A

Floating-Point Exponentiation Units for Reconfigurable Computing

Florent de Dinechin, École Normale Supérieure de Lyon
Pedro Echeverría, Universidad Politécnica de Madrid
Marisa López-Vallejo, Universidad Politécnica de Madrid
Bogdan Pasca, École Normale Supérieure de Lyon

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 sub-micron 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 [Underwood 2004; Scrofano et al. 2008; 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 thoses found in microprocessors[Shirazi et al. 1995; Louca et al. 1996; Belanović and Leeser 2002], then more recently for operators which are more FPGA-specific, for instance accumulators [Wang et al. 2006; Luo and Martonosi 2000; Bodnar et al. 2006; de Dinechin et al. 2008] or elementary functions [Piñeiro et al. 2004; Doss and Riley 2004; Detrey and de Dinechin 2007; de Dinechin and Pasca 2010].

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 of an operator for the power function x^y in this context. This function appears in many scientific or financial computing kernels [Rebonato 2002; Echeverrí a Aramendi 2011].

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:

$$x^y = e^{y \times \ln x} \ . \qquad (1)$$

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

$$x^y \ = \ (-1)^y \cdot |x|^y \ = \ (-1)^y e^{y \times \ln |x|} \ . \qquad (2)$$

[1] `http://flopoco.gforge.inria.fr/`

Table I. Exception handling for exponentiation functions of the IEEE 754 standard. NaN in the inputs lead to NaN output, except in the two cases for *pow* mentioned in the table.

	pow		*powr*	
Input	**Case**	**Result**	**Case**	**Result**
$x^{\pm 0}$	any x (even NaN)	1	finite $x > 0$	1
$\pm 0^y$	$y < 0$ odd integer	$\pm\infty$	finite $y < 0$	$+\infty$
	$y < 0$ even integer	$+\infty$		
	$y > 0$ odd integer	± 0	finite $y > 0$	$+0$
	$y > 0$ even integer	$+0$		
$\pm 0^{-\infty}$	-	$+\infty$	-	$+\infty$
$-0^{+\infty}$	-	$+0$	-	NaN
$+0^{+\infty}$	-	$+0$	-	$+0$
$-1^{\pm\infty}$	-	1	-	NaN
1^y	any y (even NaN)	1	finite y	1
			$y = \pm\infty$	NaN
$\pm 0^{\pm 0}$	-	1	-	NaN
$+\infty^{\pm 0}$	-	1	-	NaN
x^y	finite $x < 0$ and finite, non integer y	NaN	finite $x < 0$	NaN
$x^{+\infty}$	$\lvert x\rvert < 1$	$+0$	$0 < x < 1$	$+0$
$x^{+\infty}$	$\lvert x\rvert > 1$	$+\infty$	finite $x > 1$	$+\infty$
$x^{-\infty}$	$\lvert x\rvert < 1$	$+\infty$	$0 < x < 1$	$+\infty$
$x^{-\infty}$	$\lvert x\rvert > 1$	$+0$	finite $x > 1$	$+0$
$+\infty^y$	$y > 0$	$+\infty$	$y > 0$	$+\infty$
$+\infty^y$	$y < 0$	$+0$	$y < 0$	$+0$
$-\infty^y$	y even integer	$+\infty$	-	NaN
	y odd integer	$-\infty$		

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 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 $x^{\frac{1}{n}}$.

— `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[2], or in the MPFR library[3].

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 [Ercegovac 1973; Wrathall and Chen 1978; Vázquez and Antelo 2003; Doss and Riley 2004; Detrey and de Dinechin 2007; Detrey et al. 2007; Pottathuparambil and Sass 2009; Altera 2008a; Wielgosz et al. 2009; de Dinechin and Pasca 2010] and logarithm [Ercegovac 1973; Wrathall and Chen 1978; Detrey and de Dinechin 2007; Detrey et al. 2007; Altera 2008b; de Dinechin 2010] 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 precision. First, an in-depth analysis of equation (1) in a floating-point context is presented in Sec-

[2]`http://lipforge.ens-lyon.fr/www/crlibm/`
[3]`http://www.mpfr.org/`

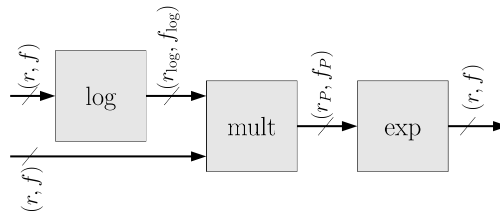

Fig. 1. Simplified overview of a power function unit for an input floating-point format (r, f) where r is the size in bits of the exponent field (defining the range) and f is the size in bits of the fraction field (defining the precision). The intermediate formats $(r_{\log}, f_{\log})$ and (r_P, f_P) must be determined to ensure last-bit accuracy at the minimum cost.

tion 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 exponential, 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 interested 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 rounding, 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, ensuring 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 [Ziv 1991; de Dinechin et al. 2007]. This solution does not allow a fixed-latency hardware implementation. 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 operator 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 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 $E_0 = 2^{r-1} - 1$.

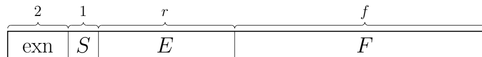

Fig. 2. Format of a floating-point number x.

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 support, 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

Lets α and ω be the smallest and largest positive floating-point numbers respectively. If the exact value of x^y is smaller than α or greater than ω, 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:

$$\begin{array}{rcll} m_p & = & \ln(\alpha), & \text{rounded up} \\ M_p & = & \ln(\omega), & \text{rounded down} \end{array}$$

For practical floating-point formats, this range is very small, as can be seen in Table II for single and double precision.

Table II. Range analysis for the intermediate product

Format	m_p	M_p
FloPoCo single precision (8,23)	-88.03	89.41
FloPoCo double precision (11,52)	-709.09	710.48

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 $\overline{\epsilon}_{\text{total}} = 1u$ where $u = 2^{-f}$ is the relative value of the ulp. Our goal is to ensure that $\epsilon_{\text{total}} < \overline{\epsilon}_{\text{total}}$. This total relative error of the operator is defined by

$$\epsilon_{\text{total}} = \frac{R - x^y}{x^y} = \frac{R}{x^y} - 1. \tag{3}$$

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

We will denote `mul`, `E`, and `Ln` the implementations of the multiplication, exponential and logarithm used here. They will entail the following relative errors:

$$\epsilon_{\text{mul}}(a,b) = \frac{\texttt{mul}(a,b)}{ab} - 1 \; < \; \overline{\epsilon}_{\text{mul}}, \tag{4}$$

$$\epsilon_{\exp}(x) = \frac{\texttt{E}(x)}{e^x} - 1 \; < \; \overline{\epsilon}_{\exp}, \tag{5}$$

and

$$\epsilon_{\log}(x) = \frac{\texttt{Ln}(x)}{\ln x} - 1 \ < \ \overline{\epsilon}_{\log}. \tag{6}$$

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

Rewriting (3), we obtain

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

There, we have in the first operator

$$\texttt{Ln}(x) \ = \ \ln(x)(1 + \epsilon_{\log})$$

while in the multiplier

$$\begin{aligned}
\texttt{mul}(y, \texttt{Ln}(x)) \ &= \ y\texttt{Ln}(x)(1 + \epsilon_{\text{mul}}) \\
&= \ y\ln(x)(1 + \epsilon_{\log})(1 + \epsilon_{\text{mul}}) \\
&= \ y\ln(x)(1 + \epsilon_{\log} + \epsilon_{\text{mul}} + \epsilon_{\log}\epsilon_{\text{mul}}) \\
&= \ y\ln(x) \\
&\quad + y\ln(x)(\epsilon_{\log} + \epsilon_{\text{mul}} + \epsilon_{\log}\epsilon_{\text{mul}})
\end{aligned}$$

and hence in the final exponential

$$\begin{aligned}
R \ &= \texttt{E}(\texttt{mul}(y, \texttt{Ln}(x))) \\
&= e^{\texttt{mul}(y, \texttt{Ln}(x))}(1 + \epsilon_{\exp}) \\
&= e^{y\ln(x) \ + \ y\ln(x)(\epsilon_{\log} + \epsilon_{\text{mul}} + \epsilon_{\log}\epsilon_{\text{mul}})}(1 + \epsilon_{\exp}) \\
&= e^{y\ln(x)} \cdot e^{y\ln(x)(\epsilon_{\log} + \epsilon_{\text{mul}} + \epsilon_{\log}\epsilon_{\text{mul}})}(1 + \epsilon_{\exp})
\end{aligned}$$

and finally the total relative error corresponds to

$$\epsilon_{\text{total}} = e^{y\ln(x)(\epsilon_{\log} + \epsilon_{\text{mul}} + \epsilon_{\log}\epsilon_{\text{mul}})}(1 + \epsilon_{\exp}) - 1 \tag{7}$$

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

$$\epsilon_{\text{total}} = y\ln(x)\epsilon_{\log} + y\ln(x)\epsilon_{\text{mul}} + \epsilon_{\exp} \ + \epsilon_{\text{order2}} \tag{8}$$

where ϵ_{order2} 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 $\epsilon_{\text{order2}} < \overline{\epsilon}_{\text{order2}} = u/16$ (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

$$\overline{\epsilon}_{\text{total}} = M_p\overline{\epsilon}_{\log} + M_p\overline{\epsilon}_{\text{mul}} + \overline{\epsilon}_{\exp} + \overline{\epsilon}_{\text{order2}} \tag{9}$$

The bound of $1u$ on ϵ_{total} thus leads to the constraint

$$M_p\overline{\epsilon}_{\log} + M_p\overline{\epsilon}_{\text{mul}} + \overline{\epsilon}_{\exp} \ < 1u - \overline{\epsilon}_{\text{order2}} = (1 - 1/16)u \tag{10}$$

A rule of thumb, for an efficient implementation, is to try and balance the contributions to the total error of the three sub-operators, *i.e.* balance the impact of ϵ_{mul}, ϵ_{exp} and ϵ_{log} in the previous equation. Indeed, if one sub-operator 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

$$M_p \overline{\epsilon}_{\mathrm{log}} \approx M_p \overline{\epsilon}_{\mathrm{mul}} \approx \overline{\epsilon}_{\mathrm{exp}}. \tag{11}$$

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

$$\overline{\epsilon}_{\mathrm{exp}} = (1/2 + t.2^{-g_{\mathrm{exp}}})u \tag{12}$$

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

$$M_p \overline{\epsilon}_{\mathrm{log}} + M_p \overline{\epsilon}_{\mathrm{mul}} + t.2^{-g_{\mathrm{exp}}} u \ < (1 - 1/2 - 1/16)u \tag{13}$$

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 $\mathtt{mul}(y, \mathtt{Ln}(x))$ 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 that, we have to modify the architecture of the exponential slightly so that the least significant bit after this shift has weight $2^{-f-g_{\mathrm{exp}}}$, 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 $f + r + g_{\mathrm{exp}}$ bits of the product $\mathtt{mul}(y, \mathtt{Ln}(x))$.

2.5. Logarithm error analysis

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

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. $\mathtt{Ln}(x)$ is slightly larger, we just saw that its mantissa will have $f_{\mathrm{log}} = f + r - 1 + g_{\mathrm{exp}} - \lfloor \log_2 t \rfloor$ bits.

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

— Round this product to $f + r - 1 + g_{\mathrm{exp}} - \lfloor \log_2 t \rfloor$ bits, which (as for the log) entails a relative error such that $M_p \bar{\epsilon}_{\mathrm{mul}} < t.2^{-g_{\mathrm{exp}}-1} u$.
— Truncate the multiplier result instead of rounding it, which saves a cycle but doubles the error.
— Use a truncated multiplier [Wires et al. 2001; Banescu et al. 2010] faithful to $2^{-r-g_{\mathrm{exp}}+\lfloor \log_2 t \rfloor} u$.

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

$$t.2^{-g_{\mathrm{exp}}} u + t.2^{-g_{\mathrm{exp}}} u + t.2^{-g_{\mathrm{exp}}} u \; < (1 - 1/2 - 1/16) u \tag{14}$$

which defines the constraint on g_{exp}:

$$g_{\mathrm{exp}} > -\log_2 \tfrac{0.4375}{3t} \tag{15}$$

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

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

Table III. Internal precision

format (r, f)	$f_{\log}$
(8, 23) (single)	33
(10, 32)	45
(11, 52) (double)	66

3. ARCHITECTURE AND IMPLEMENTATION

This analysis has been integrated in the FloPoCo tool [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 is relatively independent on the implementation of the underlying exponential, logarithm and multiplier cores. Ultimately, most of these computations involve pipelined adders [de Dinechin et al. 2010b] and possibly truncated multipliers [Wires et al.

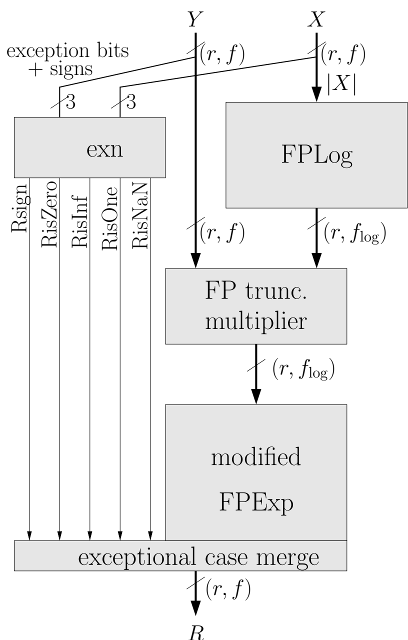

Fig. 3. Detailed architecture.

2001] [Banescu et al. 2010], which are still being actively developed in FloPoCo. The results are expected to improve as these sub-components 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 [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 [Detrey et al. 2007; de Dinechin 2010]. 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 (18×18-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 [Wires et al. 2001; Banescu et al. 2010].

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 18×18 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_{exp} may also be used to obtain a higher percentage of correctly rounded results.

Table IV. Synthesis results for Stratix-4 for `powr` function

precision	performance		resources			
(r, f)	cycles	MHz	ALMs	M9K	M144K	DSP 18-bit
8,23	35	274	1464	13	1	19
10,32	48	228	2689	31	0	41
11,52	64	195	4350	57	1	68

Table V. Synthesis results for Virtex-5 for `powr` function

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

Table VI. Break-down into sub-components on the Virtex-5 target

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

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-ff1738) 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 sub-components

Table VI studies the relative performance and costs of the three main sub-components. 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), but there is room for improvement there, as already discussed in Section 3. Should these improvements be implemented, the FPPowr 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

Table VII. Comparison with previous work on Virtex-4

Operator(r, f)	latency	freq	slices	BRAM	DSP
[Echeverría and López-Vallejo 2008] (8,23)	34	210	1508	3	13
This work FPPow(8, 23)	36	199	1356	11	13

Table VIII. Cost of the exception control unit

	(8, 23)		*(11, 52)*	
	pow	*powr*	*pow*	*powr*
Slices	23	9	111	13

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 $(r_{\log}, f_{\log}) = (8, 23)$ 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.

5. CONCLUSION

The availability of elementary functions for FPGA is essential for developing hardware co-processors 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 sub-components, in particular the logarithm unit, are improved.

REFERENCES

Altera 2008a. *Floating Point Exponent (ALTFP_EXP) Megafunction User Guide*. Altera.

Altera 2008b. *Floating Point Natural Logarithm (ALTFP_LOG) Megafunction User Guide*. Altera.

Banescu, S., de Dinechin, F., Pasca, B., and Tudoran, R. 2010. Multipliers for floating-point double precision and beyond on FPGAs. *ACM SIGARCH Computer Architecture News 38*, 73–79.

Belanović, P. and Leeser, M. 2002. A library of parameterized floating-point modules and their use. In *Field Programmable Logic and Applications*. LNCS Series, vol. 2438. Springer, 657–666.

Bodnar, M. R., Humphrey, J. R., Curt, P. F., Durbano, J. P., and Prather, D. W. 2006. Floating-point accumulation circuit for matrix applications. In *Field-Programmable Custom Computing Machines*. IEEE, 303–304.

de Dinechin, F. 2010. A flexible floating-point logarithm for reconfigurable computers. LIP research report RR2010-22, ENS-Lyon.

de Dinechin, F., Joldes, M., and Pasca, B. 2010a. Automatic generation of polynomial-based hardware architectures for function evaluation. In *Application-specific Systems, Architectures and Processors*. IEEE.

de Dinechin, F., Lauter, C. Q., and Muller, J.-M. 2007. Fast and correctly rounded logarithms in double-precision. *Theoretical Informatics and Applications 41*, 85–102.

de Dinechin, F., Nguyen, H. D., and Pasca, B. 2010b. Pipelined FPGA adders. In *Field-Programmable Logic and Applications*. IEEE, 422–427.

de Dinechin, F. and Pasca, B. 2010. Floating-point exponential functions for DSP-enabled FPGAs. In *Field Programmable Technologies*. IEEE, 110–117.

de Dinechin, F. and Pasca, B. 2011. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*.

de Dinechin, F., Pasca, B., Creţ, O., and Tudoran, R. 2008. An FPGA-specific approach to floating-point accumulation and sum-of-products. In *Field-Programmable Technologies*. IEEE, 33–40.

Detrey, J. and de Dinechin, F. 2005. Table-based polynomials for fast hardware function evaluation. In *Application-specific Systems, Architectures and Processors*. IEEE, 328–333.

Detrey, J. and de Dinechin, F. 2007. Parameterized floating-point logarithm and exponential functions for FPGAs. *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing 31,* 8, 537–545.

Detrey, J., de Dinechin, F., and Pujol, X. 2007. Return of the hardware floating-point elementary function. In *18th Symposium on Computer Arithmetic*. IEEE, 161–168.

Doss, C. C. and Riley, R. L. 2004. FPGA-Based implementation of a robust IEEE-754 exponential unit. In *Field-Programmable Custom Computing Machines*. IEEE, 229–238.

Echeverrí a Aramendi, P. 2011. Hardware acceleration of Monte Carlo-based simulations. Ph.D. thesis, Universidad Politecnica De Madrid, Escuela Tecnica Superior De Ingenieros De Telecomunicacion.

Echeverría, P. and López-Vallejo, M. 2008. An FPGA implementation of the powering function with single precision floating point arithmetic. In *8th Conference on Real Numbers and Computers*. 17–26.

Echeverría, P. and López-Vallejo, M. 2011. Customizing floating-point units for FPGAs: Area-performance-standard trade-offs. *Microprocessors and Microsystems 35,* 6, 535 – 546.

Ercegovac, M. 1973. Radix-16 evaluation of certain elementary functions. *IEEE Transactions on Computers C-22,* 6, 561–566.

Harris, D. 2004. An exponentiation unit for an OpenGL lighting engine. *IEEE Transactions on Computers 53,* 3, 251 – 258.

IEEE Computer Society. 2008. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008. available at `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`.

Kuon, I. and Rose, J. 2007. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26,* 2, 203–215.

Lauter, C. and Lefèvre, V. 2009. An efficient rounding boundary test for pow(x, y) in double precision. *IEEE Transactions on Computers 58,* 2, 197–207.

Louca, L., Cook, T., and Johnson, W. 1996. Implementation of IEEE single precision floating point addition and multiplication on FPGAs. In *FPGAs for Custom Computing Machines*. IEEE, 107–116.

Luo, Z. and Martonosi, M. 2000. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. *IEEE Transactions on Computers 49,* 3, 208–218.

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

Muller, J.-M. 2006. *Elementary Functions, Algorithms and Implementation* 2nd Ed. Birkhäuser.

Piñeiro, J. A., Ercegovac, M. D., and Bruguera, J. D. 2004. Algorithm and architecture for logarithm, exponential and powering computation. *IEEE Transactions on Computers 53,* 9, 1085–1096.

Pottathuparambil, R. and Sass, R. 2009. A parallel/vectorized double-precision exponential core to accelerate computational science applications. In *Field Programmable Gate Arrays*. ACM/SIGDA, 285–285.

Rebonato, R. 2002. *Modern pricing of interest-rate derivatives: The LIBOR market model and beyond*. Princeton Univ Pr.

Scrofano, R., Gokhale, M., Trouw, F., and Prasanna, V. 2008. Accelerating molecular dynamics simulations with reconfigurable computers. *IEEE Transactions on Parallel and Distributed Systems 19,* 6, 764–778.

Shirazi, N., Walters, A., and Athanas, P. 1995. Quantitative analysis of floating point arithmetic on FPGA based custom computing machine. In *FPGAs for Custom Computing Machines*. IEEE, 155–162.

Tang, P. 1989. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software 15,* 2, 144–157.

Underwood, K. 2004. FPGAs vs. CPUs: trends in peak floating-point performance. In *Field Programmable Gate Arrays*. ACM/SIGDA, 171–180.

Vázquez, A. and Antelo, E. 2003. Implementation of the exponential function in a floating-point unit. *Journal of VLSI Signal Processing 33,* 1-2, 125–145.

Wang, X., Braganza, S., and Leeser, M. 2006. Advanced components in the variable precision floating-point library. In *FCCM*. IEEE, 249–258.

Wielgosz, M., Jamro, E., and Wiatr, K. 2009. Accelerating calculations on the RASC platform: A case study of the exponential function. In *ARC '09: Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*. Springer-Verlag, Berlin, Heidelberg, 306–311.

Wires, K. E., Schulte, M. J., and McCarley, D. 2001. FPGA Resource Reduction Through Truncated Multiplication. In *Field-Programmable Logic and Applications*. Springer-Verlag, 574–583.

Woods, N. and VanCourt, T. 2008. FPGA acceleration of quasi-Monte Carlo in finance. *Field Programmable Logic and Applications*, 335–340.

Wrathall, C. and Chen, T. C. 1978. Convergence guarantee and improvements for a hardware exponential and logarithm evaluation scheme. In *4th Symposium on Computer Arithmetic*. 175–182.

Zhuo, L. and Prasanna, V. 2008. High-performance designs for linear algebra operations on reconfigurable hardware. *IEEE Transactions on Computers 57,* 8, 1057–1071.

Ziv, A. 1991. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software 17,* 3, 410–423.