Low-cost multiplier-based FPU for embedded processing on FPGA

Bogdan Pasca
Altera European Technology Centre, UK

***Abstract*—Industrial applications often require processing data with large dynamic ranges at low sample rates. As algorithms become more complex, handling the data range of variables required for fixed-point implementations becomes time consuming, and can also lead to inefficient designs. Floating-point solutions leverage these limitations trading automatic data range handling for a usually higher implementation cost. The adoption of floating-point solutions for this class of applications is conditioned by area and performance requirements. In this paper we present a low-cost floating-point unit which can either be used standalone, or can be attached to a RISC microprocessor. The proposed unit targets modern, multiplier-based FPGAs, computes efficiently costly operations: $\times$, $\div$, $1/x$, $\sqrt{x}$ and $1\sqrt{x}$, requires less than 700LE and 4-9bit multipliers on a CycloneIV and runs close to 150MHz.**

I. Introduction

Industrial applications often work with low sample rates. The low throughput requirement of these applications is ideal for processor based implementations [1]. Existing RISC processor implementations on FPGAs can be sufficiently efficient when dealing with integer datatypes which are directly supported in silicon. Software libraries offering fixed-point data-types and associated functions use the underlying integer hardware [2]. The performance of applications using these depends on the fixed-point types used – the wider the slower, and the operation types – square root will be slower than multiplication.

As algorithms become more complex using wide fixed-point formats in order to cope with the range problems becomes inefficient. Floating-point arithmetic overcomes these limitations to automatically cope with the dynamic range of data. Adding floating-point support to a processor implementation can also be done by using software libraries: GCC, Glibc, μClibc, GoFast Floating-Point Library, [3] for VLIW. Depending on the processor frequency and architecture the performance provided by this solution might be sufficient for some classes of applications. In the FPGA context low-cost devices such as CycloneIV [4] will run soft-core processors like NiosII [5] at frequencies that can reach 150MHz. The long latencies associated to floating-point operations combined with the low frequency yields delays which are too large for our industrial class of applications.

One solution for this low-cost FPGAs is to enhance the processor with a dedicated floating-point unit (FPU). The FPGA reconfigurability enables users to instantiate custom FPUs which match the performance requirements of the target application.

Real application data suggests that the operations to have in the dedicated floating-point coprocessor include: $+$, $-$, $\times$, $\div$, comparators, min/max units, but also $\sqrt{x}$, $1/\sqrt{x}$ and $1/x$ [6]. Some operators which are pervasive in user designs $+, -, \times$ usually need to be fully pipelined to offer high throughput; others such as comparators and min/max units need to have very low latencies, typically 1 or 2 cycles; less frequently encountered ones $\div, 1/x, \sqrt{x}, 1/\sqrt{x}$ need to have reasonably low latencies (10-30 cycles for single precision) and require few resources.

The main contribution of this paper is an efficient and low resource footprint implementation for the FPU section handling less frequently encountered operations such as $\div, 1/x, \sqrt{x}, 1/\sqrt{x}$.

II. Background

The Newton-Raphson technique is a well known iterative method for determining a root y of a function $f(y)$ starting with an initial value y_0 [7], [8]. Under certain conditions, subsequent iterations will converge quadratically towards the root of the function, doubling the accuracy with each iteration. The recurrence formula for the general case is $y_{n+1} = y_n - f(y_n)/f'(y_n)$. The method may determine the inverse of a number x by finding the root y of a function $f(y) = 1/y - x$ having $f'(y) = -1/y^2$. The recurrence formula for the inverse of a number is therefore:

$$y_{n+1} = 2y_n - xy_n^2 \tag{1}$$

The method may similarly determine the inverse square root of a number x by having the function $f(y) = 1/y^2 - x$ with $f'(y) = -2/y^3$. The recurrence formula for the inverse square root is:

$$y_{n+1} = y_n(1.5 - \frac{xy_n^2}{2}) \tag{2}$$

III. Fixed-point kernel extraction

The kernel for most floating-point operations is in fact a fixed-point operation. There are usually several possible algorithms for implementing the same fixed-point function. Our choice of algorithm is guided by: 1/ efficient mapping to FPGA architecture, 2/ resource sharing between kernels and 3/ performance of the implementation.

In this paper we focus on the Newton-Raphson approximation technique for implementing the fixed-point kernels. We have chosen this method because 1/2/ the recurrences are sufficiently simple both for the inverse and inverse square root requiring only multiplications and subtractions; these resources are available in all recent FPGAs and 3/ the method has a quadratic convergence doubling the accuracy with each iteration; this allows for shorter latencies than the digit recurrence methods. An alternative approach still under investigation is to reuse the piecewise polynomial approximation techniques presented [9] and implemented in [10] with fewer polynomials but of higher degree for each function.

A. Inverse Square Root implementation details

The IEEE-754 standard on binary floating-point arithmetic [11] uses a triplet (sign, exponent, fraction) to represent a floating-point number $x = (-1)^s 2^e 1.f$. Let x be our function input and $f(x) = 1/\sqrt{x}$. For $x < 0$ (except negative for which $f(-0) = 1/-0 = -\infty$) the function returns NaN. In the general case $f(x) = 1/\sqrt{2^e 1.Fx}$. We distinguish two cases for the result:

$$r = \begin{cases} 2^{-\frac{e}{2}} \dfrac{1}{\sqrt{1.Fx}} = 2^{eR} \dfrac{1}{\sqrt{1.Fx}} & e \text{ is even} \\ 2^{-\frac{(e-1)}{2}} \dfrac{1}{\sqrt{2 \times 1.Fx}} = 2^{eR} \dfrac{1}{\sqrt{2 \times 1.Fx}} & e \text{ is odd} \end{cases}$$

For inputs with e even and $Fx = 0$ the computed exponent eR will also be the final result exponent. For the all other cases, since $\frac{1}{\sqrt{1.Fx}} \in (1/\sqrt{2}, 1)$ and $\frac{1}{\sqrt{2 \times 1.Fx}} \in (1/2, 1/\sqrt{2}]$ a normalization stage is potentially required ($eR \leftarrow eR - 1$ if fraction $\notin [1,2)$). Regarding the fraction computation, we essentially need to compute the inverse square root for a function $1/\sqrt{z}$ where $z \in [1,4)$.

B. Square Root implementation details

Let $f(x) = \sqrt{x}$. For $x < 0$ (except $f(-0) = -0$) the function returns NaN. In the general case $f(x) = \sqrt{2^e 1.Fx}$ we distinguish two cases, similar to the inverse square root:

$$r = \begin{cases} 2^{e/2} \sqrt{1.Fx} & e \text{ is even} \\ 2^{\frac{e-1}{2}} \sqrt{2 \times 1.Fx} & e \text{ is odd} \end{cases}$$

As it can be observed in both cases, the right-hand square root term is $\in [1,2)$, hence no normalization is necessary. In terms of the fraction, we essentially need to compute the square root of a function $\sqrt{z}$ where $z \in [1,4)$.

C. Reciprocal implementation details

Let $f(x) = 1/x$. The function is defined for the entire range of x: $f(-0) = -\infty$, $f(+0) = +\infty$ and in the general case:

$$f(x) = \frac{1}{2^e 1.Fx} = 2^{-e} \frac{1}{1.Fx} = 2^{eR} \frac{1}{1.Fx}$$

Let eR be the temporary exponent value pre normalization. When the result is in $(1/2,1)$ the new exponent is $eR \leftarrow eR - 1$. Regarding the fraction computation, we need to compute the inverse $1/z$ for $z \in [1,2)$.

D. Division implementation details

Let $f(x,y) = x/y$. Division has a few undefined cases (output NaN): $0/0, \infty/\infty$. In the general case,

$$f(x) = \frac{(-1)^{sx} 2^{ex} 1.Fx}{(-1)^{sy} 2^{ey} 1.Fy} = (-1)^{sR} 2^{ex-ey} \frac{1.Fx}{1.Fy}$$

The term $1.Fx/1.Fy$ belongs to the interval $(1/2,2)$. Let $eR = ex - ey$ be the temporary exponent. If $1.Fx/1.Fy \in (1/2,1)$ then $eR \leftarrow eR - 1$. Division will be implemented as the inverse of y multiplied by x and hence the procedure to compute the fraction will consist of fist computing the fixed-point inverse of y and then multiplying it by the fraction of x. A similar implementation for the divider can be found in [12] where the piecewise polynomial approximation and Newton-Raphson techniques are combined in the context of a high throughput, low latency operator.

E. Multiplication implementation details

Let $f(x,y) = x \times y$. Similar to division, for specific inputs the function returns NaN: $0 \times \infty$.

In the general case:

$$\begin{aligned} f(x) &= (-1)^{sx} 2^{ex} 1.Fx \times (-1)^{sy} 2^{ey} 1.Fy \\ &= (-1)^{sR} 2^{ex+ey} 1.Fx \times 1.Fy \end{aligned}$$

The term $1.Fx \times 1.Fy$ will be $\in [1,4)$. When this result is $\in [2,4)$ a normalization is required where $eR \leftarrow eR + 1$.

IV. Minimal Datapath Width

We denote by (wE, wF) the floating-point format (wE exponent width, wF fraction width) and we target faithful rounding (the returned result can be any of the 2 floating-point numbers closest to the mathematical result of the operation; for more details see [13]). Faithful rounding allows for an error budget of 1 unit in the last place – *ulp* (which is the distance between 2 floating-point numbers). When rounding the current fraction value to the nearest output format, a maximum error of $0.5ulp$ is possible. The remaining budget for the approximation error is $0.5ulp = 1ulp - 0.5ulp$.

This calculation assumes that the fraction has been normalized and that the approximation error of $0.5ulp$ holds after normalization. In other words, we need to produce a fraction result having $1 + wF + 1$ meaningful bits (first '1' is the implicit one) and the approximation error is smaller than $0.5ulp$ (the magnitude of the LSB) after normalization. In section III we have shown that the dynamic data range for the fraction calculation belongs to the interval $[1/2,4)$. When the values are in $[1/2,1)$ the MSB will be in binary position -1 and $1 + wF + 1$ bits (25 for single precision) are required to the right. When the values are in $(4,2]$ the MSB will be in binary position 1. Between the 2 extreme cases, the fraction computation datapath will require $2 + 1 + wF + 1$ bits (27 in single-precision). This proves that both inputs and outputs can be represented with sufficient precision using our proposed format (2 integer bits, $1 + wF + 1$ fractional). Next we show that the internal iteration calculation can also be represented with sufficient accuracy using this precision.

A. Inverse square root

The Newton-Raphson technique is used for the fixed-point mantissa calculation of the inverse square root. The technique inputs an initial approximation, and performs a number of iterations to converge to the final solution using Eq.2. The technique has quadratic convergence, doubling the accuracy (number of significant digits in the result) with each iteration. For single precision an initial approximation accurate to 7 bits and 2 iterations would produce a solution accurate to approx. 28 bits. For a detailed error analysis of the Newton-Raphson iteration see [14].

The initial approximation is read from the table then it gets squared (y_0^2). Since the initial approximation is in $(1/2,1]$ squaring would produce a the result in (1/4,1]. Next, the result gets multiplied by the input x which is in $[1,4)$ (xy_0^2). Using basic interval arithmetic we can clearly see that the resulting interval will be in $(1/4,4)$, which does not overflow on the 2.25

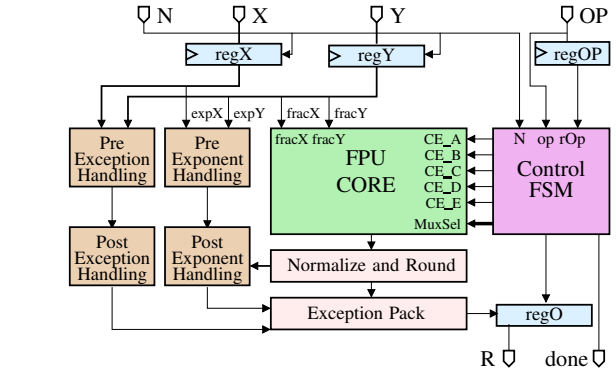

Fig. 1. High level diagram of the multplier based FPU

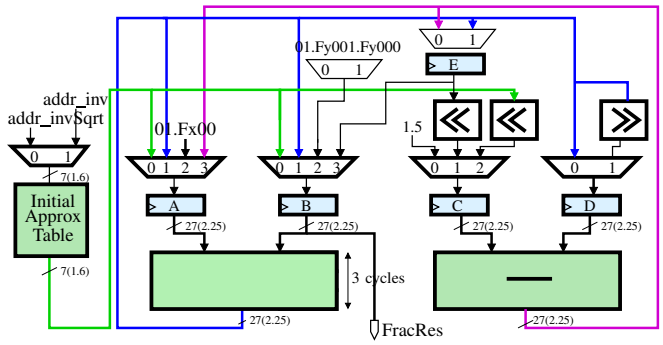

Fig. 2. The multiplier-based core of the floating-point unit

target format. However, observing the calculation $x \times 1/\sqrt{\tilde{x}^2}$ we see that the returned value is close 1. Dividing this value by 2 returns a value in the 0.5 range and subtracting this value out of 1.5 will return again a value close to 1. Since the calculation will always return a positive value the following multiplication can be performed on unsigned data. The result of multiplying the initial value of the approximation by this value close to 1 produces a new approximation y_1 which is twice as accurate. The whole iteration is performed once more in order to obtain y_2 which will be accurate to at least 25 significant bits.

For single precision the entire calculation can be performed using a a 27-bit wide datapath. Similarly it can be shown that the same datapath can be used for the inverse, square root, division and multiplication.

V. Implementation

The high-level diagram of the floating-point unit is presented in Figure 1. The multiplier-based core, capable of performing 5 operations is detailed in Figure 2. The main units of this core are a 27-bit fixed-point multiplier, a fixed-point subtracter and a unit providing the initial approximation values for the inverse and inverse square root. These units are interconnected using a network of multiplexers. The inputs to both the multiplier and the subtracter are registered, with the enable lines of these register being controlled by the ControlFSM. The size of the multiplexers is deliberately kept small which allows implementing them using one level of look-up tables. The static shift blocks feeding the subtracter unit do not take any logic and will be implemented as a simple rewiring.

The fully distributed way this FPU unit is built allows performing a tighter operation scheduling which significantly

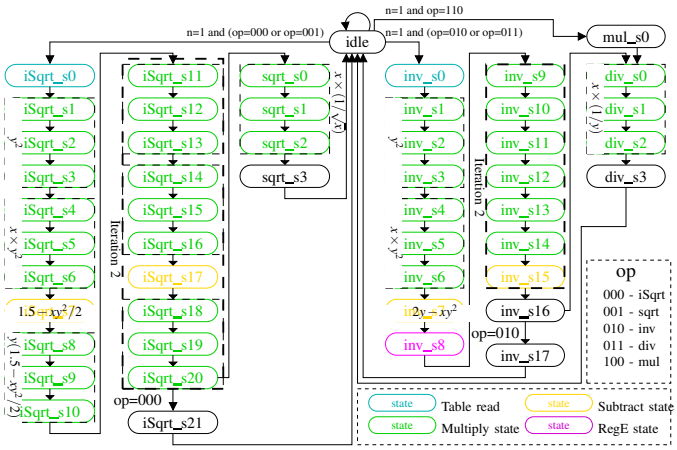

Fig. 3. Detailed scheme of the control FSM

reduces the latency of operations.

In the optimized version, since the inputs to the approximation table ROM are combinatorial and the table is implemented using distributed memory, the outputs are available at the same cycle. Therefore, reading the approximation and writing it back for squaring to the input registers of the multiplier is performed in one clock cycle. The multiplication takes two cycles and the resulting product will then need to be multiply by x. Fetching the value of x in the input B register of the multiplier is therefore done in parallel with the multiplication.

Similarly, following the second multiplication xy_0^2 the product will need to be shifted right and then subtracted from the value 1.5. This shifting and buffering into the input register of the subtracter is done in one cycle together with preparing the C input of the subtracter. Parallel with the subtraction the approximation y_0 is read from the approximation table and both the difference $1.5 - xy_0^2/2$ and y_0 will be buffered in the input registers of the multiplier. Multiplication is performed and result written to the A and E registers, ready for the next iteration.

In comparison with a regular scheduling of these operations on a unit non optimized for these specific computations the latency reduction for the first iteration is from 15 cycles to 9 cycles, or roughly 40%.

The detailed description of the control FSM is depicted in Figure 3. It shows the steps required for each operation and groups the states in order to depict the operations being performed during these states. From left to right the states corresponding to the inverse square root, square root, inverse, division and finally multiplication are depicted.

Since a significant part of the stages used by both square root and division are in fact common to the corresponding inverse square root and inverse stages, these stages are factored out. Similarly, the stages in division corresponding the the multiplication will also be reused.

VI. Results and Discussion

Table I shows the latencies of the supported functions for both the fast (3 stages/mult) and slow (2 stages/mult) versions a Cyclone-IV device. The fast version runs close to 150MHz on the slowest Cyclone-IV device and the slow runs at 105MHz.

Operation	Latency Fast	Latency Slow	Resources
$1/\sqrt{x}$	23 cycles	17 cycles	693/664 LEs
$\sqrt{x}$	26 cycles	19 cycles	
$1/x$	19 cycles	15 cycles	4 9-bit mults
x/y	22 cycles	17 cycles	
$x \times y$	6 cycles	5 cycles	

TABLE I. FUNCTIONS/LATENCIES FOR 155MHZ/105MHZ

Operation	Fast	Fast (half)	Slow	Slow (half)
$1/\sqrt{x}$	23 cycles	13 cycles	17 cycles	10 cycles
$\sqrt{x}$	26 cycles	16 cycles	20 cycles	12 cycles
$1/x$	19 cycles	10 cycles	15 cycles	8 cycles
x/y	22 cycles	17 cycles	18 cycles	10 cycles
$x \times y$	6 cycles	-	5 cycles	-

TABLE II. LATENCY FOR EARLY TERMINATIONS (1 ITERATION)

The total number of logic elements used by the implementation is 693 for the fast version (664 for the slow version), together with a 4 9-bit multiplier elements (2 DSPs).

The presented unit is based on convergence-based algorithms and therefore we can take advantage of this and return a potentially less accurate result in fewer cycles. For the targeted single precision datapath we can return a result either after 1/ fetching the initial approximation from memory; this would correspond to approximately 2 decimal digits of accuracy; 2/ after one iteration; the accuracy of the result will roughly be 4 decimal digits. The modifications to the control FSM are minor in order to support these options. The latency of the operations will be significantly shorter for these cases. Table II presents these values for both the fast and the slow architectures.

The current architecture can easily be ported to more recent devices such as CycloneV/ArriaV with DSP blocks supporting 27-bit multipliers in 2 cycles. On one hand, the latencies for the operations will now be equal to what we presented so far as the *slow* Cyclone-IV version. On the other hand, the frequency is expected to be significantly higher in these devices. Preliminary results on a CycloneV C8 (slowest speedgrade) show that the version having 2 cycles for the multiplier now requires 243ALMs, 1DSP and runs at 148MHz (105MHz for CycloneIV) and on ArriaV (C4 speedgrade) the frequency is 371MHz.

When extending the architecture to support higher precisions such as double (binary64) a tradeoff is to be made between the number DSPs and operation latency. Additionally, a double precision unit will probably need to support single precision as well, and possible other precisions in between these two, possibly supporting early termination. The simple extension is to actually perform folding inside the multiplier block and keep everything else unchanged. The initial approximation tables are sufficiently accurate to cover the double-precision case since 7x2x2x2=56.

The presented architecture is an iterative one, accepting one new set of inputs only when the current operation is done. However, the multiplication is currently implemented in a fully pipelined way and hence the unit could potentially be modified to accept one set of inputs per cycle providing that the operation is multiplication. This would have a positive effect on performance and would require very little logic to implement.

Additionally, we can modify the architecture so that it accepts an new set of inputs for multiplication even if it is already processing some other operation. This can be accomplished by splitting the output port and dedicating one pair of data/done for multiplication.

VII. CONCLUSION

We have presented in this paper a floating-point unit targeting industrial applications and having 5 functional units: $1/\sqrt{x}$, $\sqrt{x}$, $1/x$, x/y, $x \times y$. The advantage of this fused iterative unit are 1/ its low resource usage, requiring less than 700 logic elements and 4 9-bit multipliers on a Cyclone-IV FPGA and 2/ its frequency and 3/ its flexibility with the early termination cycles. We also raise in this paper several questions about implementation alternatives and tradeoffs for higher precisions (latency vs. DSPs), combining higher and lower precisions (support single and double in one unit), concurrency (fully pipelined multipliers, multiplier in parallel with other function, 2 single multiplications in parallel in the double-precision version of the unit). These questions correspond to new research directions opened by this work.

REFERENCES

[1] Altera Corporation, 2011. [Online]. Available: http://www.altera.com/literature/wp/wp-01154-flexible-industrial.pdf

[2] M. Moise, "Fixed point arithmetic library for SpiNNaker," Master's thesis, University of Manchester, School of Computer Science, Sept. 2011. [Online]. Available: http://studentnet.cs.manchester.ac.uk/resources/library/thesis_abstracts/MSc12/FullText/Moise-Mircea-fulltext.pdf

[3] G. Revy, "Implementation of binary floating-point arithmetic on embedded integer processors - polynomial evaluation-based algorithms and certified code generation," Ph.D. dissertation, Université de Lyon - École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France, December 2009.

[4] *CycloneIV Device Handbook*, 2013, http://www.altera.com/literature/hb/cyclone-iv/cyclone4-handbook.pdf.

[5] *Nios II Processor Reference Handbook*, feb 2014, http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.

[6] *User Reference Manual – Digital Signal Controller; General Functions Library*, 2011, http://cache.freescale.com/files/microcontrollers/doc/user_guide/56800E_GFLIB.pdf?fpsp=1.

[7] T. J. Ypma, "Historical development of the newton-raphson method," *SIAM Rev.*, vol. 37, no. 4, pp. 531–551, Dec. 1995. [Online]. Available: http://dx.doi.org/10.1137/1037125

[8] M. Cornea-Hasegan, R. Golliver, and P. Markstein, "Correctness proofs outline for newton-raphson based floating-point divide and square root algorithms," in *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on*, 1999, pp. 96–105.

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

[10] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design and Test*, 2011.

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

[12] B. Pasca, "Correctly rounded floating-point division for DSP-enabled FPGAs," in *22th International Conference on Field Programmable Logic and Applications (FPL'12)*. Oslo, Norway: IEEE, Aug. 2012.

[13] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010, ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[14] M. Joldes, J.-M. Muller, and V. Popescu, "On the computation of the reciprocal of floating point expansions using an adapted Newton-Raphson iteration," Tech. Rep., 2014. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00957379