Floating-Point Exponentiation Units for Reconfigurable Computing

FLORENT DE DINECHIN
PEDRO ECHEVERRÍA and MARISA LOPEZ-VALLEJO
BOGDAN PASCA

The high performance and capacity of current FPGAs makes them suitable as acceleration co-processors. This article studies the implementation, for such accelerators, of the floating-point power function \( x^y \) as defined by the C99 and IEEE 754-2008 standards, generalized here to arbitrary exponent and mantissa sizes. Last-bit accuracy at the smallest possible cost is obtained thanks to a careful study of the various subcomponents: a floating-point logarithm, a modified floating-point exponential, and a truncated floating-point multiplier. A parameterized architecture generator in the open-source FloPoCo project is presented in details and evaluated.

Categories and Subject Descriptors: B.2.4 [Arithmetic and Logic Structures]: High-Speed Arithmetic

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Reconfigurable Computing, floating-point, power function, exponentiation unit

1. INTRODUCTION

1.1. Floating Point Acceleration Using FPGAs

Current deep submicron technologies brought FPGAs with extraordinary logic density and speed [Kuon and Rose 2007]. They can be used to implement complex applications, including floating-point ones [Scrofano et al. 2008; Underwood 2004; Woods and VanCourt 2008; Zhuo and Prasanna 2008]. This has sparked the development of floating-point units targeting FPGAs, first for the basic operators mimicking those found in microprocessors [Belanović and Leeser 2002; Louca et al. 1996; Shirazi et al. 1995], then more recently for operators which are more FPGA-specific, for instance accumulators [Bodnar et al. 2006; de Dinechin et al. 2008; Luo and Martonosi 2000; Wang et al. 2006] or elementary functions [de Dinechin and Pasca 2010; Detrey and de Dinechin 2007; Doss and Riley 2004; Piñeiro et al. 2004].

The FloPoco project\(^1\) aims at providing high-quality, portable and open-source operators for floating-point computing on FPGAs. This article describes the implementation

\(^1\)http://flopoco.gforge.inria.fr/
of an operator for the power function \( x^y \) in this context. This function appears in many scientific or financial computing kernels [Echeverría Aramendi 2011; Rebonato 2002].

### 1.2. The Power Function and Its Floating-Point Implementation

The power function is classically defined as \( x^y \), and its usual floating-point implementation was included as the `pow` function of the C99 standard. For positive \( x \), the value of \( x^y \) can be computed thanks to the equation:

\[
x^y = e^{y \cdot \ln x}.
\]

For \( x < 0, x^y \) is still defined for integer \( y \), although not through the previous formula: in this case

\[
x^y = (-1)^y \cdot |x|^y = (-1)^y e^{y \cdot \ln |x|}.
\]

The value one wants to assign to special cases such as \( 0^0 \) depends on the context. On the one hand, one may consider that this \( y = 0 \) is an integer. Then \( 0^0 \) will naturally occur in polynomial evaluation, for instance, and its value should be defined by continuity as 1. On the other hand, if one considers that \( y = 0 \) is the limit of some real series converging to 0, then the value of \( 0^0 \) can be anything, and should be left undefined.

The C99 `pow` function was a necessarily inconsistent tradeoff between the requirements of these different contexts. Its behaviour is summarized in Table I: `pow(x,y)` is defined for \( x \) and \( y \) real numbers, and special handling is provided for negative \( x \) in combination with integer \( y \).
The 2008 revision of the floating-point standard IEEE 754 [IEEE Computer Society 2008] had to keep this pow function for backward compatibility, but provided a way to avoid its inconsistencies by offering in addition the following two cleaner and more consistent functions.

— pown(x,n) is defined only for integer n. This is the function that should be used in polynomial evaluators for instance. The standard also includes rootn(x,n) defined as $x^{1/n}$.

— powr(x,y) is defined only by Equation (1), and in particular is undefined for negative x.

This work covers the two IEEE 754-2008 functions whose special cases are presented in Table I: the traditional pow function inherited from C99, the more modern powr function, which is actually simpler to implement. Note however that these implementations are not strictly speaking IEEE 754-compliant, since they operate on the FPGA-specific floating-point formats presented in Section 2.1.

1.3. Previous Work

As the pow function is defined by the C99 standard, most mathematical libraries (libm) provide an implementation of this function, based on Equations (1) and (2). A detailed description of last-bit accurate implementations in single, double, double-extended and quadruple precisions is given in Markstein's book [Markstein 2000]. The interested reader will also find accurate open-source implementations in the libultim library included in the GNU glibc, in the CRLibm project, or in the MPFR library.

Concerning hardware implementations, [Harris 2004] presents an implementation of (1) targeted to OpenGL lightning applications: it inputs and outputs single-precision floating-point numbers, but the inputs have a restricted range and the output actually wraps a low-accuracy (8 bits) fixed-point result. A detailed error analysis based on this specification enables a low area, table-based implementation of both exponential and logarithm. However, this work is very application-specific. A more generic implementation is presented in Piñeiro et al. [2004]. It is also based on (1), where exponential and logarithm are evaluated by high-radix iterative algorithms using redundant number representations. It is unclear whether this work was implemented, and the choices made (in particular the use of redundant arithmetic) are more suited to VLSI than FPGA, where both efficient additions (through fast-carry lines) and efficient multiplications (through embedded multipliers and DSP blocks) are available. More recently, some of us [Echeverría and López-Vallejo 2008] presented an $x^y$ operator for single precision. However, that implementation only partially handled the error propagation and was therefore inaccurate for some inputs. Moreover, the architecture was too specifically tuned to single precision.

As Figure 1 shows, Equation (1) has a straightforward translation into an architecture. Several hardware implementations of exponential [Altera 2008a; de Dinechin and Pasca 2010; Detrey and de Dinechin 2007; Detrey et al. 2007; Doss and Riley 2004; Ercegovac 1973; Pottathuparambil and Sass 2009; Vázquez and Antelo 2003; Wielgosz et al. 2009; Wrathall and Chen 1978] and logarithm [Altera 2008b; de Dinechin 2010; Detrey and de Dinechin 2007; Detrey et al. 2007; Ercegovac 1973; Wrathall and Chen 1978] have been published. Some are easily available: two versions of the FPLibrary
operators released in 2005 [Detrey and de Dinechin 2007] and 2007 [Detrey et al.
2007] respectively, the Altera megafunctions since 2008 [Altera 2008a, 2008b], and
the FloPoCo operators since 2009 [de Dinechin 2010; de Dinechin and Pasca 2010].
In this work, we use the latest because they alone offer the unique combination of
features required for this work: they scale to double precision, they are pipelined, and
they are open-source; we will need to modify both the exponential and the multiplier
to implement pow and powr efficiently.

1.4. Outline and contribution
This article shows how to build an accurate architecture for powr and pow for any preci-
sion. First, an in-depth analysis of Equation (1) in a floating-point context is presented
in Section 2. This analysis is parameterized by the floating-point format, and has been
integrated in the FloPoCo tool. The architecture of the operator, built upon existing ex-
ponential, logarithm and multipliers, is presented in Section 3. We need to modify the
existing exponential and use a truncated rectangular multiplier. A brief discussion of
the impact of several design decisions is provided. Section 4 evaluates this architecture
for various precisions. Section 5 concludes.

An important contribution of this work is an open-source implementation: the in-
terested reader may reproduce or extend our results, and find any missing technical
details, using the FloPoCo distribution, starting from version 2.3.0.

2. RANGE AND ERROR ANALYSIS
In floating-point arithmetic, the result of an operation (here \( r = x^y \)) must be rounded
to the target format. Rounding the infinitely accurate result is called correct round-
ing, and entails a maximum error of 0.5 ulps, where the ulp (unit in the last place)
is the weight of the least significant bit of the result. For an elementary function, en-
suring this error bound requires evaluating the function on a very large intermediate
precision, typically twice the result precision \( f \) [Muller 2006]. A solution, sometimes
used in software, is to compute with such precision only when needed [de Dinechin
et al. 2007; Ziv 1991]. This solution does not allow a fixed-latency hardware imple-
mentation. Therefore (and in line with most software implementations) our goal will
be to allow a slightly larger error, with an error bound that remains smaller than 1
ulp. This is called faithful rounding. From another point of view, our hardware oper-
ator shall return one of the two floating-point numbers surrounding the exact result.
This error bound also ensures that if the exact result is a floating-point number (which

Fig. 1. Simplified overview of a power function unit for an input floating-point format \((r,f)\) where \( r \) is the
size in bits of the exponent field (defining the range) and \( f \) is the size in bits of the fraction field (defining the
precision). The intermediate formats \((r_{\log},f_{\log})\) and \((r_p,f_p)\) must be determined to ensure last-bit accuracy
at the minimum cost.
indeed happens quite often for $x^y$ [Lauter and Lefèvre 2009]), then our operator will return it.

We first have to discuss the handling of special situations (overflow and underflow), which will define the intervals of possible values for the intermediate variables.

2.1. Floating-Point Formats

In FloPoCo, a floating-point number $x$ is composed of a sign bit $S$, an exponent field $E$ on $r$ bits, and a normalized significand fraction $F$ on $f$ bits. In addition, a two-bit exn code is used for exceptional cases such as zeroes, infinities and NaN (Not a Number). Figure 2 depicts such a number, whose value is $x = (-1)^S \times 1.F \times 2^{E-E_0}$ with $E_0 = 2^{r-1} - 1$.

This format is inspired by the IEEE 754 standard, but is not strictly compliant with it. This standard was designed for processor implementations and makes perfect sense there, but for FPGAs, many things can be reconsidered. Firstly, a designer should not restrict himself to the 32-bit and 64-bit formats of IEEE 754: he should aim at optimizing both exponent and significand size for the application at hand. The floating-point operators should be fully parameterized to support this.

Secondly, the IEEE 754 encodings were designed to make the most out of a fixed number of bits. In particular, exceptional cases are encoded in the two extremal values of the exponent. However, managing these encodings has a cost in terms of performance and resource consumption [Echeverría and López-Vallejo 2011]. In an FPGA, this encoding/decoding logic can be saved if the exceptional cases are encoded in two additional bits. This is the choice made by FloPoCo and other floating-point libraries. A small additional benefit is to free the two extremal exponent values, slightly extending the range of normal numbers.

Finally, we choose not to support subnormal numbers, with flushing to zero instead. This is the most controversial issue, as subnormals bring with them important properties such as $(x - y = 0) \iff (x = y)$, which is not true for FP numbers close to zero if subnormals are not supported. However the cost of supporting subnormals is quite high, as they require specific shifters and leading-one detectors [Echeverría and López-Vallejo 2011]. Besides, one may argue that adding one bit of exponent brings in all the subnormal numbers, and more, at a fraction of the cost: subnormals are less relevant if the format is fully parameterized. We believe there hasn’t been a clear case for subnormal support in FPGA computing yet.

2.2. Range Analysis

Let $a$ and $\omega$ be the smallest and largest positive floating-point numbers respectively. If the exact value of $x^y$ is smaller than $a$ or greater than $\omega$, we have to return $+0$ or $+\infty$ respectively (this is a constraint of the output format). In an implementation based on $x^y = e^{y \times \ln x}$, the result comes from the exponential function. This defines the useful range $[m_p, M_p]$ of the product $p = y \times \ln x$ which is input to this function:

$$m_p = \ln(a), \quad \text{rounded up}$$
$$M_p = \ln(\omega), \quad \text{rounded down}$$

For practical floating-point formats, this range is very small, as can be seen in Table II for single and double precision.
<table>
<thead>
<tr>
<th>Format</th>
<th>$m_p$</th>
<th>$M_p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>FloPoCo single precision (8,23)</td>
<td>-88.03</td>
<td>89.41</td>
</tr>
<tr>
<td>FloPoCo double precision (11,52)</td>
<td>-709.09</td>
<td>710.48</td>
</tr>
</tbody>
</table>

Analytically, we have $\omega < 2^{2^{r-1}}$ (this bound is valid both for the IEEE 754 formats and the FloPoCo format), therefore $M_p < 2^{r-1}$.

### 2.3. General Error Analysis

A last-bit accurate result corresponds to a bound on the total relative error of $\varepsilon_{total} = 1u$ where $u = 2^{-f}$ is the relative value of the ulp. Our goal is to ensure that $\varepsilon_{total} < \bar{\varepsilon}_{total}$. This total relative error of the operator is defined by

$$
\varepsilon_{total} = \frac{R - x^y}{x^y} = \frac{R}{x^y} - 1.
$$

(3)

We compute the power function using the formula $x^y = e^{y \times \ln x}$, implemented as three suboperators: a logarithm, a multiplication, and an exponential. Obviously, each error introduced by the suboperators must be kept under control in order not to harm the accuracy of the exponentiation unit.

We will denote $\text{mul}$, $\text{E}$, and $\text{Ln}$ the implementations of the multiplication, exponential and logarithm used here. They will entail the following relative errors:

$$
\varepsilon_{\text{mul}}(a, b) = \frac{\text{mul}(a, b)}{ab} - 1 \quad \text{with} \quad |\varepsilon_{\text{mul}}(a, b)| < \bar{\varepsilon}_{\text{mul}},
$$

(4)

$$
\varepsilon_{\text{exp}}(x) = \frac{E(x)}{e^x} - 1 \quad \text{with} \quad |\varepsilon_{\text{exp}}(x)| < \bar{\varepsilon}_{\text{exp}},
$$

(5)

and

$$
\varepsilon_{\text{log}}(x) = \frac{\text{Ln}(x)}{\ln x} - 1 \quad \text{with} \quad |\varepsilon_{\text{log}}(x)| < \bar{\varepsilon}_{\text{log}}.
$$

(6)

The purpose of this section is to define the relationship between $\varepsilon_{total}$, $\bar{\varepsilon}_{log}$, $\bar{\varepsilon}_{exp}$, and $\bar{\varepsilon}_{mul}$, and deduce from it the architectural parameters that will enable faithful rounding at a minimal cost.

Rewriting (3), we obtain

$$
\varepsilon_{total} = \frac{E(\text{mul}(y, \text{Ln}(x)))}{e^{y \times \ln x}} - 1.
$$

There, we have in the first operator

$$
\text{Ln}(x) = \ln(x)(1 + \varepsilon_{\text{log}}),
$$

while in the multiplier

$$
\text{mul}(y, \text{Ln}(x)) = y \text{Ln}(x)(1 + \varepsilon_{\text{mul}})
$$

$$
= y \ln(x)(1 + \varepsilon_{\text{log}})(1 + \varepsilon_{\text{mul}})
$$

$$
= y \ln(x)(1 + \varepsilon_{\text{log}} + \varepsilon_{\text{mul}} + \varepsilon_{\text{log}}\varepsilon_{\text{mul}})
$$

$$
= y \ln(x)
$$

$$
+ y \ln(x)(\varepsilon_{\text{log}} + \varepsilon_{\text{mul}} + \varepsilon_{\text{log}}\varepsilon_{\text{mul}}),
$$
and hence in the final exponential

\[ R = E(mul(y, \ln(x))) \]

\[ = e^{mul(y, \ln(x))}(1 + \epsilon_{\exp}) \]

\[ = e^{y\ln(x)} + y\ln(x)(\epsilon_{\log} + \epsilon_{\mul} + \epsilon_{\log\mul})(1 + \epsilon_{\exp}) \]

\[ = e^{y\ln(x)} \cdot e^{y\ln(x)(\epsilon_{\log} + \epsilon_{\mul} + \epsilon_{\log\mul})(1 + \epsilon_{\exp})} \]

and finally the total relative error corresponds to

\[ \epsilon_{\text{total}} = e^{y\ln(x)(\epsilon_{\log} + \epsilon_{\mul} + \epsilon_{\log\mul})(1 + \epsilon_{\exp})} - 1. \quad (7) \]

In Equation (7), all the terms in the exponential must be small compared to 1, otherwise the operator cannot be accurate. Therefore, using \( e^x \approx 1 + x + x^2/2 \), this equation becomes

\[ \epsilon_{\text{total}} = y\ln(x)\epsilon_{\log} + y\ln(x)\epsilon_{\mul} + \epsilon_{\exp} + \epsilon_{\text{order}2}, \quad (8) \]

where \( \epsilon_{\text{order}2} \) is a term that gathers all the terms of order 2 or higher in the development of (7). Each of these terms is of the order of \( u^2 \), and for practical values of \( f \) (i.e., \( f > 8 \)) we have \( \epsilon_{\text{order}2} < \epsilon_{\text{order}2} = u/16 \) (a tighter bound is not needed).

Replacing in (8) all the other errors by their upper bounds, using the bound \( M_p \) on \( y\ln(x) \) defined in 2.2, we obtain

\[ \overline{\epsilon}_{\text{total}} = M_p\overline{\epsilon}_{\log} + M_p\overline{\epsilon}_{\mul} + \overline{\epsilon}_{\exp} + \overline{\epsilon}_{\text{order}2}. \quad (9) \]

The bound of 1u on \( \epsilon_{\text{total}} \) thus leads to the constraint

\[ M_p\overline{\epsilon}_{\log} + M_p\overline{\epsilon}_{\mul} + \overline{\epsilon}_{\exp} < 1u - \overline{\epsilon}_{\text{order}2} = (1 - 1/16)u. \quad (10) \]

A rule of thumb, for an efficient implementation, is to try and balance the contributions to the total error of the three suboperators, that is, balance the impact of \( \epsilon_{\mul} \), \( \epsilon_{\exp} \), and \( \epsilon_{\log} \) in the previous equation. Indeed, if one suboperator contributes an error much smaller than another one, we should try to degrade its accuracy in order to save resources.

This means here that we should aim at

\[ M_p\overline{\epsilon}_{\log} \approx M_p\overline{\epsilon}_{\mul} \approx \overline{\epsilon}_{\exp}. \quad (11) \]

Let us now turn to each of these terms.

### 2.4. Exponential Error Analysis

We begin with the exponential that produces the output, and we want to extend its error analysis to that of the complete exponentiation unit.

A hardware implementation of the exponential must use internal guard bits to control its output accuracy. This is typically expressed as

\[ \overline{\epsilon}_{\exp} = (1/2 + t \cdot 2^{-g_{\exp}})u, \quad (12) \]

where the 1/2 is due to the final rounding, \( g_{\exp} \) is the number of guard bits that controls the accuracy of the internal datapath, and \( t \) is a factor that counts the number of last-bit errors along the datapath, and is determined by the error analysis of the chosen implementation. For example, the value \( t = 18 \) in Detrey and de Dinechin [2007] was
refined to \( t = 7 \) in the FloPoCo implementation used here [de Dinechin and Pasca 2010]. An iterative implementation such as Piñeiro et al. [2004] or Detrey et al. [2007] may have a value of \( t \) that depends on the input/output formats.

Equation (10) now becomes

\[
M_p e_{\log} + M_p e_{\text{mul}} + t.2^{-g_{\exp}}u < (1 - 1/2 - 1/16)u. \tag{13}
\]

There is one subtlety here. The error analysis in publications related to the exponential, such as Detrey and de Dinechin [2007] and de Dinechin and Pasca [2010], assumes an exact input to the exponential. In the implementation of \( x^y \), however, the input is \( \text{mul}(y, \ln(x)) \) which is not exact. Its error has been taken into account in the error analysis in previous section, however the exponential begins with a shift left of the mantissa by up to \( r \) bit positions, which could scale up this error by up to \( 2^r \).

To avoid this, we have to modify the architecture of the exponential slightly so that the least significant bit after this shift has weight \( 2^{-g_{\exp}} \), ensuring that the rest of the error analysis of the exponential remains valid. Instead of inputting to the exponential an \( f \)-bit mantissa, we should input \( f + r + g_{\exp} \) bits of the product \( \text{mul}(y, \ln(x)) \).

2.5. Logarithm Error Analysis

To ensure Equation (11), we have to implement a logarithm with a relative error \( e_{\log} \approx t.2^{-g_{\exp}}u/M_p \). The simplest way is to use a faithful logarithm for a mantissa on \( f_{\log} = f + r - 1 + g_{\exp} - [\log_2 t] \) bits. Its error will be bounded by \( 2^{-r-1-g_{\exp}+[\log_2 t]}u \), and as \( M_p < 2^{r-1} \) we have \( M_pe_{\log} < t.2^{-g_{\exp}}u \).

Architecturally, the mantissa of the input \( x \) is simply padded right with zeroes, then fed to this logarithm unit.

2.6. Multiplier Error Analysis

We first remark that this multiplier has asymmetrical input widths: \( y \) is the input, and has an \( f \)-bit mantissa. \( \ln(x) \) is slightly larger, we just saw that its mantissa will have \( f_{\log} = f + r - 1 + g_{\exp} - [\log_2 t] \) bits.

Multiplying these two inputs leads to a \( 2f + r - 1 + g_{\exp} - [\log_2 t] \)-bit mantissa when computed exactly. We then have three options:

— round this product to \( f + r - 1 + g_{\exp} - [\log_2 t] \) bits, which (as for the log) entails a relative error such that \( M_pe_{\text{mul}} < t.2^{-g_{\exp}-1}u \);

— truncate the multiplier result instead of rounding it, which saves a cycle but doubles the error;

— use a truncated multiplier [Banescu et al. 2010; Wires et al. 2001] faithful to \( 2^{-r-g_{\exp}+[\log_2 t]}u \).

This last option will entail the lowest resource consumption, especially in terms of embedded multipliers. In addition, the output mantissa size of this multiplier perfectly matches the extended input to the exponential unit discussed above. This is the choice made for the current implementation.

2.7. Summing Up

With the implementation choices detailed above, we have only one parameter left: \( g_{\exp} \), and Equation (13) now becomes

\[
t.2^{-g_{\exp}}u + t.2^{-g_{\exp}}u + t.2^{-g_{\exp}}u < (1 - 1/2 - 1/16)u, \tag{14}
\]

which defines the constraint on \( g_{\exp} \):

\[
g_{\exp} > -\log_2 \frac{0.4375}{3t}. \tag{15}
\]
Table III. Internal Precision

<table>
<thead>
<tr>
<th>format (r,f)</th>
<th>(f_{\text{log}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8, 23) (single)</td>
<td>33</td>
</tr>
<tr>
<td>(10, 32)</td>
<td>45</td>
</tr>
<tr>
<td>(11, 52) (double)</td>
<td>66</td>
</tr>
</tbody>
</table>

With the chosen implementation of the exponential \[\text{de Dinechin and Pasca 2010}\] \((t = 7)\), we deduce that \(g_{\text{exp}} = 6\) ensures faithful rounding.

Table III shows the resulting value of \(f_{\text{log}}\) for several floating-point formats.

3. ARCHITECTURE AND IMPLEMENTATION

This analysis has been integrated in the FloPoCo tool \[\text{de Dinechin and Pasca 2011}\]. FloPoCo is an open-source core generator framework that is well suited to the construction of complex pipelines such as the exponentiation unit described here. Its salient feature, compared to traditional design flows based on VHDL and vendor cores, is to enable the modular design of flexible pipelines targeting a user-specified frequency on a user-specified target FPGA. FloPoCo is written in C++ and generates human-readable synthesizable VHDL, optimized for a range of Altera and Xilinx targets.

Figure 3 depicts the architecture for the exponentiation unit \(x^y\), showing the intermediate data formats. As can be seen, an exceptions unit completes the logic, while only positive bases are fed to the logarithm. This avoids obtaining a NaN result on the logarithm unit for the case of negative base and integer exponent. A final unit merges exceptions derived directly from the input (following Table I) and exceptions due to overflows or underflows in the computation.

This figure is relatively independent on the implementation of the underlying exponential, logarithm and multiplier cores. Ultimately, most of these computations involve pipelined adders \[\text{de Dinechin et al. 2010b}\] and possibly truncated multipliers \[\text{Banescu et al. 2010; Wires et al. 2001}\], which are still being actively developed in FloPoCo. The results are expected to improve as these subcomponents themselves are improved.

Let us now detail the different subcomponents and discuss possible improvements.

3.1. Exceptions Unit

As seen in Section 1.2, the standard \[\text{IEEE Computer Society 2008}\] defines three variants of the power function, with two of them implemented in this work, pow and powr. The differences among them only impact the exception unit, in charge of handling the exception cases summarized in Table I. In addition to the exceptions related to infinity, zero, NaN, or \(x = 1\), this unit is in charge of detecting if \(y\) is an integer, as in this case a negative \(x\) is allowed for the pow function. This resumes to determining the binary weight of the least significant 1 of the mantissa of \(y\). Let

\[
e = E_y - E_0 - f + z,
\]

where \(z\) is the count of ‘0’ bits to the right of the rightmost ‘1’ bit in \(y\’s\) mantissa. If \(e\) is negative, \(y\) is fractional. Otherwise, \(y\) is an integer, and we have to extract its parity bit which, according to (2), determines the sign of the result.

3.2. Logarithm

The logarithm unit currently used is based on an iterative range reduction \[\text{de Dinechin 2010; Detrey et al. 2007}\]. As Table VI shows, this is the most time- and resource-consuming part of the operator.
In this architecture, there is a parameter, \( k \), such that each iteration involves reading a table with \( k \)-bit input, and performing a multiplication of a large number with a \( k \)-bit number. To match the table size with the embedded RAM size, values of \( k \) close to 10 should be used, but then the embedded multipliers (18x18-bits, or larger) are not fully exploited. A value of \( k = 18 \) is optimal for multiplier usage, but leads to prohibitively large tables. In this work, we use \( k \) between 10 and 12 as a trade-off, but we conclude that this iterative range reduction, designed before the arrival of embedded multipliers and RAM blocks, is not well suited to them.

Therefore, we believe there is much room for improvement here. A natural idea is to use a polynomial approximation. For low-precision, this polynomial could use table-based methods [Detrey and de Dinechin 2005] as they were in [Echeverría and López-Vallejo 2008]. This method has been ported to FloPoCo but currently does not generate pipelined designs. For larger precisions, we could use a generic polynomial approximator designed for embedded memories and multipliers [de Dinechin et al. 2010a].

3.3. Multiplier
As mentioned in Section 2.3, we need a rectangular multiplier, and we choose to use a truncated one. For large precisions, almost half the DSP blocks are saved compared to a standard floating-point multiplier [Banescu et al. 2010; Wires et al. 2001].

3.4. Exponential
The exponential currently used [de Dinechin and Pasca 2010] is based on a table-based [Tang 1989] reduction followed by a polynomial approximation [de Dinechin et al. 2010a]. The single precision version consumes only one 18x18 multiplier and 18Kbit of RAM, which matches very well current FPGAs both from Xilinx and Altera. For larger precisions, the resource consumption remains moderate, as Table VI shows.
Compared to a standard exponential unit, there have been two modifications. At the input, the precision is extended from the standard \( f \) to the \( f + r + g_{\text{exp}} \), as detailed in 2.4. At the output, information from the exception unit is taken into account.

4. EXPERIMENTAL RESULTS

4.1. Testing

FloPoCo comes with a parametric testbench generator framework [de Dinechin and Pasca 2011] that may produce arbitrary numbers of random tests for any floating-point format or pipeline. The random generator may be overloaded in order to test each operator where it needs to be. This is the case here: when using uniformly distributed inputs, the vast majority of the results is either NaN, 0, or +\( \infty \). Testing the accuracy of the unit requires to focus the random number generator on inputs that result in a normal floating-point value.

For each random input, the admissible outputs are computed using arbitrary-precision arithmetic. Here, our requirement of last-bit accuracy (or faithful rounding) allows for two possible values of the output: the two consecutive floating-point numbers that correspond to the rounding up and down of the infinitely accurate result to the target format.

The generated testbench feeds the random inputs to the operator, and compares the output to these two admissible values. An error is reported in case of a discrepancy. A small modification of this setup also allows us to count the percentage of correct rounding.

The reported operators have been tested in this framework for several precisions and thousands of test vectors. We report below synthesis results for operators that passed all the tests they were subjected to, and are therefore faithful to the best of our knowledge; the testbench generator is distributed along the operator to challenge this claim [de Dinechin and Pasca 2011].

We also measured on these random tests that the operator returns the correctly rounded result in more than 95% of the cases. This high quality is due to the fact that we have a worst-case error analysis for all the subcomponents of a very large operator: the probability of matching all the worst cases is low. A larger \( g_{\text{exp}} \) may also be used to obtain a higher percentage of correctly rounded results.

4.2. Synthesis on Virtex-5 and StratixIV

This section reports synthesis results for the FPPowr operators for Altera StratixIV (EP4SGX70HF35C2) on Quartus 11 environment, and Xilinx Virtex-5 (xc5vfx100T-3-fl1738) on ISE 12 environment. To illustrate that the design is fully parameterized, we report in Tables IV and V results obtained on these two targets for the two main floating-point formats, single and double, and for one intermediate precision totalling 45 bits. In addition, Table V shows results of the generated architectures for several frequencies.

For both targets, a single-precision operator consumes a few percents of the resources of a high-end FPGA, and double-precision one consumes about 10%. We do not report post-place-and-route results of this operator alone, which would be meaningless.

It should be noted that FloPoCo adapts the architecture it generates to the target, so the results on Stratix and Virtex come from different VHDL codes.

4.3. Relative Costs of the Subcomponents

Table VI studies the relative performance and costs of the three main subcomponents. The bulk of an exponentiation operator is its logarithm unit, which is also responsible for the critical path. This is partly due to its high accuracy (see Table III and Figure 3),
Table IV. Synthesis Results for Stratix-4 for powr Function

<table>
<thead>
<tr>
<th>precision</th>
<th>performance</th>
<th>resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r,f)</td>
<td>cycles MHz</td>
<td>ALMs M9K</td>
</tr>
<tr>
<td>8,23</td>
<td>35 274</td>
<td>1464 13 1</td>
</tr>
<tr>
<td>10,32</td>
<td>48 228</td>
<td>2689 31 0</td>
</tr>
<tr>
<td>11,52</td>
<td>64 195</td>
<td>4350 57 1</td>
</tr>
</tbody>
</table>

Table V. Synthesis Results for Virtex-5 for powr Function

<table>
<thead>
<tr>
<th>precision</th>
<th>performance</th>
<th>resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r,f)</td>
<td>cycles MHz</td>
<td>Regs+slices BRAMs DSP48</td>
</tr>
<tr>
<td>8,23</td>
<td>15 78</td>
<td>768R + 1631L 7 11</td>
</tr>
<tr>
<td>8,23</td>
<td>27 214</td>
<td>1260R + 1828L 7 11</td>
</tr>
<tr>
<td>10,32</td>
<td>25 80</td>
<td>1341R + 2772L 8 24</td>
</tr>
<tr>
<td>10,32</td>
<td>53 271</td>
<td>2857R + 3291L 8 24</td>
</tr>
<tr>
<td>11,52</td>
<td>37 118</td>
<td>2718R + 4715L 20 42</td>
</tr>
<tr>
<td>11,52</td>
<td>62 192</td>
<td>4511R + 5783L 20 42</td>
</tr>
<tr>
<td>11,52</td>
<td>77 266</td>
<td>5675R + 6153L 20 42</td>
</tr>
</tbody>
</table>

Table VI. Breakdown into Subcomponents on the Virtex-5 Target

<table>
<thead>
<tr>
<th>precision</th>
<th>performance</th>
<th>resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r,f)</td>
<td>cycles MHz</td>
<td>Regs+slices BRAMs DSP48</td>
</tr>
<tr>
<td>FPPowr 8,23</td>
<td>27 214 1260R + 1828L 7 11</td>
<td></td>
</tr>
<tr>
<td>FPLog</td>
<td>15 214 886R + 1302L 6 8</td>
<td></td>
</tr>
<tr>
<td>FPExp</td>
<td>7 230 200R + 441L 1 1</td>
<td></td>
</tr>
<tr>
<td>trunc mult</td>
<td>2 506 26R + 67L 0 2</td>
<td></td>
</tr>
<tr>
<td>FPPowr 11,52</td>
<td>62 192 4511R + 5783L 20 42</td>
<td></td>
</tr>
<tr>
<td>FPLog</td>
<td>34 192 2956R + 3477L 15 24</td>
<td></td>
</tr>
<tr>
<td>FPExp</td>
<td>20 259 1113R + 1565L 5 11</td>
<td></td>
</tr>
<tr>
<td>trunc mult</td>
<td>5 260 354R + 597L 0 7</td>
<td></td>
</tr>
</tbody>
</table>

but there is room for improvement there, as already discussed in Section 3. Should these improvements be implemented, the FPowr unit will inherit them.

4.4. Comparison with Previous Work
Table VII compares the FloPoCo results for single precision for a target frequency of 200 MHz with those from [Echeverría and López-Vallejo 2008] on a Virtex-4 (4vfx100ff1152-12). The results from both units are very similar, only clearly differing in the use of BRAMs (due to the different algorithms used for the logarithm and the exponential functions). We point out that [Echeverría and López-Vallejo 2008] is less accurate, using \((r,f) = (8,23)\) instead of the needed \((8,33)\), losing up to 10 bits of accuracy for some inputs. We are also comparing VHDL code hand-tuned for a specific FPGA and a specific precision [Echeverría and López-Vallejo 2008] to automatically generated code.

4.5. Exceptions Unit
Finally, Table VIII summarizes the cost of the exception control unit for pow(x,y) and powr(x,y), which is the only difference between both operators.

  The bulk of this cost for pow(x,y) is in determining if y is an integer. However it remains very small with respect to the full exponentiation unit.
Table VII. Comparison with Previous Work on Virtex-4

<table>
<thead>
<tr>
<th>Operator(r, f)</th>
<th>latency</th>
<th>freq</th>
<th>slices</th>
<th>BRAM</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Echeverría and López-Vallejo 2008] (8,23)</td>
<td>34</td>
<td>210</td>
<td>1508</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>This work FPPow(8, 23)</td>
<td>36</td>
<td>199</td>
<td>1356</td>
<td>11</td>
<td>13</td>
</tr>
</tbody>
</table>

Table VIII. Cost of the Exception Control Unit

<table>
<thead>
<tr>
<th>Slices</th>
<th>pow</th>
<th>powr</th>
<th>pow</th>
<th>powr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(8, 23)</td>
<td>(11, 52)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slices</td>
<td>23</td>
<td>9</td>
<td>111</td>
<td>13</td>
</tr>
</tbody>
</table>

5. CONCLUSION
The availability of elementary functions for FPGA is essential for developing hardware coprocessors to enhance the performance of computational-heavy applications such as scientific computing or financial or physics simulations. Providing these functions for FPGA is the aim of FloPoCo project.

In this work we have extended FloPoCo’s functions set with the floating-point exponentiation operator, following the functions pow and powr defined in the IEEE 754-2008 standard. This implementation is fully parametric to take advantage of FPGA flexibility. It is portable to mainstream FPGA families in a future-proof way. It is designed to be last-bit accurate for all its inputs, with a careful error analysis to minimize the cost of this accuracy, in particular by using truncated multiplier. We expect the performance to improve in the future as subcomponents, in particular the logarithm unit, are improved.

REFERENCES


Received January 2012; revised May 2012; accepted July 2012