Efficient Floating-Point Implementation of the Probit Function on FPGAs

Mioara Joldes, 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 [14, Chap. 8], [16]. 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 $\Phi : \mathbb{R} \to [0, 1]$,

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

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 Fig. 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: $\max_\sigma = |f(x_{\min})|$.

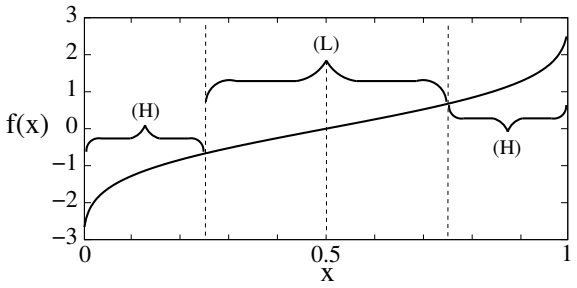


Figure 1: Probit Function

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 [17, 13] 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 [18] 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 [12] 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. [3] 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 [8] adapt to hardware the software-based FP implementation from [10]. 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. [7]. It uses a hierarchical segmentation from [3] adapted to the FP format. Unlike [8], 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.

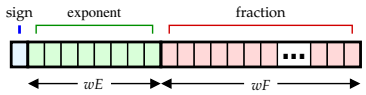
– concerning higher precision formats, a generic implementation strategy based on fused fixed-point piecewise polynomial approximation is proposed. It is applied for generating efficient architectures for three floating-point formats, including double precision.

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. Piecewise polynomials, together with a generic fused polynomial evaluator, provide hardware implementations accurate to 3 ulps. The features of these cores are detailed in Section 4. Finally, the synthesis results are discussed and compared in Section 5.

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 used by our architectures are $wE = 11$, $wF = 26$ and respectively $wE = 11$, $wF = 35$.

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 [14, 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 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 Fig. 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, on which we particularly focus in the sequel.

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 [18] and related methods [12] 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 Fig. 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 $z < 0$, one has, $\Phi(z) \leqslant B(z)$, with $B(z) := -\frac{e^{-\frac{z^2}{2}}}{z\sqrt{2\pi}}$ and, $\lim\limits_{z\to-\infty} \frac{B(z)}{\Phi(z)} = 1$, which states that for sufficiently large $|z|$, for negative z, $B(z)$ is a good approximation for $\Phi(z)$. Hence, to solve $\Phi(z) = x$, one can consider $B(z) \approx x$, take the natural logarithm $\log$ and obtain by rewriting:

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

Thus, for $z \ll 0$,

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

Eq. (6) provides a means for computing an approximation for f near its asymptotics. As an example, Fig. 2 shows the linearization effect of such

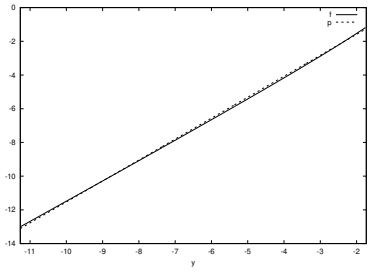


Figure 2: Plot of $f(x)$, on a "$-\sqrt{-\log_2(x)}$"-*linear scale*, for $x \in [2^{-127}, 2^{-3}]$; a linear approximation P with the Wichura's variable change: $P(y) := P\left(-\sqrt{-\log_2(x)}\right)$.

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 [18] 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 [18] tail breakpoint for performing the change of variable is $x < b = .075$. After the variable change, Wichura [18] 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.

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 4. For double-precision (last 4 rows of Table 4), 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 3.3 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. Fig. 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 [11] or quintic Hermite interpolations [8]. We consider the minimax constrained-coefficients polynomial approximations provided by the Sollya software tool [4]. 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 [14, Chap. 10], [6, 16]. 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 [5].

Unfortunately, the probit function was not implemented in Sollya. However, Sollya provides arbitrary precision, as well as certified computations (interval arithmetic) for erf (and erfc). Hence, we developed an arbitrary precision faithfully-rounded implementation[2] for the probit in Sollya based on inverting the erfc function, with interval Newton method [15], as follows.

Our arbitrary precision implementation of the probit function in Sollya is based on solving for z the equation $\Phi(z) - x = 0$, which is equivalent to $T(x) = 0$ where T and its derivative with respect to z are:

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

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

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

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

1: $T \leftarrow \mathrm{erfc}\left(\frac{z}{\sqrt{2}}\right) - 2 + 2x$
2: **while** `intervalDiam`$(\mathcal{I})/2 \leqslant \varepsilon$ **do**
3: $\quad z_0 \leftarrow$ `intervalMidpoint`$(\mathcal{I})$
4: $\quad \mathcal{T}' \leftarrow$ `evaluate`$(T', \mathcal{I})$
5: $\quad \mathcal{Z} \leftarrow$ `evaluate`$(z_0 - T(z_0)/\mathcal{T}')$
6: $\quad \mathcal{I} \leftarrow$ `intervalIntersect`$(\mathcal{Z}, \mathcal{I})$
7: **end while**
8: **return** `intervalMidpoint`$(\mathcal{I})$

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 ε[3], 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 [15]. This algorithm was also easily coded in C based on the MPFR library [9] 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. We consider an approach similar to hierarchical segmentation, which was already used in [3, 7, 16] in both the fixed-point and floating-point setting.

3.3 Logarithmic subdivision

Let us focus first 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 con-

[3]The absolute accuracy test in line 2 can be made relative by dividing with z_0, provided that $z_0 \neq 0$

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
$[-35,\ldots,-93]$	2
$[-94,\ldots,-126]$	1

(b) Degree 3

catenating 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.

The remaining (L) range $[0.25, 0.5)$ is handled in both cases by uniform segmentation: 128 subintervals for degree-2 polynomials and 16 subintervals for degree-3.

For a double-precision implementation we propose degree 8 minimax polynomials as a good compromise: on the "nice" (L) input range $x \in [2^{-2}, 2^{-1})$ we proceed to a uniform subdivision as in the single precision case, with 16 subintervals. Then, 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 performing a non-uniform mantissa subdivision per binade, which is explained by an example.

Example of non-uniform mantissa subdivision. 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 $2^{1/8}$, $2^{2/8}, \ldots, 2^{7/8}, 1$. 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 segment approximations, whose bounds corre-

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

spond to a_F from the nearest rounding of $2^{i/8}$ 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.

3.4 Efficient polynomial fixed-point and FP evaluations

When generating the polynomial approximations, several argument and function scaling techniques are required for efficiency.

FP evaluation. It is preferred for SP, to take advantage of the HFP-DSP. On the (L) domain, the reduced input argument $z = 2|0.5 - x| \in [0, 0.5]$ 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 by uniform subdivision. For (H), which proceeds by binade $x \in [2^e, 2^{e+1})$, potential overflow in the polynomial coefficients for high magnitude e, is avoided by rescaling the polynomial input to $z = x/2^e$:

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

where I will be some subinterval of $[1, 2)$. To obtain z, this simply translates to concatenating a new sign (0) and 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. For the input range, the same ideas as in the FP case are employed, together with a further shifting and scaling. Specifically, for (L), when $z \in [l, r] \subset [0, 0.5)$, the evaluation is performed in $z_{\text{shift}} = z - l$, which translates for instance to $0 \leqslant z_{\text{shift}} < 2^{-5}$, when the input

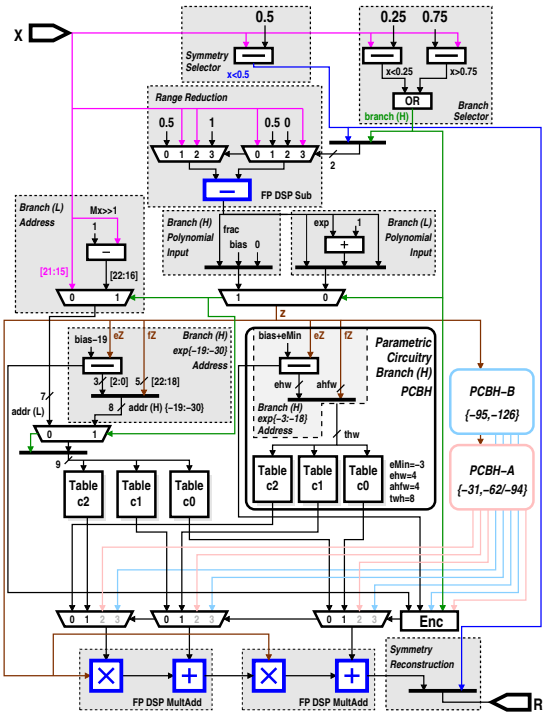


Figure 3: SP architecture of FP probit for HFP-enabled FPGAs.

is split in 16. A similar argument shift is performed for (H), which 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.

4 Architecture

4.1 Single Precision - degree 2

A SP architecture targeting HFP DSP-Enabled FPGAs is presented in Fig. 3. 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. 3.4 *(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)

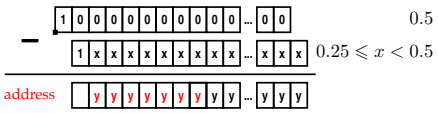


Figure 4: Fixed-point alignment for (L) branch address computation, when $x < 0.5$. The operation is a $2 + wF$-bit subtraction.

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 the fixed-point difference $0.5 - x$; the alignment of both terms is known, as shown in Fig. 4.

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 (see Sec. 3.3) technique, 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 (†) in Table 4 is $\max_\sigma = 6$.

The handled exponent range can be further increased by adding auxiliary circuitry PCBH-A, which itself has two configurations, denoted by (‡) and (‡‡) in Tab. 4, with tail accuracy of $\max_\sigma = 8.92$ or $\max_\sigma = 11.11$:

– *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.

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, corresponding to -126 ($\max_\sigma = 12.94$, denoted by ($\diamond$) in Tab. 4). 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 3. 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.

4.2 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 5.

As presented in Sec 3.3, (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 -93. First, for exponent range {-3,-34} each exponent requires 8 subintervals but for the range {-35, -93} only 4 subintervals suffice for meeting the approximation error objective.

The addressing is detailed in Fig. 5 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.

4.3 Generic architecture

A generic architecture is depicted in Fig. 6. As proposed in Sec. 3.3, 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 frac-

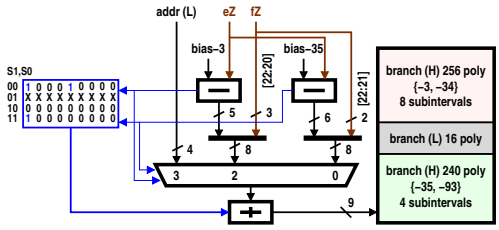


Figure 5: Coefficient memory composition and addressing for a SP architecture based on a degree 3 polynomial evaluator.

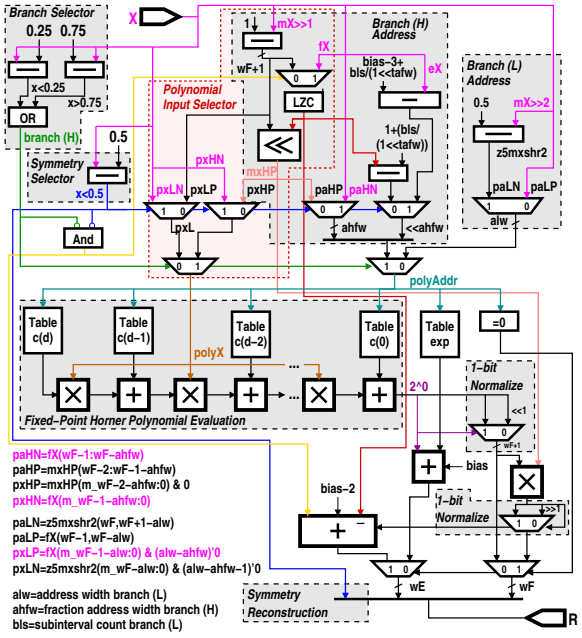


Figure 6: Generic architecture based on fixed-point piecewise-polynomial approximations

tional 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. Fig. 4). 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 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 3.3 we have also described a more fine-grain selection of the subintervals corresponding to a binade, using a non-uniform mantissa segmentation. Although not depicted in Fig. 6, for our cores, this consists of using the top 6 fractional bits to index a 3-bit wide 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. 3.4 *(b)* $x - l$ (l for "left" interval bound) is needed for the evaluation: an additional 6-bit wide LUT6 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 3. Note that, as explained in Sec. 3.4 *(b)*, a final multiplication is performed outside

Table 3: 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)

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.

5 Results

The synthesis results for our proposed architectures are presented in Table 4. These were obtained using Quartus 19.3.0, targeting Intel Arria 10, fastest speedgrade. First, for SP the most relevant implementation of the FP Probit function is [8]. We propose a family of architectures offering trade-offs between resource utilization and the tail accuracy $\max_\sigma$. For comparable $\max_\sigma$, our proposed architecture outperforms [8], especially in terms of logic utilization. Moreover, our architectures are accurate to 3 ulps, whereas [8] reports 20 fractional bits of accuracy, which translates to 8 ulps.

Wichura's algorithm was implemented using Intel DSP Builder Advanced (both the single and double-precision). For SP architectures, which have comparable tail accuracy with Wichura, our proposed cores outperform the Wichura's one, despite the availability of FP DSP Blocks. Moreover, our SP implementation $d = 3, \triangle$ based on the generic architecture, which does not use HFP DSPs, is less efficient but remains a good choice on devices without HFP capabilities.

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 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 applica-

Table 4: Synthesis results for the proposed cores.

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
	[8]	55	2022		15	5	185MHz	6.23
	Ours d=2 †	18	225	590	3	6	483MHz	6
	Ours d=2 ‡	18	263	607	3	9	483MHz	8.61
	Ours d=2 ‡‡	18	270	608	3	12	483MHz	10.86
	Ours d=2 ⋄	18	324	547	3	14	483MHz	12.94
	Ours d=3	23	329	658	4	4	483MHz	11.18
	Ours d=3 △	30	453	1293	5	5	481MHz	9.08
11, 26	Ours d=4	34	532	1427	7	6	481MHz	9.08
	Wichura	206	8000	18928	26	17	446MHz	-
11, 35	Ours d=5	55	1115	2878	13	8	549MHz	9.08
	Wichura	291	13398	31115	42	20	449MHz	-
11, 52	Ours d=8	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	-

tions [13]), whereas the Wichura algorithm has full tail accuracy.

6 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 we have showed 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 logarithmic segmentation. 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. This includes non-uniform segmentation schemes for all architectures, as well as 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)*, pages 1–84, July 2019.

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

[4] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. In *International Congress on Mathematical Software*, pages 28–31. Springer, 2010.

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

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

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

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

[9] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM TOMS*, 33(2), 2007.

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

[11] D.-U. Lee, R. C. Cheung, J. D. Villasenor, and W. Luk. 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*, pages 33–40. IEEE, 2006.

[12] T. Luu. *Fast and accurate parallel computation of quantile functions for random number generation*. PhD thesis, UCL (University College London), 2016.

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

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

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

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

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

[18] M. J. Wichura. 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.