EFFICIENT IMPLEMENTATION OF A SINGLE-PRECISION FLOATING-POINT ARITHMETIC UNIT ON FPGA

Wilson José⋆, Ana Rita Silva⋆, Horácio Neto⋆, Mário Véstias†

⋆INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal,
‡INESC-ID, Instituto Superior de Engenharia de Lisboa, IPL, Portugal,
wilsonmaltez@inesc-id.pt, anaritasilva@inesc-id.pt, hcn@inesc-id.pt, mvestias@deetc.isel.pt

ABSTRACT
This paper presents a single precision floating point arithmetic unit with support for multiplication, addition, fused multiply-add, reciprocal, square-root and inverse square-root with high-performance and low resource usage. The design uses a piecewise 2nd order polynomial approximation to implement reciprocal, square-root and inverse square-root. The unit can be configured with any number of operations and is capable to calculate any function with a throughput of one operation per cycle. The floating-point multiplier of the unit is also used to implement the polynomial approximation and the fused multiply-add operation. We have compared our implementation with other state-of-the-art proposals, including the Xilinx CoreGen operators, and conclude that the approach has a high relative performance/area efficiency.

I. INTRODUCTION
In this work we have designed and implemented a reconfigurable single-precision floating-point unit (FPU) for FPGA to be interfaced with some known embedded processors. The MEIKO from Sun is an FPU used, for example, with the LEON processor [1]. Another FPU unit designed for the LEON processor is the GRFPU (Gaisler Research FPU) [2] distributed in VHDL for both ASIC and FPGA. Addition, subtraction and multiplication are fully pipelined. Divide and square-root are based on iterative algorithms that calculate a series expansion from approximation tables. Milk [3] was proposed as a scalable FPU for embedded systems. The unit is configurable and each functional unit operates independently from the others. Besides the basic operations it also supports division and square-root. However, it does not implement the fused multiply-add operation. All functions are fully pipelined. DFPAU [4] is a FPU unit optimized for Altera FPGAs that supports single-precision addition, subtraction, multiplication, division and square root. DFPAU is well-suited for use with the 32-bit Nios II processor. Each FP function can be included or not in the FPU at a configuration level. The Virtex-5 Auxiliary Processor Unit (APU) [10] is a FPU designed for the PowerPC 440 found in the Virtex-5 FXT family of FPGAs. The FPU supports multiply, add, subtract, divide square-root and fused multiply-add. The fused multiply-add operation is implemented as individual multiply and add operations. Add, subtract and multiplication can be issued on every clock cycle, but divide and square-root are sequential. The unit is fast, but the sequential division and square-root are slow.

In this work we aimed at implementing a reconfigurable FPU for FPGA. The unit can be configured with different sets of single precision floating-point functions from the supported operations, namely, add, multiply, fused multiply-add, reciprocal, square-root and inverse square-root. The FPU is fully pipelined and can issue any function on every clock cycle at low latency. To guarantee a high performance/area ratio, a single multiplier is used to implement both the multiplier and the fused multiply-add, reciprocal, square-root and inverse square-root. Compared to the other state-of-the-art FPU, we consider that our proposed FPU is more appropriate for many-core processing since it is fully pipelined for high performing, with high area efficiency.

II. PREVIOUS WORK
A number of floating point units have been proposed in both ASIC (Application Specific Integrated Circuit) and FPGA (Field-Programmable Gate Array) to be interfaced with some known embedded processors.
III. FUNCTION EVALUATION

The reciprocal, square-root and inverse square-root functions are evaluated using a piecewise polynomial approximation with a minimax polynomial [5].

The \( x \) inputs of the one-operand functions are represented by \( s \)-bit significands, where \( s = 24 \) for the single precision operands, normalized in the interval \([0.5, 1]\).

\[
x = 0.1 x_{-2} x_{-3} x_{-4} \cdots x_{s} \quad (1)
\]

For the standard IEEE operands, normalized in the interval \([1, 2]\), this corresponds to a simple fixed one-bit shift-right at the input of the unit. This also means that, in the case of the reciprocal and inverse square root, the significand of the result will be directly normalized in the interval \([1, 2]\). In the case of the square root a final one-bit shift-left will be necessary to normalize it to the IEEE standard.

Handling the exponent term of the floating-point input is trivial, in all cases. However, for the computation of the square root (and inverse) we must consider different approximations for odd and even exponents, which is trivial, in all cases. However, for the computation of the square root and inverse square root, were experimentally evaluated and showed that the global errors are always lower than \(2^{-28}\), which is more than adequate for supporting the single precision operations with the output faithfully rounded.

The polynomial coefficients are stored in the BRAM as (unsigned) absolute values. Negative values are simply handled by configuring the DSP block with a subtraction instead of an addition. For the three functions considered, the absolute values of the coefficients are bounded by the reciprocal approximation in the first interval, that is \(|a| < 2\), \(|b| < 4\) and \(|c| < 8\), such that the maximum number of integer bits \((I)\) required for each, is \(I_a = 1\), \(I_b = 2\) and \(I_c = 3\).

We determined analytically (see [12] for details) the truncation error upper-bounds for the two multiplication-additions and selected the coefficient formats that minimize the error bounds (in this case, \(e_{T_p} < 2^{-29.3}\)) given the above referred constraints:

\[
a \rightarrow Q1.35, \quad b \rightarrow Q2.21, \quad c \rightarrow Q3.10 \quad (6)
\]

where the fixed-point number format is represented as \(Q[I], F\), where \(I\) is the number of integer bits and \(F\) is the number of fractional bits.

The global errors for the three functions, reciprocal, square root and inverse square root were experimentally evaluated and showed that the global errors are always lower than \(2^{-28}\), which is more than adequate for supporting the single precision operations with the output faithfully rounded.

IV. FLOATING-POINT ARITHMETIC UNIT

The proposed floating-point arithmetic unit supports the fused multiply-add operation (which follows the IEEE guidelines) as well as multiplication, addition, reciprocal, square root and inverse square root (see figure 1).

The unit is composed of a fused pipelined FP multiplier-adder. The hardware in the multiplier part is also capable of calculating a \(2^{nd}\) order polynomial to calculate the one-operand functions. To perform standalone multiplications
IEEE defines a standard for fused FP operations [9]. It states that the fused operation multiply-add, should compute \((A \times B) + C\) as if with unbounded range and precision, rounding only once to the destination format. To achieve the increased precision, we maintain the whole 48 bits from the multiplier to the adder. In the adder we maintain full precision in all operations till the rounding stage.

In the proposed fused multiply-add, the single-precision operands are first unpacked into their signs, exponents and mantissas (the hidden one included). The exponent and the mantissa determination depend on the type of operation and are explained in the following sections. In the final stages we add the mantissas, normalize and round the number. The rounding process is defined by the round to nearest even method.

IV-A. Multiplication/Function Mantissa Computation

The block for calculation of the mantissa is essentially composed by two FPGA DSPs and some additional logic (see figure 2). The two DSP blocks are (dynamically) configured to alternatively compute a \(24 \times 24\) mantissa product or the required approximating polynomial of the selected one-operand function. The 2\(^{nd}\) order polynomial approximation is implemented, using the 2 DSP blocks, according to equations (4) and (5).

IV-B. Multiplier/Function Exponent Computation

The calculation of the exponent also depends on the function being calculated. For multiplication, we must add the two exponents and subtract the bias. The square root exponent is determined by dividing the input exponent by 2, which corresponds to one-bit right shift (after subtracting the bias). To calculate the reciprocal and to invert the square root the exponent must be negated. As for the square root, the bias is first removed and then added again at the end.

IV-C. Multiplier/Function Normalization

The last stage of the multiplier block is the normalization stage. The mantissa of multiplication result is in the fixed-point format \(Q2.46\). If the most significant bit is ‘1’, the mantissa is shifted right (and the exponent is adjusted) to normalize it into the \(Q1.47\) format. The mantissa which results from the square root has always to be left shifted one bit to be normalized as the input of the operation was previously right shifted one bit.

IV-D. Floating-point Adder

The FP adder consists of swap, alignment, mantissa addition and normalization. The swap module compares the exponents to find the operand with the smaller exponent. This determines which operand has to be aligned (avoiding the cost of two shifters) and by what amount. The bigger exponent will be the exponent of the result. The output of alignment module is the 48-bit shifted mantissa and the bits to be discarded, which are used to calculate the sticky bit used to correctly round the result.

The mantissa adder module adds two 48-bit integers. The sticky bit is calculated in parallel with the addition of the mantissa by doing a logical OR of the discarded bits.

The normalizing module normalizes the result mantissa to the format \(Q1.47\). A leading zeros counter determines the amount by which the result mantissa is shifted left. Accordingly, the exponent is subtracted by the shift amount.

IV-E. Rounding

The final rounding module determines if the result has to be rounded or not based on the round to nearest scheme. The result is then rounded to the single-precision format by adding the rounding bit to the 23 most significant bits of the mantissa concatenated with the exponent. This way, the exponent is adjusted if an overflow occurs and the mantissa remains correct.

V. RESULTS

All designs were implemented in a Virtex-7 FPGA (XC7V585T-1). The multiplier is the common operator for all functions. We have compared our implementation with those from CoreGen and FloPoCo [11] (see table I)
Compared to the multiplier from FloPoCo, our multiplier is about 30% smaller but about 10% slower. Compared to Coregen, ours is about 10% faster but about 30% smaller. In fact, all three solutions represent different pareto points in a performance/area design space.

We have also implemented the FPU with several combinations of functions, namely the fused multiply-add, multiply and reciprocal, multiply and elementary functions (see results in table II).

The fused multiply-add unit with the lowest latency has a frequency of 326 MHz, lower than that of a single multiplier. The decrease in frequency is due to the 48-bit adder. Increasing the latency of the adder four cycles increases the frequency about 50%. The CoreGen achieves similar results with an extra cycle and a extra DSP that reduces the number of LUTs. FloPoCo has 50% more LUTs and more four DSPs, than our solution with lowest latency. The frequency is also smaller but the latency is four cycles less. We estimate that, for the same latency and technology, the frequencies would be close. The PCS-FMA implementation has only five cycles of latency but consumes six times more LUTs and more four DSPs. Using a performance/area efficiency metric - Freq/DSP/Latency - ours and the reciprocal only implementation due to the multiplexers needed to route the data inputs to the DSPs.

A comparison between our implementation of the FPU with multiply, square-root and inverse square-root and the square-root implementations from CoreGen is similar to the reciprocal comparison. Our unit works with a lower frequency but with only 9 cycles of latency, while the CoreGen functions have latencies of 27 and 35.

## VI. CONCLUSION

A configurable FPU with high performance and area efficiency was presented. The FPU supports addition, multiplication, fused multiply-add, reciprocal, square-root and inverse square-root. The multiplier and the $2^{nd}$ order polynomial approximation share the mantissa multiplier implemented with two DSP of the FPGA. The results show that the performance of the unit is competitive with that of independent implementations of each function.

## ACKNOWLEDGMENT

This work was supported by national funds through FCT, Fundação para a Ciência e Tecnologia, under projects PEst-OE/EEI/LA0021/2013 and PTDC/EEA-ELC/122098/2010.

## VII. REFERENCES


### Table I. Results for single-precision multiplier

<table>
<thead>
<tr>
<th>Function</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>Latency</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoreGen</td>
<td>128</td>
<td>160</td>
<td>2</td>
<td>8</td>
<td>459 MHz</td>
</tr>
<tr>
<td>FloPoCo</td>
<td>271</td>
<td>270</td>
<td>2</td>
<td>8</td>
<td>590 MHz</td>
</tr>
<tr>
<td>Ours</td>
<td>183</td>
<td>249</td>
<td>2</td>
<td>8</td>
<td>525 MHz</td>
</tr>
</tbody>
</table>

### Table II. Results of the FPU for several sets of functions

<table>
<thead>
<tr>
<th>Function</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM</th>
<th>Latency</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiply+Add</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ours</td>
<td>967</td>
<td>968</td>
<td>3</td>
<td>0</td>
<td>15</td>
<td>326 MHz</td>
</tr>
<tr>
<td>CoreGen</td>
<td>989</td>
<td>1270</td>
<td>3</td>
<td>0</td>
<td>19</td>
<td>493 MHz</td>
</tr>
<tr>
<td>FloPoCo</td>
<td>802</td>
<td>1233</td>
<td>4</td>
<td>0</td>
<td>20</td>
<td>488 MHz</td>
</tr>
<tr>
<td>PCS-FMA² [8]</td>
<td>5832</td>
<td></td>
<td>21</td>
<td>0</td>
<td>5</td>
<td>531 MHz</td>
</tr>
</tbody>
</table>

| **Reciprocal**            |     |    |     |      |         |           |
| Ours                      | 302 | 327| 2   | 1    | 9       | 435 MHz   |
| CoreGen                   | 149 | 404| 2   | 0    | 27      | 414 MHz   |

| **Multiply+SQRT+InvSQRT** |     |    |     |      |         |           |
| Ours                      | 453 | 541| 2   | 1    | 9       | 370 MHz   |
| SQRT²                     | 442 | 929| 0   | 0    | 27      | 529 MHz   |
| ImSQRT²                   | 237 | 561| 9   | 0    | 15      | 429 MHz   |

²Generated with CoreGen

The fused multiply-add unit with the lowest latency has a frequency of 326 MHz, lower than that of a single multiplier. The decrease in frequency is due to the 48-bit adder. Increasing the latency of the adder four cycles increases the frequency about 50%. The CoreGen achieves similar results with an extra cycle and a extra DSP that reduces the number of LUTs. FloPoCo has 50% more LUTs and more four DSPs, than our solution with lowest latency. The frequency is also smaller but the latency is four cycles less. We estimate that, for the same latency and technology, the frequencies would be close. The PCS-FMA implementation has only five cycles of latency but consumes six times more LUTs and more 18 DSPs. Using a performance/area efficiency metric - Freq/DSP/Latency - our fused units are the best among all.

Our FPU with a multiplier and reciprocal has a latency of 9 cycles at 372 MHz. A reciprocal implementation with CoreGen has 27 cycles of latency at 414 MHz and uses 8 DSPs. Compared to the CoreGen reciprocal, ours uses twice the number of LUTs but only a quarter of the DSPs, is slightly faster at only one third of the latency. The algorithms used to calculate the reciprocal are different and this explains the disparity in the resources used. Our reciprocal plus multiply implementation is slower than the reciprocal only implementation due to the multiplexers needed to route the data inputs to the DSPs.