

# 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 an automatic data range handling for a usually higher implementation cost. The acceptance of floating-point solution for this class of applications is conditioned by area and while meeting 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.

## 1. 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,  $\mu$ Clibc, 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 which 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.

In this paper we focus on providing an efficient implementation for the FPU section handling less frequently encountered operations such as  $\div$ ,  $1/x$ ,  $\sqrt{x}$ ,  $1/\sqrt{x}$ . The proposed architecture uses the Newton-Raphson algorithm for computing the inverse and the inverse square root and uses these values for computing the division and the square root. The fixed-point datapath of the floating-point architecture first extracted, minimized and mapped on a small fixed-point multiplier, a subtracter and a well chosen network of multiplexers. The size of the multiplexers is selected so that is maps well to FPGAs. The tight operation scheduling extracts instruction level parallelism in order to reduce latency. The results demonstrate that a suited unit for these applications can be built using less than 700 logic elements (LE) and 4-9bit multipliers on a CycloneIV and can close timing at close to 150MHz.

The rest of the paper is organized as follows: after a brief overview of floating-point arithmetic and the general Newton-Raphson technique in Section 2 we start presenting our algorithm in Section 3 by first extracting the algorithmic fixed-point datapath for our floating-point functions; next we compute the minimum datapath width for a fixed-point datapath capable of performing all these functions. The Newton-Raphson algorithm is selected for computing the mantissa datapath, and based on implementation trade-

offs the number of iterations is determined; the operations are then tightly scheduled on the minimal hardware architecture. The implementation is presented in section 4 and Section 5 presents the synthesis results and latencies of the supported operations and also presents a discussion around the possible extensions of this work. In section 6 we present the conclusions and open research topics based on this work.

## 2. BACKGROUND

The IEEE-754 standard on binary floating-point arithmetic (revised in 2008 [7]) uses a triplet (sign, exponent, fraction) to represent a floating-point number:

$$x = (-1)^s 2^e 1.f$$

The number of bits used to store the exponent and fraction ( $w_E$  and  $w_F$ ) define the formats of the IEEE-754 standard. Some of the most commonly used formats are  $w_E = 8$  and  $w_F = 23$  for single-precision and  $w_E = 11$  and  $w_F = 52$  for double precision. Custom formats (pairs of  $w_E, w_F$  not specified in the standard) are used to bridge the precision gap between single and double precision. These formats can be used to exploit the flexibility of FPGAs and outperform microprocessors [8].

Newton-Raphson technique is a well known iterative method for determining a root  $y$  of a function  $f(y)$  starting with an initial value  $y_0$  [9, 10]. 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 given by:

$$y_{n+1} = y_n - f(y_n)/f'(y_n)$$

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:

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

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:

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

## 3. ALGORITHM

### 3.1. Fixed-point kernel extraction

The kernel computation of most floating-point operations is in fact a fixed-point operation. Therefore, the first step in implementing the proposed functions in the FPU is to identify

the fixed-point computing kernels. Once identified and once decided on what algorithm is used for each function, we will try to fuse the kernel implementations. The main objective for constructing the fused fixed-point kernel is sharing as many resources as possible.

In this paper we focus on the Newton-Raphson approximation technique for implementing the fixed-point kernels. We have chosen this method because 1/ 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 2/ 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 [11] and implemented in [12] with fewer polynomials but of higher degree for each function.

#### 3.1.1. Inverse Square Root implementation details

Let:  $x = (-1)^s 2^e 1.Fx$  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:

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

For inputs with an even exponent and  $Fx = 0$  the computed exponent of the result  $eR$  will also be the final exponent of the result. For the 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 necessary. During this stage the fraction of the result is brought into the  $[1, 2)$  by multiplying it by 2 while decrementing the exponent  $eR \leftarrow eR - 1$ .

Regarding the fraction computation, we essentially need to compute the inverse square root for a function  $1/\sqrt{z}$  where  $z \in [1, 4)$ .

#### 3.1.2. Square Root implementation details

Let:  $x = (-1)^s 2^e 1.Fx$  and  $f(x) = \sqrt{x}$ . 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  $z \in [1,4)$ . An efficient FPGA implementation of the square root operator which uses the piecewise polynomial approximation technique can be found in [13]. In this paper we focus on another multiplicative technique, Newton-Raphson which has fewer memory block requirements.

### 3.1.3. Reciprocal implementation details

Let:  $x = (-1)^{sx}2^{ex}1.Fx$  and  $f(x) = \frac{1}{x}$ . The function is defined for the entire range of  $x$ :  $f(-0) = -\infty$ ,  $f(+0) = +\infty$  and in the general case:

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

$$\frac{1}{1.Fx} \in (1/2, 1) \rightarrow \text{normalize}$$

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  $z \in [1, 2)$ .

### 3.1.4. Division implementation details

Let:  $x = (-1)^{sx}2^{ex}1.Fx$ ,  $y = (-1)^{sy}2^{ey}1.Fy$  and  $f(x,y) = \frac{x}{y}$ .

Division has a few cases for which the result is 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. We distinguish two cases in determining the final exponent value:

$$eR = \begin{cases} eR & : 1.Fx/1.Fy \in [1, 2) \\ eR - 1 & : 1.Fx/1.Fy \in (1/2, 1) \end{cases}$$

Division will be implemented as the inverse of  $y$  multiplied by  $x$  and hence the procedure to compute the fraction will consist of first 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 [14] where the piecewise polynomial approximation and Newton-Raphson techniques are combined in the context of a high throughput, low latency operator.

### 3.1.5. Multiplication implementation details

Let:  $x = (-1)^{sx}2^{ex}1.Fx$ ,  $y = (-1)^{sy}2^{ey}1.Fy$  and  $f(x,y) = x \times y$ . Similar to division, for specific inputs the function returns NaN:  $0 \times \infty$ .

In the general case:

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

$$= (-1)^{sR}2^{ex+ey}1.Fx \times 1.Fy$$

The term  $1.Fx \times 1.Fy$  will be  $\in [1, 4)$ . When this result is  $\in [2, 4)$  a normalization is required. Let  $eR = eX + eY$  be the temporary exponent value. Normalization consists in:

$$eR = \begin{cases} eR & : 1.Fx \times 1.Fy \in [1, 2) \\ eR + 1 & : 1.Fx \times 1.Fy \in [2, 4) \end{cases}$$

## 3.2. Minimal Datapath Width

We denote by  $(wE, wF)$  the floating-point format 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 [15]). 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 running fraction value to the output format the smallest maximum error is obtained for rounding to nearest, and has a value of  $0.5ulp$ . This leaves  $0.5ulp$  for the approximation error budget.

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 3.1 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  $(1, 2]$  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 accuracy using our proposed format (2 integer bits,  $1 + wF + 1$  fractional).

We next need to show that the internal datapaths computation do not exceed this format. All the computed fractions belong to  $[1/2, 4)$ .

### 3.2.1. 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, accurate to a number of bits, and uses a number of iterations to converge to the final result using Eq.2. For single precision we need to make sure that there are at least  $1 + wF + 1 = 25$  significant bits after the iterations (meaning that the error needs to be less than  $1ulp$ ). It is well known that Newton-Raphson technique has

quadratic convergence, meaning that it doubles the accuracy (number of significant digits in the result) with each iteration. Using this information we chose an initial approximation of 7 bits and we target 2 iterations so that our results are sufficiently accurate. For a detailed error analysis of the error of Newton-Raphson iteration see [16].

The inverse square root has two distinct cases. For an even input exponent the initial approximation is obtained from a table indexed by the leading bits of  $1.Fx$ . The values for this initial approximation belong to  $(1/\sqrt{2}, 1]$ . For the odd exponent a secondary table is indexed by the leading bits of  $1.Fx$ . However, the function tabulated now is  $\sqrt{2} \times 1.F$  having an image in  $(1/2, 1/\sqrt{2}]$ . The two logical tables (one for each case) are fused in one physical table which is indexed by the by concatenating the exponent parity bit to the MSB of the 7-input address. The output fixed-point semantic of the table will be 1.6 (1 bit of integer, 6 of fraction).

One can observe that except for  $e$  even and first 6 leading bits of  $Fx$  zero, the most significant bit of the table is always zero. We can therefore store only the 6 leading bits in the table and apply a mask at the table output to obtain the final result.

The Newton-Raphson iteration used is presented in Equation 2. First, 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 target format. However, looking closer at the calculation we observe that the returned value is close 1 (since the computation is  $x \times 1/\sqrt{\tilde{x}^2}$  where  $\tilde{x}$  is an approximation of  $x$  by selecting the most significant digits).

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 27-bit wide datapath. Looking at the operations used to calculate the inverse square root we identify that fixed-point multipliers, a fixed-point subtracter and tables containing initial approximations are the main compute units.

### 3.2.2. Inverse

The inverse is implemented similarly, using the Newton-Raphson technique presented to Equation 1. The input to

the calculation is in the  $[1, 2)$  interval having an output in  $(1/2, 1]$ . We target 7 bits of precision for the initial approximation. For reducing the size of the table we use a similar technique separating the case when  $\tilde{x} = 1$ , for which the approximation result is also 1.

In the first iteration  $y_0 \in (1/2, 1]$  is squared  $y_0^2 \in (1/4, 1]$ . This is then multiplied by  $x \in [1, 2)$  resulting in a worst case interval of  $(1/4, 2)$ . However, accounting for the dependency between  $x$  and  $y_0$  we can reduce the second interval to  $(1/2, 1]$ . In fact, the product  $xy_0^2$  will be very close to the value of the inverse of  $x$ . This information is used for bounding the resulted interval of the last subtraction which now will produce a positive result being twice as accurate. A subsequent iteration is required for obtaining  $y_2$  which will be accurate to at least 25 significant bits.

### 3.2.3. Square root and division

The result after 2 iterations will produce  $y_2 \in (1/2, 1]$  for both the square root and the division. This result needs to be multiplied by  $x \in [1, 2)$ . For division the inverse was calculated on  $y$ . The resulting interval will be  $(1/2, 2)$ . Before returning the final result a normalization is necessary if the result is in  $(1/2, 1)$ .

### 3.2.4. Multiplication

The two inputs in  $[1, 2)$  will produce a result in  $[1, 4)$  and will require normalizing if in  $[2, 4)$ .

## 4. IMPLEMENTATION

The high-level diagram of the floating-point unit is presented in Figure 1. The two data inputs ( $X$  and  $Y$ ) together with the control signals ( $N$  – going high for one cycle together with the new pair of data inputs and  $Op$  – denoting the code of the operation to be executed) are the inputs of the component. The main computational blocks are: 1/ FPU core which computes iteratively the fixed-point kernel values for each operation; Pre/Post Exponent Handling blocks computing exponents for the functions; Pre/Post Exception Handling blocks computing the exception cases for the functions; the Control FSM controlling the execution of both the FPU Core but also of the buffering registers; Normalize/Round block which is required to normalize and round the result produced by the FPU Core which belong to the interval  $(1/2, 4)$ .

The core raises **done** when the operation completes and the result can be read from the output register **regO**.

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

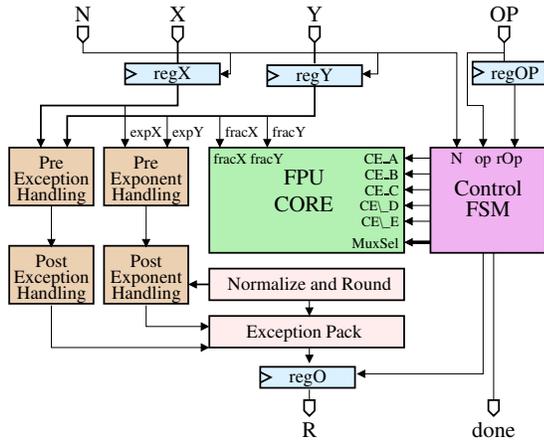


Fig. 1. High level diagram of the multiplier based FPU

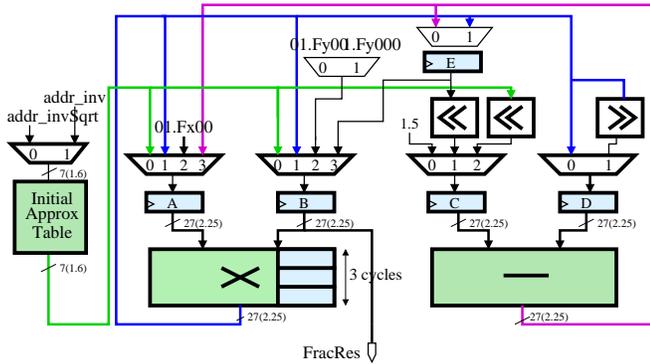


Fig. 2. The multiplier-based core of the floating-point unit

both the multiplier and the subtractor 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 subtractor 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 reduces the latency of operations. Figure 3 presents a comparison between a regular, sequential operation scheduling for the inverse square root computation and the parallel, optimized scheduling of our unit.

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

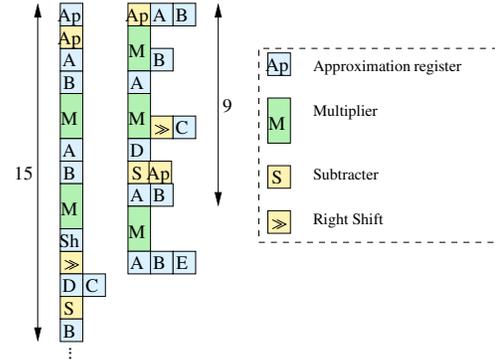


Fig. 3. Tight computation scheduling

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.

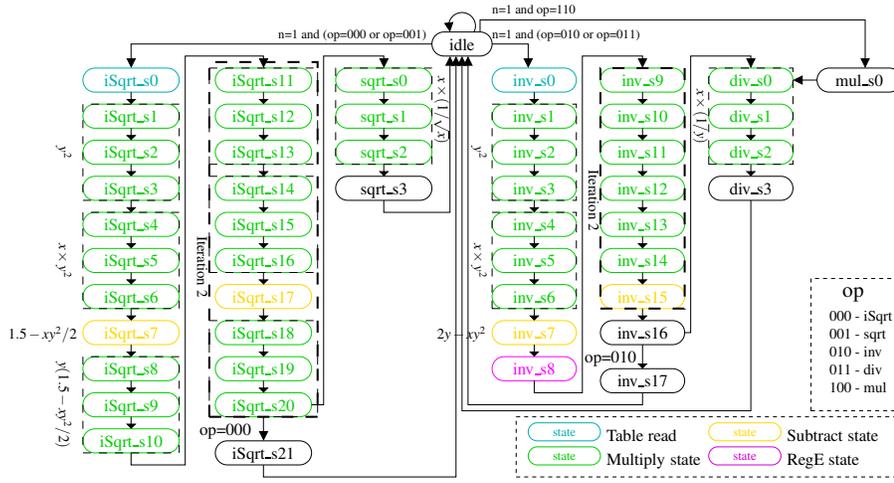
Similarly, following the second multiplication  $xy_0^2$  the product will need to be shifted right and then subtracted from the value 1.5. This shifting and buffering into the input register of the subtractor is done in one cycle together with preparing the  $C$  input of the subtractor. Parallel with 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 4. 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. For instance, when transitioning from stage **iSqrt\_20** depending on the operation code one would carry on into stage **sqrt\_s0** if the operation code for square root was valid, or would carry on into stage **iSqrt\_21** and then back to **idle**. A similar decision is being taken transitioning from state **inv\_s16** for continuing with division or preparing the output result of the inverse.

Since the stages performing the mantissa multiplications essentially consist in one multiplication, the division stages corresponding to the multiplication between  $x$  and the inverse will be shared between the division and the multiplication.



**Fig. 4.** Detailed scheme of the control FSM

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	

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 1.** Supported functions latencies for fast mode 155MHz and for slow mode 105MHz on Cyclone-IV devices, slowest speedgrade (C7)

**Table 2.** Operation latency for early terminations (after 1 Newton-Raphson iteration) in both the fast and slow case

## 5. RESULTS AND DISCUSSION

Table 1 presents the latencies for the supported functions for both the fast and slow versions of the unit on a Cyclone-IV device. The main difference between the fast and the slow versions is the number of pipeline stages in the fixed-point multiplier with fast having 3 stages and slow having 2 stages. In terms of performance, the fast version corresponds to a working frequency close to 150MHz on the lowest speedgrade Cyclone-IV device (slowest) whereas the slow version reaches roughly 105MHz on the same device.

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 wide datapath we can return a result either after:

- fetching the initial approximation from memory; this would correspond to approximately 2 decimal digits of accuracy.

- 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 2 presents these values for both the fast and the slow architectures.

The current architecture can easily be ported to more recent targets such as the CycloneV/ArriaV devices. Due to the extended capabilities of the DSP block in these devices the  $27 \times 27$ -bit multiplication can now be performed in 2 cycles at high frequencies in one single DSP. 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) there is a tradeoff to be made between the number of DSPs used for the multiplica-

tion and the latency of the operations. 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  $7 \times 2 \times 2 \times 2 = 56$ .

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 clearly have a positive effect on performance and would require very little logic to implement.

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

## 6. 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.

## 7. 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 SpiN-Naker," 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](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](http://cache.freescale.com/files/microcontrollers/doc/user_guide/56800E_GFLIB.pdf?fpsp=1).
- [7] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1-58, 29 2008.
- [8] F. de Dinechin, J. Detrey, I. Trestian, O. Creț, and R. Tudoran, "When FPGAs are better at floating-point than microprocessors," École Normale Supérieure de Lyon, Tech. Rep. ensl-00174627, 2007, <http://prunel.ccsd.cnrs.fr/ensl-00174627>.
- [9] 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>
- [10] 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.
- [11] 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.
- [12] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design and Test*, 2011.
- [13] F. de Dinechin, M. Joldes, B. Pasca, and G. Revy, "Multiplicative square root algorithms for FPGAs," in *International Conference on Field Programmable Logic and Applications*. IEEE, aug 2010.
- [14] 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.
- [15] 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.
- [16] 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>