracy and open-source golden reference for the probit function, allowing both for generating tuned polynomial approximations (with various coefficients constraints) based on `fpminimax` command and, for a posteriori rigorous testing and validation of the results.

The remaining question is how to select a suitable subdivision of the input range, so as to balance the number of polynomials and their degree.

[3] available at `http://homepages.laas.fr/mmjoldes/probit`

[4] The absolute accuracy test in line 2 can be made relative by dividing with w_0, provided that $w_0 \neq 0$

4 Input Range Subdivision Strategies

We discuss in what follows several custom segmentations, similar to the hierarchical one, which was already used in [4, 8, 18] in both the fixed-point and floating-point setting. First, the most simple one i.e, the uniform subdivision is briefly recalled.

4.1 Uniform subdivision

On the (L) range $[0.25, 0.5)$ of the input interval, the function is sufficiently *nice* and thus, a uniform segmentation is the most common and often very efficient strategy.

For a single-precision target implementation this strategy needs 128 subintervals for degree-2 polynomials and respectively, 16 subintervals when degree-3 polynomial approximations are considered.

For a double-precision implementation we propose degree-8 minimax polynomials, as a good compromise for uniform subdivision, which requires 16 subintervals on the (L) range.

4.2 Logarithmic subdivision

Let us now focus on the (H) part of the input interval, where the variable change is to be avoided. The intuition is that the integer $I = (-1)^s(e + F)$ aliased to a floating-point number $x = (-1)^s 2^e M$ (where $F = M - 1$), is a scaled and shifted approximation of the logarithm. Hence, taking a uniform segmentation on the aliased integer provides a non-uniform segmentation of the range of x suitable for the change of variable $y = \log_2 x$. Thus, a suitable segmentation scheme for probit can be composed by concatenating the exponent e and a specific variable number of fractional bits (depending on each binade), which are obtained function of the approximation constraints.

For a single-precision target implementation Table 1 shows the minimum required number of address fractional bits aw_F, depending on the exponent range, when imposing degree-2 (and respectively degree 3) minimax approximations and $\varepsilon_{\text{approx}} \leqslant 2^{-23}$ for each corresponding polynomial. As expected, one observes that aw_F decreases proportionally to $\sqrt{e}$, which is in fact dictated by Wichura's change of variable (6), and thus, this segmentation roughly "simulates" it.

For a double-precision implementation, we propose degree 8 minimax polynomials as a good compromise: 16 subintervals were needed on the

Table 1: Logarithmic segmentation scheme for single-precision and targeted 2^{-23} relative approx error.

Exponent	aw_F
$[-3, \ldots, -26]$	5
$[-27, \ldots, -90]$	4
$[-91, \ldots, -126]$	3

(a) Degree 2

Exponent	aw_F
$[-3, \ldots, -34]$	3
$[-35, \ldots, -94]$	2
$[-95, \ldots, -126]$	1

(b) Degree 3

Table 2: Logarithmic mantissa segmentation scheme: real bound vs. 6 fraction bits approximation, for double-precision.

Segment bound	a_F
$2^{1/8}$	000110
$2^{2/8}$	001100
$2^{3/8}$	010011
$2^{4/8}$	011011
$2^{5/8}$	100011
$2^{6/8}$	101100
$2^{7/8}$	110101

"nice" (L) input range $x \in [2^{-2}, 2^{-1})$; to fill in memory constraints up to 512 subintervals, we consider a budget of 496 subintervals for the (H) input range $x \in [2^{-64}, 2^{-2})$. It is interesting to note the following subtle improvement obtained by also performing a non-uniform mantissa subdivision per binade.

Example of non-uniform mantissa subdivision for double-precision. Consider 8 equally sized subintervals of the mantissa in the binade $x \in [2^{-3}, 2^{-2})$. The best relative approximation error for a degree-8 polynomial approximation on the range $x \in [2^{-3}, 2^{-3} \cdot 1.125)$ is $\varepsilon_{\text{approx}} \simeq 2^{-50}$, which does not provide ulp accuracy. However, by subdividing the range $y = \log_2(x) \in [-3, -2)$, in 8 equally sized intervals and checking the resulting degree 8 approximation for $x \in [2^{-3}, 2^{-3+1/8})$ one obtains $\varepsilon_{\text{approx}} \simeq 2^{-54}$, which is ulp accurate. Similar results are obtained for the other intervals. Thus, the more accurate resulting segmentation bounds (uniform on the $\log_2$ range) are $2^{1/8}, 2^{2/8}, \ldots, 2^{7/8}, 2$. A simple addressing scheme is done by a lookup table, which maps the first 6 fractional bits of each input x to one of the 8 corresponding non-uniform segments, by approximating the fractional part a_F of $2^{i/8}$ on 6 bits cf. Table 2.

This trick which again "simulates" the $\log_2$ change of variable, without actually computing it, allows for keeping degree 8 polynomials over the entire considered range $x \in [2^{-64}, 2^{-1})$ and avoid a roughly 10% overhead by increasing the degree of the approximation to 9.

ALGORITHM 2: GENERATESUBDIVISIONLUT(aw_F, w_o).

1: $a_{F_0} \leftarrow 0$
2: **for** $i \leftarrow 1$ **to** 2^{w_o} **do**
3: $\quad a_{F_i} \leftarrow \texttt{nearestint}\left((2^{i/2^{w_o}} - 1) \cdot 2^{aw_F}\right)$
4: $\quad$ **for** $j \leftarrow a_{F_{i-1}}$ **to** $a_{F_i} - 1$ **do**
5: $\quad\quad$ LUT$[j] \leftarrow i - 1$
6: $\quad$ **end for**
7: **end for**
8: **return** LUT

4.3 Further improvements on subdivision techniques

Custom segmentation schemes are well-known [18, 4], but are rather seldom used in practice, due to the typically high-cost associated with finding the interval that an input x belongs to. This cost can sometimes outweigh the benefits of reducing the number of stored coefficient sets.

However, the previously mentioned trick, implemented in [12] only for the double-precision case, proved very effective. It turns out that such custom non-uniform subdivision techniques can be very practical also for the single-precision implementation, as discussed in what follows.

The (H) branch for single-precision. When generalizing the example above, the goal is to provide for a given binade $x \in [2^{-e_x}, 2^{-e_x+1})$ a subdivision strategy into say, 2^{w_o} subintervals, which is more accurate than the logarithmic one (for which the mantissa is uniformly segmented). Furthermore, the overhead required for the new addressing scheme should be kept small. To this end, a uniform segmentation of the range of $y = \log_2(x) \in [-e_x, -e_x + 1)$ is used, from which a direct calculation gives the real-number segmentation bounds on the mantissa: $2^{1/2^{w_o}}, 2^{2/2^{w_o}}, \ldots, 2^{(2^{w_o}-1)/2^{w_o}}, 2$.

Subsequently, a simple encoding scheme is designed via a lookup table, to map the first $aw_F \geqslant w_o$ fractional bits of each input x, to one of the 2^{w_o} corresponding non-uniform segments. This is done by approximating the fractional part of the bounds $2^{i/2^{w_o}}$ on aw_F bits. For instance, the first aw_F fractional bits of x corresponding to the first segment must take values between $00\ldots0$ and the nearest integer to $(2^{1/2^{w_o}} - 1) \cdot 2^{aw_F}$ (see also Table 2 for an example). Finally, the resulting w_o bits are then used for addressing the polynomial coefficients. This process is summarized in Algorithm 2.

This new strategy was tested and implemented on the (H) range of the single-precision case. An improved segmentation scheme was thus obtained, as shown in Table 3. For instance, for degree-2 polynomial approx-

Table 3: Improved logarithmic segmentation scheme for single-precision and targeted 2^{-23} relative approx error.

Exponent	aw_F	w_o
$[-3, \ldots, -10]$	5	5
$[-11, \ldots, -54]$	6	4
$[-55, \ldots, -126]$	5	3

(a) Degree 2

Exponent	aw_F	w_o
$[-3, \ldots, -18]$	5	3
$[-19, \ldots, -126]$	4	2

(b) Degree 3

imations, one needs only $w_o = 4$ bits for addressing each binade with exponents in range $[-11, \ldots, -54]$, and $w_o = 3$ bits from exponents in range $[-55, \ldots, -126]$, while with the previous classical logarithmic scheme 5 bits were required for addressing each binade with exponents in $[-3, \ldots, -26]$, and the 4, respectively 3, bits kicked-in only at -27 and respectively -91. This improvement only costs 3 small lookup tables (e.g. input on 6 bits and output on 4) for the whole single-precision range. All in all, this translates to an important resource optimization and/or better attained $\max_\sigma$, as shown and discussed in Section 7.

The (L) Branch for single-precision. As previously stated in Section 4.1 the "nice" part of the function can be handled by a uniform approximation, which for degree-2 approximations, would require 128 subintervals. However, by customizing this segmentation, we can further reduce this interval count with minimal resource logic utilization impact. The advantage is that this storage reduction for the (L) branch allows for handling a wider exponent range in the (H) branch, without increasing the total block memory count, thus improving the attained $\max_\sigma$.

The core of this improvement is an accuracy-based semi-coarse greedy segmentation scheme. The starting point is the observation that the initial uniform segmentation scheme on the (L) interval $z = 2|0.5-x| \in [0, 0.5)$ accounts for the worst-case accuracy requirements on the subintervals close to $z = 0$, where the function $\mathrm{erf}^{-1}(z)$ is harder to approximate. [5] However, it is possible to progressively widen subintervals (in coarse steps), when getting away from 0, while satisfying the accuracy constraints, since the function is easier to approximate on this part.

The widening is roughly based on a greedy aggregation of several adjacent subintervals from the initial uniform scheme, as long as the accuracy constraints are still met. If the number of adjacent intervals considered is a power of 2, this method is the hierarchical segmentation of [4] and

[5] The multiplication by 2 in the previous change of variable is simply related to the fact that the probit function is the inverse erf function of argument $2x - 1$ (see eq. (2)), so the approximations and segmentations will be directly related to those of the inverse error function which is symmetric around 0.

can also be seen as a recursive binary splitting of the initial interval until the accuracy constraints are met. For instance, in the original uniform segmentation scheme the subinterval width is 2^{-8}, which implies that the range $[2^{-4}, 2^{-3})$ is split into 16 segments. However, 8 segments of size 2^{-7} are sufficient to meet single-precision accuracy on this range. With this strategy the minimal number of segments obtained for (L) was 55. Note that with a uniform segmentation, the required number of segments was 128, cf. Section 4.1.

Moreover, better segment sizes may not be a convenient power of two. For example, the segment $[2^{-5}, 2^{-5} + 1.5 \cdot 2^{-8})$ on which accuracy requirements are met is wider than the initial scheme step, so it could be considered, provided an efficient addressing scheme. The most flexible approach, given in [18], consists in: (1) storing strictly non-overlapping segment bounds either in lookup tables or block RAMs (for an increased number of segments); (2) given an input, the corresponding segment is obtained via a binary search, which boils down to the hardware implementation of a comparison at each stage. To improve on this method by avoiding the block RAM storage, as well as reducing the number of comparisons, our approach is a hybrid between the fully flexible one [18] and the traditional uniform one: it builds a small set of (non necessarily disjoint) subintervals on which uniform segmentation schemes are employed. Clearly, each considered uniform scheme may have a different internal subinterval size.

Therefore, (L) is split in the following branches, as also shown in Figure 4:

Branch #0 The first interval near $z = 0$ is the most challenging. For an approximation error bound of 2^{-25}, the interval size needs to be as small as 2^{-10}. The next interval can be 3 times wider ($3 \cdot 2^{-10}$) – covering the input range $[2^{-10}, 2^{-8})$ – and still producing a similar approximation error.

Branch #1 Spans the interval $[2^{-8}, 2^{-5})$. This range is split into 7 intervals of size 2^{-8}.

Branch #2 Spans the interval $[2^{-5}, 2^{-4} \cdot 1.10001_2)$. On this range 6 subintervals of size $1.5 \cdot 2^{-8}$ are used.

Branch #3 Spans the interval $[2^{-4}, 2^{-3})$. This overlaps slightly on the final interval of the previous branch, but this is expected and is selected in order to ease addressing. On this range a total of 8 sub-intervals of size $2 \cdot 2^{-8}$ are used.

Branch #4 Spans the interval $[2^{-3}, 2^{-1})$. A total of 32 subintervals of size $3 \cdot 2^{-8}$ are used on this range.

The circuitry used for decoding the currently active branch is presented on the top of Figure 5: it is based on comparing the input with the bounds of each branch. However, since these bounds have particular formats, this comparators (highlighted as gray boxes) can be much simplified. Once the active branch was decoded, the final address is computed as the sum between the branch offset (the right MUX) and the local index within the branch (the left MUX). For the sake of clarity, the inputs of the left MUX are also depicted in Figure 4.

5 Efficient polynomial fixed-point and FP evaluations

When generating the polynomial approximations, several argument and function scaling techniques are required for efficiency. It is important to note that depending on the evaluation used, either a FP or a fixed-point input is built for a corresponding FP, or respectively, a fixed-point evaluation. One main distinction between the two cases is the following: the input of the polynomial fixed-point evaluation is relative to the subinterval bounds, hence, for each subinterval, a subtraction is necessary to shift it relatively to the left subinterval bound. This can be implicitly done, for a uniform segmentation by taking only the corresponding lower bits, but it is performed in fixed-point, when the bounds are not uniform. On the other hand, the FP input is *absolute*, in the sense that no additional subtraction is necessary on each subinterval. This is further detailed below.

FP evaluation. For the case of SP, a FP evaluation can be performed in order to take advantage of the HFP-DSP. This choice is experimentally justified by the proposed synthesized architectures, as discussed in Section 7.

On the (L) domain, the reduced input argument, evaluated in FP, $z = 2|0.5 - x| \in [0, 0.5]$ is the direct FP input to the polynomial evaluator $q(z)$, which is generated with:

```
q(z)=fpminimax(f((1+z)/2), 2, [|24 ...|], I);
```

where I is obtained from the subdivision strategy employed (either uniform cf. Sec. 4.1 or custom cf. Sec. 4.3).

For (H), which proceeds by binade $x \in [2^e, 2^{e+1})$, a potential overflow, in the polynomial coefficients for high magnitude e, is avoided by rescaling the polynomial input to $z = x/2^e$:

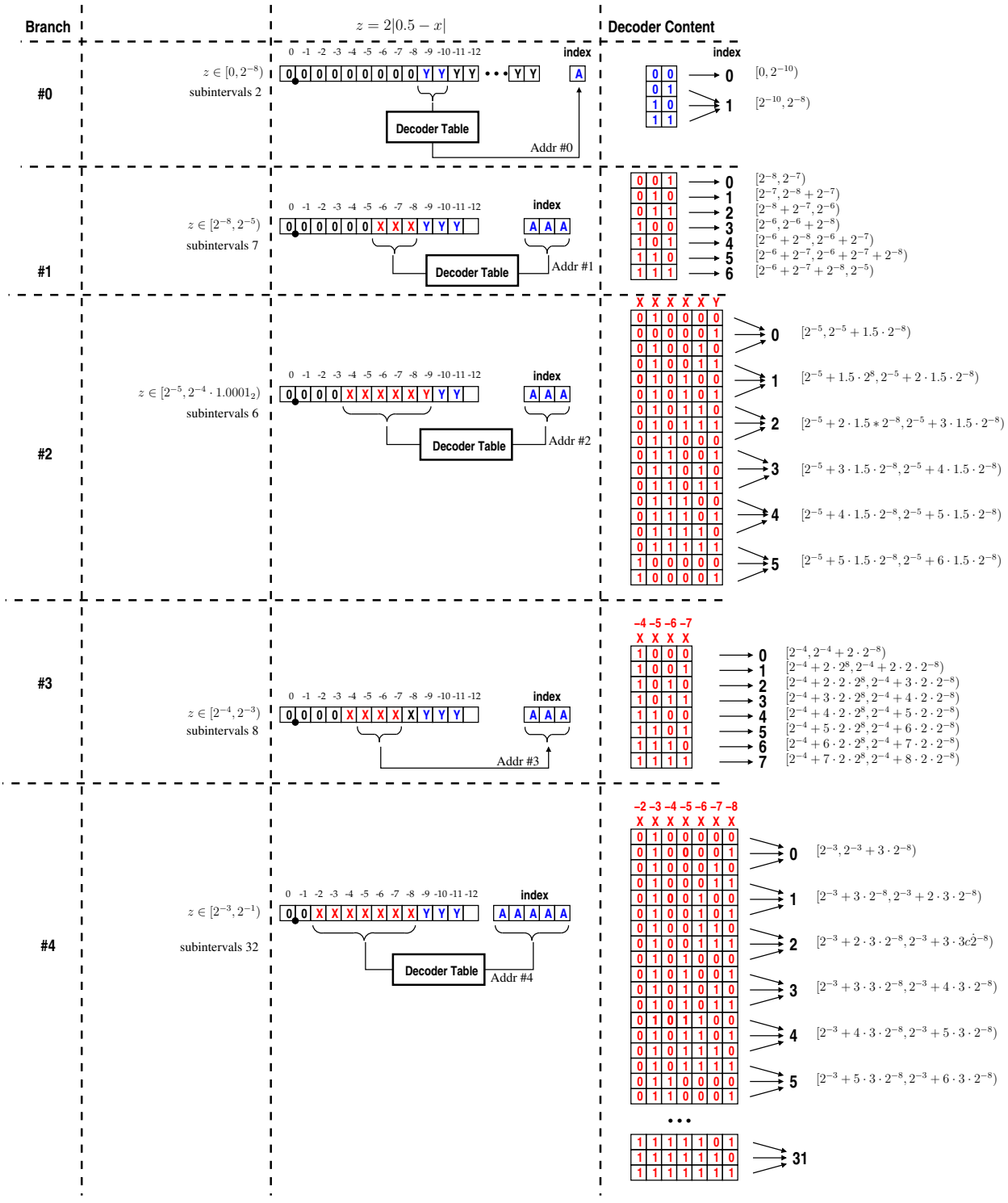


Figure 4: Custom Segmentations for the (L) range: 5 sub-branches are considered, each of which is uniformly segmented with a different internal subinterval width. For a given branch, the indexing of each of its subintervals is based on a lookup table as shown in the fourth column: when the subinterval width is a power of two, this can be simplified to a simple offset e.g., branch #1 and #3. When the width is custom e.g., branch #0, #2, #4, the lookup tables need to be explicitly filled-in.

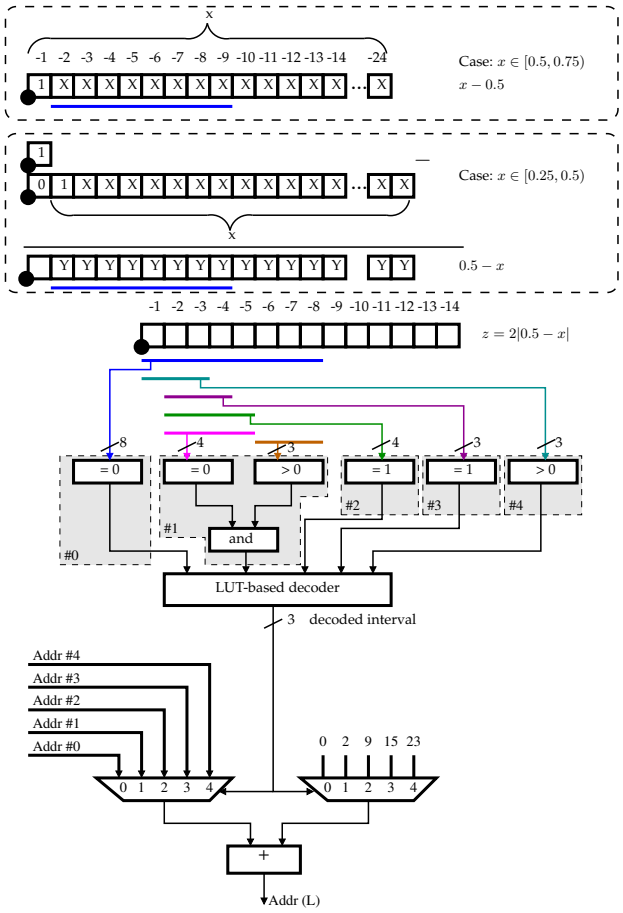


Figure 5: Addressing scheme on (L) branch: gray boxes correspond to simplified comparators, the right MUX inputs the branch offsets and the left MUX selects the local index within each branch. The input bits of $z = 2|0.5 - x|$ (underlined in blue) correspond to either top bits of F_x when $x \geqslant 0.5$ or top bits of $0.5 - x$ when $x < 0.5$.

```
q(z)=fpminimax(f(2^(-e)*z), 2, [|24...|], I);
```

where I will be some subinterval of $[1, 2)$. The reduced input argument on the (H) branch is thus obtained by concatenating a new sign (0) and the exponent value (0+*bias*) to F_x.

Fixed-point evaluation. The goal is to match the polynomial input and output ranges for both (H) and (L) evaluation branches. The output range is straightforwardly scaled to [0.5,2) by considering $f/2^{max\_exp}$, where the maximum exponent (in absolute value) is obtained when evaluating f on the two interval ends.

On the (L) branch, firstly, the same input argument reduction like in the previous FP case is done, but in this case the evaluation $|0.5 - x|$ is in fixed-point. Afterwards, for a uniform subdivision on the (L) branch, the argument is obtained by taking the corresponding lower order bits, which are denoted by z_{shift}. For example, $0 \leqslant z_{\text{shift}} < 2^{-5}$, when the input is split in 16. On the other hand, on the (H) branch, the subdivision is not uniform, so for an argument $z \in [l, r]$, the evaluation is performed in $z_{\text{shift}} = z - l$. This gives for example, when 3 fraction bits are used for addressing: $0 \leqslant z_{\text{shift}} \leqslant 2^{-3}$. Then a further scaling down is employed to match 2^{-5}. A final technicality is that for the first interval in (L), $z = 2|0.5 - x|$ is very close to zero, so to account for efficient fixed-point evaluation, a final multiplication by z is performed outside the fused polynomial evaluator.

6 Architecture

Several architectures are discussed:

- Firstly, and for the sake of simplicity, we deal with the single-precision case, when degree-2 polynomial approximations are employed, and when the classical uniform segmentation method is used for the (L) branch, together with the logarithmic one for the (H) branch, as discussed in Sec. 4.1 and 4.2. The architecture is depicted in Fig. 6. Then, the necessary changes related to the customized segmentation (Sec. 4.3) are highlighted in Fig. 8 and briefly analyzed with respect to the previous architecture.

- Secondly, the single-precision case, with degree-3 polynomial approximations is discussed from the same two perspectives: classical vs. customized segmentations.

- Finally, a generic architecture is presented (see Fig. 10), based on fixed-point piecewise-polynomial approximations.

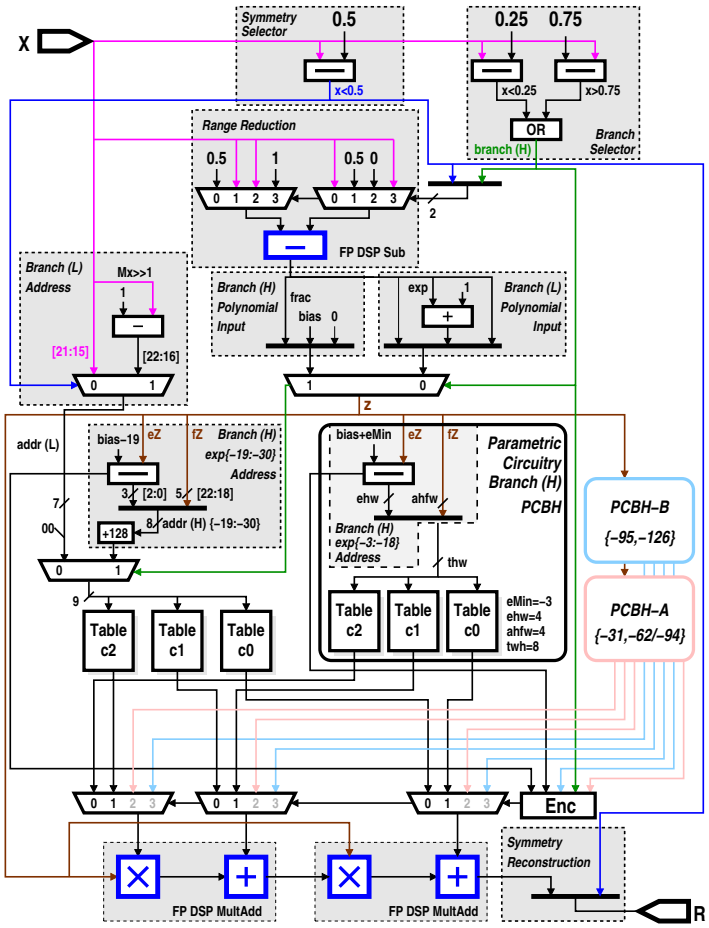


Figure 6: SP architecture of FP probit for HFP-enabled FPGAs.

6.1 Single-Precision - degree 2

A SP architecture targeting HFP DSP-Enabled FPGAs is presented in Figure 6. The implementation presents two distinct branches: (L) $x \in [0.25, 0.75]$ and (H) for the remaining range. The function is approximated by degree-2 piecewise polynomial approximations.

Branch (L) argument is reduced as presented in Sec. 5 *(a)*. Then, a total of 128 subintervals are used, with an approximation error less than 1 ulp. The subinterval selection can be done starting with z (floating-point) and then aligning it using a barrel-shifter. This costly operation is avoided by addressing from x:

– for $x \in [0.5, 0.75]$, the address line consists of the [21:15] bits from the fraction of x.

– for $x \in [0.25, 0.5)$, the address is obtained by selecting bits [22:16] of

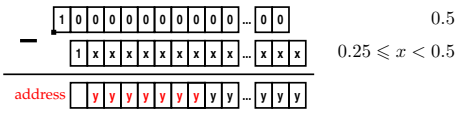


Figure 7: Fixed-point alignment for (L) branch address computation, when $x < 0.5$. The operation is a $2 + wF$-bit subtraction.

the fixed-point difference $0.5 - x$; the alignment of both terms is known, as shown in Figure 7.

Branch (H) handles inputs in the ranges $(0, 0.25)$ and $(0.75, 1)$. Values corresponding to $(0.75, 1)$ are obtained from the symmetry eq. (4). The logarithmic segmentation technique (see Sec. 4.2), with degree-2 polynomials, requires a different number of subintervals, function of the corresponding exponent as mentioned in Table 1. Based on this, the number of exponents that can be handled is found by using the coefficient table sweet-spot: 512×40-bit for the M20K blocks. Therefore, if we restrict the total number of subintervals stored to be 512, we can store as many as 512/32=16 exponent values. This covers the range of exponents from -3 to -18. Additionally, since the coefficient tables for branch (L) only use 128 out of the 512 address lines, an additional 12 exponents $\{-19, \ldots, -30\}$ can be handled by fully packing the branch (L) tables. The tail accuracy of this architecture, denoted by SP-HFP-d2-U-e30 (single precision, HFP DSP used, degree-2 approximations with Uniform/classical segmentations, and minimum handled exponent of -30) in Tables 5 and 6 is $\max_\sigma = 6$.

The handled exponent range can be further increased by adding the auxiliary circuitry PCBH-A, which itself has two configurations:

– *PCBH-A*$_1$ = *PCBH* with *eMin* $= -31$, *ehw* $= 5$, *ahfw* $= 4$ handles exponents from -31 to -62.

– *PCBH-A*$_2$ = *PCBH* with *eMin* $= -31$, *ehw* $= 6$, *ahfw* $= 4$ handles exponents from -31 to -94.

The corresponding architectures are denoted by SP-HFP-d2-U-e62 and SP-HFP-d2-U-e94 in Tab. 5 and 6, with tail accuracy of $\max_\sigma = 8.92$ and respectively $\max_\sigma = 11.11$.

Specifically, the number of bits required to encode the exponent range is denoted by *ehw*. For the range of exponents handled by *PCBH-A*$_{1/2}$, a total of 16 subintervals are required for meeting the approximation error budget, hence the address is stored on 4 bits (*ahfw* $= 4$). Finally, the number of address bits is *ahfw* + *ehw*.

Circuitry *PCBH-B* can be used in conjunction with *PCBH-A*$_2$ to increase the range of handled exponents to the full range of the SP format, corre-

sponding to -126, with $\max_\sigma = 12.94$ and denoted by SP-HFP-d2-U-e126 in Tab. 5 and 6. The logic is similar to that *PCBH-A*, with the difference that the number of subintervals required for each exponent is reduced to 8, cf. Table 1, $ahfw = 3$ bits.

A final level of multiplexers selects the coefficients depending on the branch enabled (H) or (L), and the signs of the different differences ($bias + eMin - eZ$) that are sufficient for determining the current branch.

Degree-2 polynomial SP evaluation is based on Horner's scheme. Two DSP Blocks are configured in multiply-add mode, and chained as depicted on the bottom of Fig 6. A worst case error of 2 ulps is introduced by the FP evaluation (chain of 4 operations), leading to a maximum error of 3 ulps (combined approximation and evaluation error). The final result is constructed by appending the symmetry bit (x<0.5) to the exponent and fractions returned by the polynomial evaluator.

Optimized version for SP, degree 2. A further optimized version of this architecture is depicted in Figure 8. There are two main novelties of this architecture. First, the architecture is now based on the custom greedy segmentation of the (L) branch described in Section 4.3. This reduces the number of subintervals from 128 to 55, thus allowing for more (H) branch exponents to be handled (this in turn improves $\max_\sigma$ of the architecture). Secondly, the non-uniform segmentation – discussed in Section 4.3 – is used for the exponents range of the (H) branch. This allows for further reducing the number of subintervals required for these binades. In a similar manner, the configurable architectures are denoted by SP-HFP-d2-C (single precision, HFP DSP used, degree-2 approximations with custom/non-uniform segmentations) in Tables 5 and 6

The coefficient tables corresponding to the (H) branch which, for the architecture in Figure 6, handled exponents from $\{-3, -18\}$, now pack exponents from $\{-3, -26\}$. The bottom 256 table entries handle exponents $\{-3, -10\}$, where a uniform segmentation with 32 subintervals is used for each exponent. The upper 256 table entries handle exponents $\{-11, -26\}$. For each of these exponents, 16 non-uniform subintervals are used – 6 fractional bits are decoded to 4 bits (corresponding to 16 subintervals) using the approach detailed in Algorithm 2.

The (L) branch coefficient tables store in the bottom 55 positions the coefficients corresponding to the (L) branch. The addressing of these subintervals, depicted with the black box "55 (L) address generator" in Figure 8, is detailed in Figure 5. Similarly to the architecture in Figure 6, the rest of this table is filled-up with (H) branch exponents. More precisely, exponents in the range $\{-27, -54\}$ which have a similar behavior as $\{-11, -26\}$ (non-uniform 16 subintervals) sum-up to $28 \cdot 16 = 448$ table entries. To-

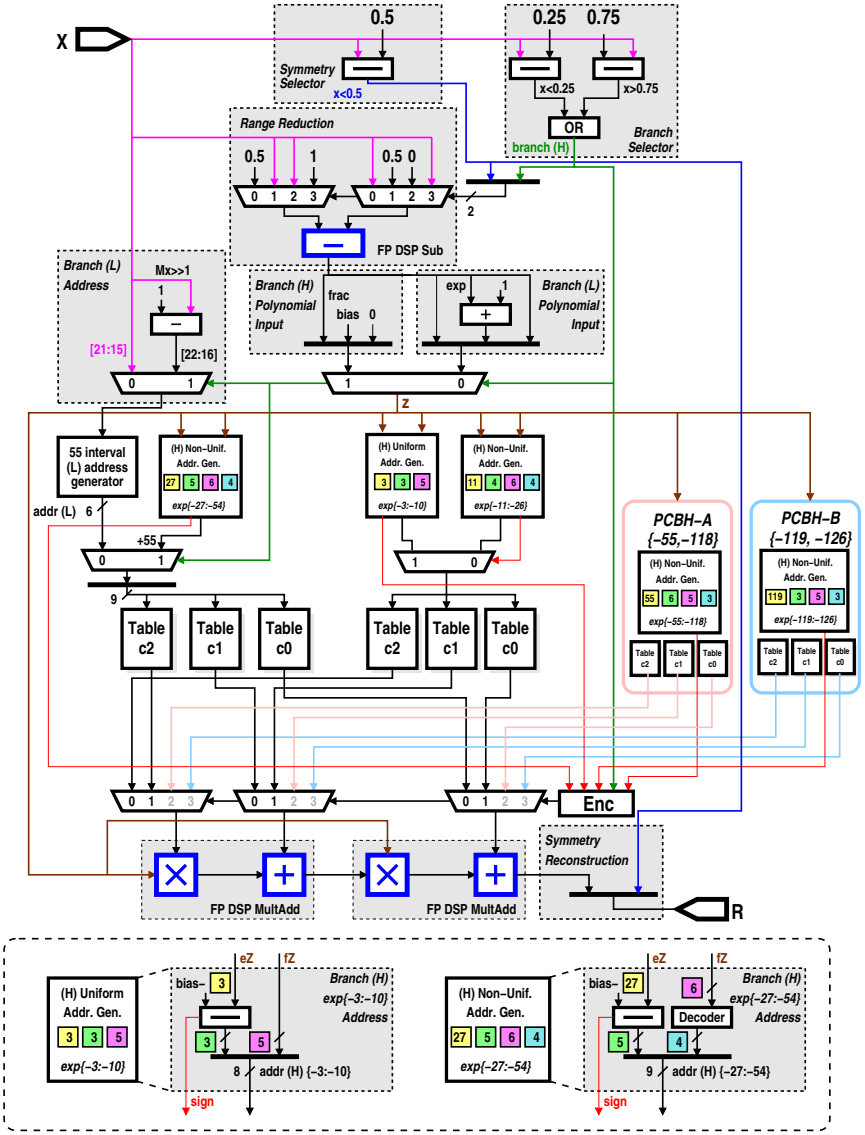


Figure 8: Optimized SP architecture of FP probit for HFP-enabled FPGAs. The bottom part of the figure shows the details of the high-level components used for the (H) branch addressing.

gether with the (L) branch exponents the total utilization of these tables is $55 + 448 = 503$. This optimized architecture, denoted by SP-HFP-d2-C-e54 in Tables 5 and 6 has $\max_\sigma = 8.3$.

In order to further extend the supported exponent range (and thus improve $\max_\sigma$) circuits PCBH-A and PCHB-B can be used, similarly to the previous architecture).

- *PCBH-A* handles exponents from $\{-55, -118\}$. For each of these exponents, a non-uniform segmentation scheme with 8 subintervals is used. This optional circuitry (SP-HFP-d2-C-e118 in Tab. 5 and 6) results in an improved $\max_\sigma = 12.51$.
- *PCBH-B* handles the rest of allowed single-precision exponents. For each of these exponents, 8 non-uniform subintervals are used. This results in a total of 8 exponents $\times\ 8$ subintervals $= 64$ table entries. The size of this table allows for an efficient 6-input LUT-based implementation, and thus no extra memory blocks are required. In conjunction with PCBH-A, this circuit (SP-HFP-d2-C-e126 in Tab. 5 and 6) allows for a full exponent range coverage.

6.2 Single-Precision - degree 3

A different trade-off between DSP and memory blocks can be obtained if the polynomial degree is increased to 3. The reduced number of subintervals on both (L) and (H) leads to the memory compaction shown in Fig 9.

As presented in Sec 4.2, (L) branch requires only 16 subintervals, and thus occupies a small size of the 512 coefficient tables. Next, two subsections of branch (H) handle exponents up to -94. First, for exponent range {-3,-34} each exponent requires 8 subintervals but for the range {-35, -94} only 4 subintervals suffice for meeting the approximation error objective. This architecture denoted by SP-HFP-d3-U-e94 in Tab. 5 and 6 has a $\max_\sigma = 11.11$.

The addressing is detailed in Figure 9 and is composed of a set base address plus offset. The signs of the subtracters s_0 and s_1 corresponding to *bias* $- 3 - e_Z$ and *bias* $- 35 - e_Z$ select the base address from 3 possible values 0, 256, and 256+16. The same signs also drive a *MUX* selecting between the 3 local offsets. The final table address is obtained by adding the base and the offset values.

Optimized version for SP, degree 3. An optimized segmentation on the (H) branch allows for an improved $\max_\sigma$. More precisely, on the exponent range $\{-3, -18\}$, a customized segmentation in 8 subintervals per binade meets the accuracy target, while on the exponent range $\{-19, -110\}$,

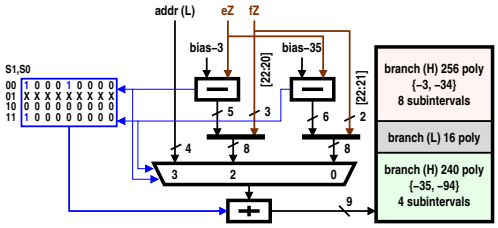


Figure 9: Coefficient memory composition and addressing for a SP architecture based on a degree 3 polynomial evaluator.

4 subintervals per binade are sufficient. Thus, the number of memory entries is $16 \cdot 8 + 92 \cdot 4$, added with 16 on the (L) branch, which totals exactly 512 (the sweet-spot for the memory block utilization). This architecture (SP-HFP-d3-C-e110 in Tab. 5 and 6) has a $\max_\sigma = 12.06$.

Similarly to the optimized degree-2 implementation, the full exponent range can also be supported. For the exponent range $\{-111, -126\}$ a non-uniform segmentation scheme with 4 subintervals per binade is sufficient. This results in $16 \times 4 = 64$ memory entries, which is cheap to implement using LUT-6, an abundant resource in modern FPGA architectures. This is denoted by SP-HFP-d3-C-e126 in Tab. 5 and 6 and provides the full-range $\max_\sigma = 12.94$.

6.3 Generic architecture

A generic architecture is depicted in Figure 10. As proposed in Sec. 4.2, the computation is split in two branches: (L) with a uniform interval subdivision and (H) with the logarithmic-based interval subdivision.

For the (L) branch, the argument is firstly reduced as in the SP architecture. Then a number *bls* of subintervals are used. The addressing is done directly from the input, as before. For that, let *alw* be the number of fractional bits used. Then, when $x \in [0.5, 0.75)$ the address is obtained from the $[wF-2, wF-1-alw]$ bits of the input fraction, whereas for $x \in [0.25, 0.5)$ the bits $[wF-1, wF-alw]$ from the fixed-point difference $0.5-(mX \gg 2)$ are used (cf. Figure 7). Finally, the polynomial input is obtained by recovering the following $wF - 1 - alw$ of fX (when $x \geqslant 0.5$), or respectively bottom $wF - alw$ bits from $0.5 - (mX \gg 2)$. The parameter values employed in our higher precision cores are $alw = 4$ and $bls = 16$.

For Branch (H), let us focus on the exponent and the fraction contribution to the address. Since the address range for (H) starts at index *bls*, this offset needs to be added when computing the address, but is omitted in

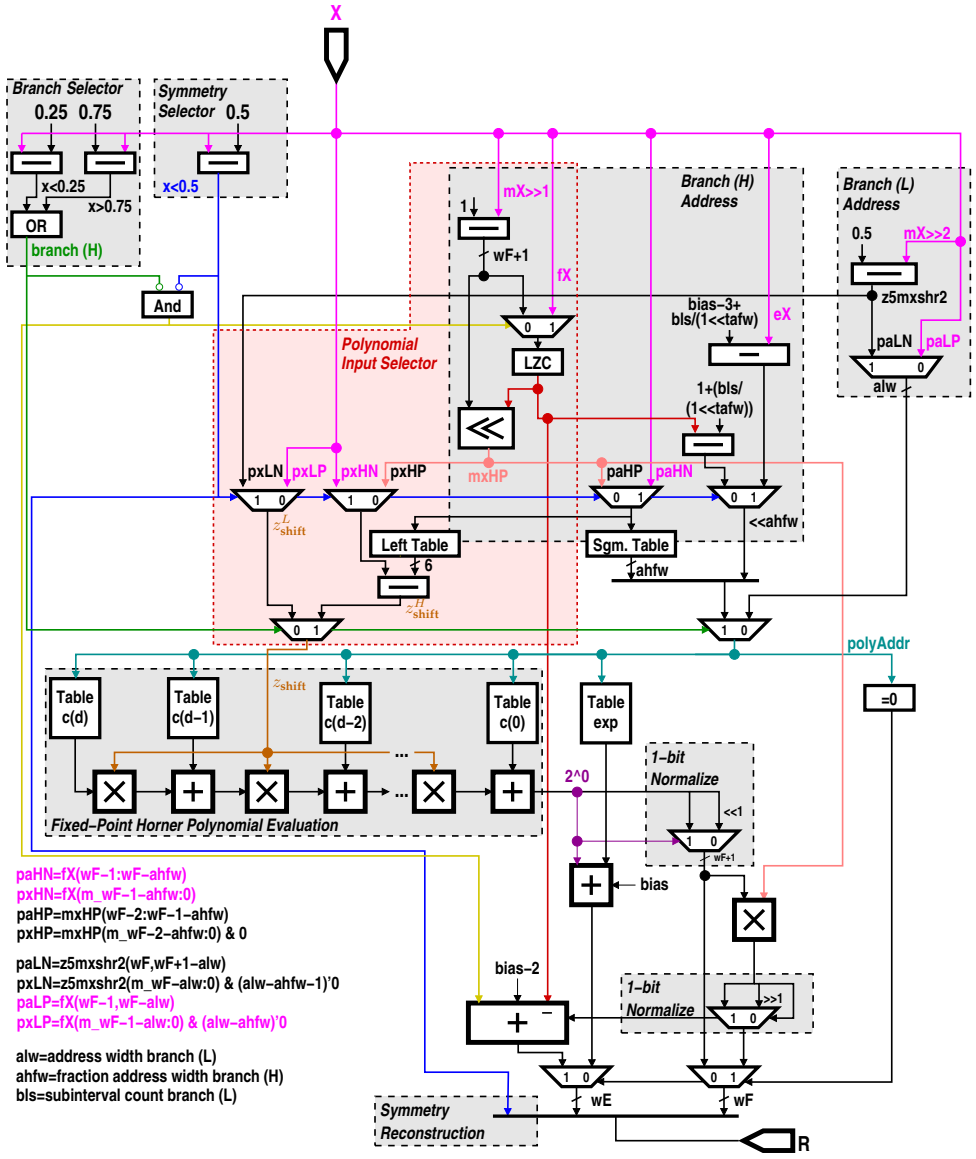


Figure 10: Generic architecture based on fixed-point piecewise-polynomial approximations

the following for simplicity. Firstly, the exponent contribution for $x < 0.25$ is obtained using $bias - 3 - eX$. For $x > 0.75$, the function input $1 - x$ is computed from a fixed-point subtraction with known alignment, equivalent to $0.5 - (mX \gg 2)$. The relative exponent, required for the address computation, is obtained by counting the leading zeros of the difference. Secondly, denote by *ahfw*, the number of fractional bits necessary for addressing the tables. When $x < 0.25$, these bits are obtained from the top of *fX*. When $x > 0.75$ the fixed-point difference $1 - x$ is normalized, by feeding the previously computed zero count together with the difference into a left shifter. The top *ahfw* bits of the resulting fraction are then used. For our cores, *ahfw*=3. Finally, the polynomial address is obtained by concatenating exponent and fractional parts.

For instance, to fill 512 table entries on (L)+(H), since $bls = 16$ is used for (L), and $ahfw = 3$ bits are required for the fractional part (H) (uniform segmentation for each binade), a total of $(512 - 16)/2^3 = 62$ exponents can be handled, which results in a 6 bits exponent addressing.

In Section 4.2 we have also described a more fine-grain selection of the subintervals corresponding to a binade, using a non-uniform mantissa segmentation. This is depicted in Figure 10, for our cores, and consists of using the top 6 fractional bits to index a 3-bit wide table (*Sgm. Table*) storing the corresponding new segment address, based on Table 2.

Then, the polynomial input is obtained from the bottom $wF - ahfw$ of *fX* for $x < 0.25$ (and respectively those of the normalized difference $1 - x$, when $x > 0.75$). Furthermore, as mentioned in Sec. 5 *(b)* $z_{\text{shift}} = z - l$ (l for "left" interval bound) is needed for the evaluation: an additional 6-bit wide LUT6 (*Left Table*) stores l and the subtraction is in fixed-point. Note that its result can be 1-bit wider in the case of the non-uniform mantissa segmentation.

Fused fixed-point polynomial evaluator. To create a single polynomial evaluator, the worst case of formats across the entire set of coefficients has to be considered. For our cores, they are presented in Table 4. Note that, as explained in Sec. 5 *(b)*, a final multiplication is performed outside the fused polynomial evaluator. Moreover, since the evaluator's output in $[0.5, 2)$, a single-bit normalization is required. The final exponent is recovered function of this bit and an additional stored relative exponent for each polynomial.

Several generic architectures were generated:

– to compare with the HFP-enabled SP architectures, a single precision architecture with fixed-point evaluation is proposed. Denoted by SP-FXP-d3-C-e64, this architecture is based on degree-3 polynomial approximations, features the custom segmentation presented above and handles

Table 4: Polynomial coefficient formats: signed(width,fraction). Number of polynomials is 512. Approximation accuracy 1ulp.

wF	**deg.**	**Coefficients Formats: Fused (L)+(H)**
26	4	(31,30), ±(38,32), (31,29), ±(30,25), (27,22)
35	5	(40,39), ±(47,41), (40,38), ±(39,34), (35,30), ±(34,27)
52	8	(57,56), ±(64,58), (57,54), ±(56,51), (52,47), ±(51,43), (49,41), ±(47,36), (43,32)

exponents $\{-3, -64\}$ on the (H) branch.

– a double precision architecture with fixed-point evaluation denoted by DP-FXP-d8-C-e64 is based on degree-8 polynomial approximations, features custom segmentation and handles exponents $\{-3, -64\}$ on the (H) branch.

– two intermediary custom formats, with the same dynamic range as double, but less precision (wF=26 and respectively, wF=35) are explored via the fixed-point evaluation generic architecture. Denoted by CP-FXP-d4-C-e64 (and respectively CP-FXP-d5-C-e64), they are based on degree-4 (respectively degree-5) polynomial approximations, feature custom segmentation and handle exponents $\{-3, -64\}$ on the (H) branch.

7 Results

In this article we proposed a family of architectures offering trade-offs between resource utilization and the tail accuracy $\max_\sigma$. The main features of all proposed architectures are summarized in Table 5. The synthesis results are presented in Table 6. These were obtained using Quartus 19.3.0, targeting Intel Arria 10, fastest speedgrade.

Firstly, let us analyze the obtained results for the single-precision case. For degree-2 polynomial approximations, the custom segmentation techniques always improve $\max_\sigma$. For the basic configuration (SP-HFP-d2-U-e30 vs SP-HFP-d2-C-e54) $\max_\sigma$ is improved from 6 to 8.3, for only 4 ALM (2% of ALM count). In an enhanced configuration (SP-HFP-d2-U-e62 vs SP-HFP-d2-C-e118) when an additional memory is used for both architectures, $\max_\sigma$ is improved from 8.61 to 12.51, which is near the full SP range;

the cost of this improvement is exactly 10 ALMs, or about 4% the ALM count. Finally, when comparing the two architectures at $\max_\sigma = 12.94$ (SP-HFP-d2-U-e126 vs SP-HFP-d2-C-e126) the custom segmentation version increases the ALM count by 35, but saves 5 M20Ks blocks - a very good trade-off to be made in most cases.

A similar trend is observed for the degree-3 SP HFP-based implementation: in the default configuration (SP-HFP-d3-U-e94 vs SP-HFP-d3-C-e110) the custom segmentation version improves $\max_\sigma$ from 11.18 to 12.06, while also reporting fewer ALMs (this reduction is mostly due to other technology mapping optimizations that we have implemented for SP-HFP-d3-C-e110). An additional 72 ALMs are reported for a configuration that offers full $\max_\sigma$ coverage (SP-HFP-d3-C-e126).

Moreover, when HFP DSPs are not used, our SP implementation SP-FXP-d3-C-e64 uses degree-3 polynomial approximation and is based on the generic architecture, cf. Sec. 6.3. Compared to the previously mentioned HFP-based SP architectures, it is slightly less efficient by most metrics. For instance, SP-HFP-d3-C-e126 outperforms it in every single metric - from latency, ALM count, DSP and M20K to $\max_\sigma$. This confirms the efficiency of specializing the architectures on devices with HFP capabilities.

Compared to previous approaches for SP, the most relevant implementation of the FP Probit function is [9]. For comparable $\max_\sigma$, our proposed architectures outperforms [9], especially in terms of logic utilization. Since [9] targets older Virtex-II devices, the 185MHz reported frequency is expected to scale up when the design is ported to recent FPGA devices. Moreover, our architectures are accurate to 3 ulps, whereas [9] reports 20 fractional bits of accuracy, which translates to 8 ulps.

Also, Wichura's algorithm was implemented using Intel DSP Builder Advanced [3] for both the single and double-precision cases (see Fig. 3 for the SP architecture). For SP architectures which have comparable tail accuracy to Wichura's, our proposed cores outperform the Wichura's one, despite the availability of FP DSP Blocks.

Beyond single, we have not found any prior works, therefore, our only comparison point is our adaptation of Wichura to these custom formats (all internal operations performed in the (wE,wF) format). The degree 7 rational polynomial approximation can likely be reduced for (11,26) and (11,35) so the Wichura results for these two formats could potentially be improved. It is clear however that in terms of resource utilization and latency, our proposed architectures (CP-FXP-d4-C-e64 and CP-FXP-d5-C-e64) will significantly outperform the Wichura adaptations. However, we chose to limit our architecture to $\max_\sigma = 9.08$, with 3 ulp relative accuracy (which seems reasonable in several applications [15]), whereas the

Table 5: Summary of main features of the proposed architectures. From left to right, the columns represent: architecture name; the approximation degree d; the segmentation type and total number of subintervals used on the (L) and (H) branches; the minimum exponent supported; the tail accuracy $\max_\sigma$; the corresponding circuitry/figure (when available).

Name	**Deg.** d	**Segmentation**				**Min exp** $\log_2 x_{\min}$	$\max_\sigma$	**Circuitry/ Figure**
		(L) $[0.25, 0.5)$		**(H)** $[0 + x_{\min}, 0.25)$				
		Type	#Interv.	Type	#Interv.			
SP-HFP-d2-U-e30	2	Uniform	128	Log	896	-30	6	Fig. 6
SP-HFP-d2-U-e62					1408	-62	8.92	Fig. 6, PCBH-A_1
SP-HFP-d2-U-e94					1920	-94	11.11	Fig. 6, PCBH-A_2
SP-HFP-d2-U-e126					2176	-126	12.94	Fig. 6, PCBH-A_2+B
SP-HFP-d2-C-e54	2	Semi-Coarse Greedy	55	Custom Log	960	-54	8.3	Fig. 8
SP-HFP-d2-C-e118					1472	-118	12.51	Fig. 8, PCBH-A
SP-HFP-d2-C-e126					1536	-126	12.94	Fig. 8, PCBH-A+B
SP-HFP-d3-U-e94	3	Uniform	16	Log	496	-94	11.11	Fig. 9
SP-HFP-d3-C-e110				Custom Log	496	-110	12.06	
SP-HFP-d3-C-e126					560	-126	12.94	
SP-FXP-d3-C-e64	3	Uniform	16	Custom Log	496	-64	9.08	Fig. 10
CP-FXP-d4-C-e64	4							
CP-FXP-d5-C-e64	5							
DP-HFP-d6-C-e64	8							

Wichura algorithm has full tail accuracy.

8 Conclusion

In this work we have proposed two sets of architectures for the FP Probit function: (a) for SP targeting the HFP DSP Blocks, and (b) a generic architecture based on a fixed-point polynomial evaluation kernel that can be implemented for any custom FP format. On one hand, it was shown that our proposed architectures both outperform existing FP SP works in terms of resource utilization for comparable tail accuracy, but also provide a level of customization regarding the tail accuracy $\max_\sigma$ that results in a resource-utilization tradeoff - potentially exploitable at application level. On the other hand, proposed generic parametrizable architectures work for custom FP formats with a tail accuracy of $\max_\sigma = 9.08$. These have a low resource utilization for double-precision compared to an FPGA implementation of the Wichura algorithm. This is due to the proposed custom segmentation scheme, which "mimics" the asymptotic behavior and the corresponding Wichura's change of variable. For instance, for the double precision implementation, this allowed for a reduction of the polynomial degree by 1 (and thus a 10% resources saving), compared to a classical

Table 6: Complete synthesis results for the proposed cores targeting Intel Arria 10 FPGAs, fastest speedgrade. Results reported for [9] correspond to Xilinx Virtex-II FPGAs.

wE, wF	**Algorithm**	**Lat.**	**Resource Utilization**					$\max_\sigma$
			ALMs	Regs	DSPs	M20K	FMax	
8, 23	Divide	17	206	625	3	3	549MHz	-
	Sqrt	11	101	309	2	3	530MHz	-
	Log	26	321	842	8	3	483MHz	-
	Wichura	87	1134	3108	25	10	483MHz	12.94
	[9]	55	*2022*		*15*	*5*	*185MHz*	6.23
	SP-HFP-d2-U-e30	18	225	590	3	6	483MHz	6
	SP-HFP-d2-C-e54	18	229	584	3	6	483MHz	8.3
	SP-HFP-d2-U-e62	18	263	607	3	9	483MHz	8.61
	SP-HFP-d2-C-e118	18	273	658	3	9	483MHz	12.51
	SP-HFP-d2-U-e94	18	270	608	3	12	483MHz	10.86
	SP-HFP-d2-U-e126	18	324	547	3	14	483MHz	12.94
	SP-HFP-d2-C-e126	18	359	532	3	9	483MHz	12.94
	SP-HFP-d3-U-e94	23	329	658	4	4	483MHz	11.18
	SP-HFP-d3-C-e110	23	264	680	4	4	483MHz	12.06
	SP-HFP-d3-C-e126	23	336	743	4	4	483MHz	12.94
	SP-FXP-d3-C-e64	30	453	1293	5	5	481MHz	9.08
11, 26	CP-FXP-d4-C-e64	34	532	1427	7	6	481MHz	9.08
	Wichura	206	8000	18928	26	17	446MHz	-
11, 35	CP-FXP-d5-C-e64	55	1115	2878	13	8	549MHz	9.08
	Wichura	291	13398	31115	42	20	449MHz	-
11, 52	DP-FXP-d8-C-e64	87	2797	7855	36	21	474MHz	9.08
	Wichura	351	18574	47389	83	45	392MHz	37.51
	Divide	38	888	3055	11	11	549MHz	-
	Sqrt	33	674	2210	8	8	549MHz	-
	Log	51	1500	4311	11	20	475MHz	-

logarithmic segmentation. Further tuning of similar custom segmentation schemes, allows for an improved architecture for the SP case: either the memory count reduces by 35% for an equivalent tail accuracy, or for a similar cost, the tail accuracy increases by 38%. Another feature is that the proposed architectures are sufficiently generic, such that higher $\max_\sigma$ can easily be obtained by choosing a different polynomial degree and/or number of subintervals. We intend to further explore the argument reduction techniques by analyzing the trade-off between pure piecewise polynomial approximations and composite ones, which make some intermediary use of the asymptotic behavior.

References

[1] Intel Arria®10 Device Overview (2018). https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/arria-10/a10_overview.pdf

[2] IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) pp. 1–84 (2019)

[3] DSP Builder for Intel FPGAs (Advanced Blockset) Handbook (2021). https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/dspb/hb_dspb_adv.pdf

[4] Cheung, R.C.C., Lee, D., Luk, W., Villasenor, J.D.: Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method. IEEE Transactions on VLSI Systems **15**(8), 952–962 (2007)

[5] Chevillard, S., Joldeş, M., Lauter, C.: Sollya: An environment for the development of numerical codes. In: International Congress on Mathematical Software, pp. 28–31. Springer (2010)

[6] Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Transactions on Mathematical Software (TOMS) **37**(1), 1–20 (2010)

[7] De Dinechin, F., Joldes, M., Pasca, B.: Automatic generation of polynomial-based hardware architectures for function evaluation. In: IEEE International Conference on Application-specific Systems, Architectures and Processors, pp. 216–222. IEEE (2010)

[8] De Schryver, C., Schmidt, D., Wehn, N., Korn, E., Marxen, H., Korn, R.: A new hardware efficient inversion based random number generator for non-uniform distributions. In: Intl. Conf. on Reconfig. Comp. and FPGAs, pp. 190–195. IEEE (2010)

[9] Echeverria, P., López-Vallejo, M.: FPGA gaussian random number generator based on quintic Hermite interpolation inversion. In: 2007 50th Midwest Symposium on Circuits and Systems, pp. 871–874. IEEE (2007)

[10] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: A multiple-precision binary floating-point library with correct rounding. ACM Transactions on Mathematical Software (TOMS) **33**(2) (2007)

[11] Hörmann, W., Leydold, J.: Continuous random variate generation by fast numerical inversion. ACM Trans. Model. Comput. Simul. **13**(4), 347–362 (2003)

[12] Joldes, M., Pasca, B.: Efficient floating-point implementation of the probit function on FPGAs. In: 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 173–180 (2020)

[13] Lee, D.U., Cheung, R.C., Villasenor, J.D., Luk, W.: Inversion-based hardware gaussian random number generator: A case study of function evaluation via hierarchical segmentation. In: 2006 IEEE International Conference on Field Programmable Technology, pp. 33–40. IEEE (2006)

[14] Luu, T.: Fast and accurate parallel computation of quantile functions for random number generation. Ph.D. thesis, UCL (University College London) (2016)

[15] Malik, J.S., Hemani, A.: Gaussian random number generation: A survey on hardware architectures. ACM Comput. Surv. **49**(3) (2016)

[16] Muller, J.M., Brunie, N., de Dinechin, F., Jeannerod, C.P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser (2018)

[17] Revol, N.: Interval Newton iteration in multiple precision for the univariate case. Num. Alg. **34**(2-4), 417–426 (2003)

[18] Thomas, D.B.: A general-purpose method for faithfully rounded floating-point function approximation in FPGAs. In: 2015 IEEE 22nd Symposium on Computer Arithmetic, pp. 42–49 (2015)

[19] Thomas, D.B., Luk, W., Leong, P.H., Villasenor, J.D.: Gaussian random number generators. ACM Comput. Surv. **39**(4) (2007)

[20] Wichura, M.J.: Algorithm as 241: The percentage points of the normal distribution. Journal of the Royal Statistical Society. Series C (Applied Statistics) **37**(3), 477–484 (1988)