A Flexible Floating-Point Format for Optimizing Data-Paths and Operators in FPGA Based DSPs

J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria
Microelectronics Research Group
Ecole Polytechnique of Montreal
CP 6079, Succ. Centre Ville
Montréal (Qc) H3C 3A7 Canada
(514) 340-4711 #4474
loiseau@grm.polymtl.ca; savaria@vlsi.polymtl.ca

D. Poirier
Miranda Technologies, Inc.
3499, Douglas B. Floréani
Ville Saint-Laurent (Qc) H4S 2C6 Canada
(514) 333-1772
dpoirier@miranda.com

ABSTRACT
Video signal processing requires complex algorithms performing many basic operations on a video stream. To perform these calculations in real-time in a FPGA, we must use innovative structures to meet speed requirements while managing complexity. As part of a project aiming at the development of a video noise reducer, we developed an optimized processing stream that required some floating-point calculations. This paper presents the rationale for developing a floating-point unit, justifies the data representation used, its implementation in a Xilinx VirtexE FPGA and reports the performance we obtained. A divider using this representation is also presented, with its implementation and performances in the same FPGA.

Keywords
Floating-point, FPGA, Video-Processing, Hardware Optimization, Data-Path Optimization, Hardware Division, Floating-point / Fixed-point Conversion

1. INTRODUCTION
In high definition television (HDTV), the size of images and the amount of information in the video signal are very important. HDTV requires a 600 Mbit/s data flow after decompression. HDTV images are meant to be of very high quality, but there is a strong requirement to compress data for more effective storage and to limit bandwidth requirements. Thus, there is a need for methods that mitigate the impact of compression/decompression artefacts. Good quality algorithms for that task are available, but they impose very heavy computational requirements. The algorithm we have been studying requires more than 300 mathematical and logical operations per pixel, which translates to required throughputs in excess of 20 Giga operations per second. This is beyond the capabilities of existing DSPs and achieving real time performance sets a requirement for hardware implementation. When such algorithms are to be implemented in FPGAs, we must take care of 2 aspects:

- The hardware complexity (number of slices in a Xilinx Virtex FPGA, and size of embedded memories) required to implement the function, in order to control the cost,
- The processing time for real-time applications. This becomes a hard constraint if the selected FPGA has difficulty meeting the required throughput (75MHz for example).

The throughput can generally be enhanced with pipelined structures. However, pipelining increases latency, which reflects in requirements for many internal registers. This can be costly when retimed data flows must be synchronized. In addition, computational structures with large latencies can also be very inefficient when used inside loops. In this case, the overall system throughput may be as low as if there was no pipelining.

The complexity of the target algorithms can set requirements for high precision arithmetic (20 bits or more). At high precision, operators are complex and slow, which further increases the requirement for heavy pipelining. A solution to manage the complexity, latency, and accuracy tradeoff is to use a floating-point representation when required. Necessity, based on overall system accuracy simulations, led us to develop floating-point units as a means to control hardware complexity and length of the critical paths. The rest of the paper reports on the development of a configurable floating-point unit implemented in a Xilinx VirtexE FPGA. Configurability is a very flexible means to manage the precision requirements since we can control it all along a data-path. All this work was performed using a specific type of FPGA devices, which is our target technology for the final implementation. The justification for using FPGAs as a preferred target technology in this research comes from industrial requirements. The work was necessity driven, with the objective to produce data processing modules that can be used in hardwired data-paths, which can generate studio-quality decompressed video streams. Successful commercial products in that category are not produced in high volumes, and since they implement leading edge DSP algorithms that keep improving, their lifetime is rather short, as they are replaced with better algorithms that are nevertheless composed of the same basic functional modules, often needed in slightly different flavors (precision and dynamic range requirements). The concepts presented here could be used in a floating-point unit module generator that would be a very useful addition to existing module generator libraries. Clearly, the underlying concepts are applicable to a variety of FPGAs, and
could also be useful in the development of cost effective ASICs that have similar requirements to tightly manage accuracy, precision, hardware complexity, high data rates, and low latency by controlling data representation, to produce cost effective data-paths. Of course, cost effective also means low design time, which is a byproduct of the improved reusability of configurable modules.

The rest of this paper presents two innovative techniques for reducing complexity and improving performances of hardware implementations using floating-point formats in FPGAs. The proposed technique consists in:

- Using a configurable floating-point format, with a reduced bit-width, to adjust data-path structure to algorithm requirements,
- Using a popular invert-and-multiply approach for division, but with significant optimization made possible by an original use of a floating-point representation to code the inverse.

2. FLOATING-POINT UNIT

Generally, a floating-point representation allows, for a fixed number of bits, to represent a larger range of values with a slightly smaller number of bits. That motivates its use in our application, because we want to limit the number of bits used to represent a given data range.

IEEE 754 Standard [1] deals with floating-point representation and proposes the following data format, for a single-precision number:

\[ (-1)^S \times 1.M \times 2^{E-127} \] (1)

- S : Sign bit,
- M : Mantissa, coded with 23 bits,
- E : Exponent, coded with 8 bits.

In this representation, the mantissa is always normalized, and its most significant bit is always 1. Thus, the MSB is not directly coded in M, but it is implied in the representation (1.M). One of the main advantages of following this standard representation is its support in most floating-point processors and high level language compilers. This is useful for portability, particularly if an algorithm is first validated in a software based development environment. However, this representation was not used.

Different papers deal with the implementation of operators using the IEEE Standard floating-point representation, both within ASICs or FPGAs. For example, the SPARC processor [10] has a floating-point unit performing operations on single or double-precision numbers. IBM's POWER2 FPU [6] was specially designed to limit latency and maximise throughput. A detailed comparison with [6] is difficult since the required hardware is not quantified in the reference. Nevertheless, the proposed solution appears complex to implement. A CMOS FPU [3], implementing addition, subtraction and multiplication, requires an entire 330000-transistor chip. It is IEEE 754 Standard compliant, and its complexity is mainly due to the large data size supported (for example, the multiplier takes 40% of the chip). This complexity is hard to tolerate in data-paths that require hundreds of operations every clock cycle, and where several such operations end up requiring a floating point representation.

Concerning implementations in FPGAs, a few papers deal with this [4], [7], [9] and show the implied complexity. These papers demonstrate that it is feasible to implement a IEEE 754 compliant floating-point unit in a FPGA, but that this representation produces bulky and slow processing units requiring heavy pipelining. In spite of the availability of million gates FPGAs, supporting IEEE floating-point standard is costly and often not required.

The objective of this work is not to compete with the highly optimized general purpose floating-point unit of modern microprocessors, but simply to demonstrate how we can cost-effectively take advantage of the dynamic range of a floating-point representation, as compared to the fixed-point representations generally used in special purpose data-paths, particularly when they are implemented in FPGAs. For instance, a 23-bit mantissa leads to a high complexity in multipliers or dividers. This is particularly inefficient in video processing, when the objective is to produce an output image with 8-bit accuracy, also known as studio-quality. A format derived from the IEEE standard was proposed [8]. It consists in a 16-bit representation, with sign bit, 6-bit exponent and 9-bit mantissa. The operators used are a lot less complex than the one required by the IEEE floating-point standard. However, it is specifically designed for an arithmetic processor, and thus, the format is not configurable and does not allow data-path configuration. Since in a dedicated optimized structure, we want to strictly manage the size of every data, a hard-coded representation does not fit our requirements.

Also in [2], a 32-bit representation derived from IEEE is presented and uses a base-8 exponent, representing a range larger than the standard IEEE format. It also leads to a monotonous increase of the mantissa for both positive and negative numbers to facilitate comparison. However, the objective of this work is to optimize code execution in a general-purpose processor, which is completely different from the work reported here. In their application (software for automotive control), comparison is the most used operator, and thus, their focus is to optimize it. Our targeted algorithm requires multiplication and division, but no floating-point comparisons.

These considerations guided us in the development of a configurable floating-point format. With this format, we can choose the number of bits used to code the mantissa and the exponent. Due to the requirements of our application, the chosen representation is not signed.

The number is coded with the following formula:

\[ M \times 2^{+E} \] (2)

Depending on the needs of the application, the exponent could be signed or not.

One of the peculiarities, in comparison with the IEEE standard, is that the mantissa is not always normalized. A normalized representation is useful for comparisons, but does not give a competitive advantage for divisions and multiplications.

The following examples show the advantages of this representation for a general use that can be adapted to the needs of the application.
Given 0100010000000000 to code with an 8-bit mantissa.
The result is 10001000 . 2^7. In that case, the mantissa is normalized.

On the other hand, given 0000000000100010 to code.
The result with a normalized mantissa would be 10001000.2^-2.
The result with our floating-point unit is 00100010.2^0.
The two results are of course completely equivalent, but the second one does not artificially add zeros to the mantissa, and it can avoid the use of a signed representation to code the exponent.

If M is an integer and E is positive, our processing unit encodes scaled integers (the exponent is positive), using the notion of floating-point. However, if E is a signed number, the proposed format can represent fractional values.

For example: given 000001000.100101 to code with an 8-bit mantissa (equivalent to 1000100101.2^-6), we get 10001001.2^-2 (equivalent to 10001001.2^-6+4). Note that in a dedicated hardware unit, the -6 can be virtual and not implemented. It is merely a matter of how the designer interprets the output produced by the hardware. Thus equation 2 is a general floating-point representation.

In many signal-processing applications, most of the data to process is fixed-point. Also, output data is often fixed-point too. This is the case with image and video processing. Thus, we need conversion modules: fixed-point ⇔ floating-point, to embed required floating-point units in DSP chains.

![Figure 1 - Floating-point system](image)

In this paper, the required conversion units will be called FIXEDtoFLOAT and FLOATtoFIXED.

### 2.1 FIXEDtoFLOAT Module

#### 2.1.1 Implementation

The FIXEDtoFLOAT module analyses the data to convert in floating-point format, and according to its value, makes an intelligent routing of significant bits. The following discussion assumes an 8-bit mantissa and is easily generalized to other sizes.

By default, the module starts by determining the position of the most significant bit that is at 1, and selects the rest of the mantissa bits on that basis. The exponent is the number of bits after the least significant bit kept for the mantissa.

For example,

\[
0|10001000|0010000|0100010000000000
\]

\[
M=10001000 \quad E=7
\]

When no bit more significant than the 7th is at 1, the 8 least significant bits are kept as mantissa. The following table illustrates this algorithm in detail in the case where a 15-bit input is mapped to an 8-bit mantissa and 3-bit exponent representation.

<table>
<thead>
<tr>
<th>Input Range</th>
<th>M, E</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0;255]</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>[256;311]</td>
<td>0000001XXXXXXX0000</td>
</tr>
<tr>
<td>[512;1023]</td>
<td>000001XXXXXXX0000</td>
</tr>
<tr>
<td>[1024;2047]</td>
<td>00001XXXXXXX0000</td>
</tr>
<tr>
<td>[2048;4095]</td>
<td>0001XXXXXXX0000</td>
</tr>
<tr>
<td>[4096;8191]</td>
<td>001XXXXXXX0000</td>
</tr>
<tr>
<td>[8192;16383]</td>
<td>01XXXXXXX0000</td>
</tr>
<tr>
<td>[16384;32767]</td>
<td>1XXXXXXX0000</td>
</tr>
</tbody>
</table>

**Table 1 – Conversion table for FIXEDtoFLOAT module**

#### 2.1.2 Performances

This module was implemented in the VHDL language, synthesized with Synplify, placed and routed on a VirtexE XCV50E-6 FPGA from Xilinx. In this implementation, the number of bits in the mantissa and the exponent are generic parameters.

In the case where a 15-bit input is mapped on an 8-bit mantissa and 3-bit exponent representation, 35 slices are required, and based on a worst-case timing analysis of the design, it can operate at 105 MHz.

### 2.2 FLOATtoFIXED Module

#### 2.2.1 Implementation

This module takes floating-point data and returns it to a fixed-point format.

Of course, it performs exactly the opposite operation to the FIXEDtoFLOAT module. The following table describes in detail the bit routing performed by the FLOATtoFIXED unit, when an 8-bit mantissa, 3-bit exponent representation is returned to a 15-bit fixed-point representation.

<table>
<thead>
<tr>
<th>E</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>2</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>3</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>4</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>5</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>6</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>7</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>

**Table 2 - Conversion table for FLOATtoFIXED module**
This module routes the mantissa bits to the appropriate scale and does the required zero filling.

2.2.2 Performances
This module is also configurable during the synthesis step for the number of bits of the mantissa, exponent and output.

Using the same 8, 3, and 15 bit values; after synthesis, placement, routing, and timing analysis, we got: a 30-slice complexity and a 85 MHz operation frequency with a VirtexE XCV50E-6 (and 101 MHz for a XCV50E-8).

2.3 Performances Comparisons With FPGA-based Implementations
Concerning the complexity, a multiplier processing 8-bit mantissa requires 34 slices, to be compared with 290 slices required for a basic 23 bits by 23 bits multiplier required in a Standard IEEE floating format. The benefit we get largely comes from reduced bit width; however, the adopted representation provides exactly what is required for our application.

By comparison, a FPU performing addition, subtraction and multiplication requires an entire Xilinx XC4044XL (3800 CLBs, equivalent to 950 Virtex's slices) [7], and 60% of that complexity is required to implement the multiplication. Similarly, the CMOS FPU reported in reference [3] uses 40% of its area to implement a multiplier.

3. MATHEMATICAL INVERSION MODULE
The floating-point representation is particularly interesting when applied to a divider, because it allows to limit the number of bits, while keeping a constant output accuracy, independent of the value of the number by which we want to divide. Also, it is difficult to implement fast division of large bit resolution numbers.

A popular approach to carry out a division is to:

- calculate the inverse of the denominator, with a floating-point representation,
- multiply this inverse by the numerator.

This method was also used in [8], but the authors did not take advantage of the floating-point representation to code the inverse and thus, the data width of the following multiplication is not optimized.

To the best of our knowledge, this paper is the first to propose using a floating-point representation to perform the inversion as a means to limit the hardware complexity. This method is very well adapted to the dynamic range produced by the division operator.

The obtained result is in floating-point format that is either put back in fixed-point format or let in floating-point format for further operations.

Compared to an iterative division algorithm [5], this method can produce a result every clock cycle with the required high throughput without resorting to heavy pipelining.

In this paper, we focus our attention on the inverter part. A classical multiplier (such as the one we get with a straight “*” operator in VHDL) can be used to complete the division.

The following example shows the interest of using a floating-point representation to code an inverse:

Given an integer to invert. The inverse is coded using the following representation:

\[ \frac{2^{-1} \cdot 2^{-2} \cdot 2^{-3} \cdot 2^{-4} \cdot 2^{-5} \cdot 2^{-6} \cdot 2^{-7} \cdot 2^{-8} \ldots}{X} \]

The number of bits used depends on the desired inverse accuracy.

The inverse of a number is calculated by:

\[ \text{Inv} = \text{Round}(\frac{2^N}{X}) \]  \hspace{1cm} (3)

- Round, rounds to the nearest value,
- X : number to invert,
- N : number of bits to code the inverse.

The round function is used to minimize the quantization error to ±1/2 LSB.

For example, with a 6 bit number to invert and N=13:

- 1/5 is coded 0011001100110,
- 1/62 is coded 0000010000100,
- 1/63 is coded 0000010000010.

To get a good accuracy, we must keep a lot of bits to code the inverse. It takes the 11th bit to differentiate between 1/62 and 1/63. Moreover, even with 11 bits, the precision of the result is only 6 bits, because the leading zeros do not add precision, but only indicate the weight of the result. With this representation, we would get an excellent precision when we invert small numbers, but low precision when we invert large numbers. Moreover, the multiplication following the inversion (to complete the division) would require large operands (11 bits in this case, for a 6-bit accuracy).

For our inverter, we calculate separately the significant part of the inverse, i.e. the part beginning by the first MSB equal to 1, and the number of shifts to do:

For example 1/63: \[ \begin{array}{c|c|c}
\text{Nb of leading zeros} & \text{Significant part} & \text{of the inverse} \\
5 & 1000010 & 130 \\
\end{array} \]

With floating-point, the precision in bits is constant for any value of the number to invert. Moreover, we limit the number of bits required to code the inverse, and, therefore, the multiplication is made less complex and faster.

Note that the inverter can deal both with floating-point and fixed-point data. It is originally hybrid, receiving fixed-point data and producing floating-point results. To process floating-point data, we have to ensure that we invert the entire mantissa. If not, we have to normalize the mantissa and keep the required number of MSB (to prevent precision problems). When the mantissa is normalized, we must subtract the number of zeros of the result.
from the exponent of the input. Moreover, the numerator can be either fixed-point or floating-point numbers.

3.1 Implementation
The design is targeted for FPGA implementation. As the number to invert in our application is not very large (6 bits), the significant part of the inverse is efficiently coded in a ROM used as look-up table (practically in the LUTs of the FPGA), and the number of zeros (shift) is calculated using logic. The required precision demands that the significant part of the inverse be coded with 8 bits. The number of shifts, limited to 5, requires a 3-bit encoding. These parameters fit the need of our application, but the inverter is fully configurable for the size of the input and the result. Note that if we want to invert larger numbers, it becomes more interesting to use RAM blocks than LUTs to implement the ROM. Of course, this also depends on the available resources in the FPGA used.

The divider receives a 6-bit input and produces an 8-bit output mantissa accurate to 1/2 LSB (assuming that the 6-bit input is an exact integer value, not previously rounded). It introduces two pipeline stages. It was placed and routed on a Xilinx VirtexE XCV50E-6 FPGA, and analyzed with the previously described tool. The content of the ROM is calculated during the synthesis step using formula (3) from which we remove the most significant zeros. The calculation of the number of shifts is decoded from the input data using the following table:

Table 3 – Conversion table to calculate the number of shifts

<table>
<thead>
<tr>
<th>Shift</th>
<th>N/A</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>[1;2]</td>
<td>[3;4]</td>
<td>[5;8]</td>
<td>[9;16]</td>
<td>[17;32]</td>
<td>[33;63]</td>
<td></td>
</tr>
</tbody>
</table>

3.2 Performances

The divider receives a 6-bit input and produces an 8-bit output mantissa accurate to 1/2 LSB (assuming that the 6-bit input is an exact integer value, not previously rounded). It introduces two pipeline stages. It was placed and routed on a Xilinx VirtexE XCV50E-6 FPGA, and analyzed with the previously described procedure. According to timing analysis, it can be clocked at 145 MHz while requiring only 31 slices.

The use of the inverter and a basic 8-bit non-pipelined multiplier ("*" of the VHDL language) produces a division with 8 bit accuracy in 3 clock cycles latency, and this module operates at 80 MHz (95 MHz with a XCV50E-8). The multiplier is the limiting speed factor. This division module (inverter and basic multiplier) fits in 65 slices. This corresponds to the basic configuration of the following table.

If we want to improve the performances of the multiplication stage, we can use a pipelined multiplier. With 3 pipeline stages for this multiplication, the clock frequency can reach 140 MHz, but we need more slices (total of 80 slices for the inverter and the pipelined multiplier). This is reported as the fast rate configuration in the following table

If the division module was implemented with fixed-point arithmetic using the same approach, a 13-bit output precision would be required in the inverter and the multiplier would be a 13 by 8 multiplier, which is significantly larger and slower. Of course, pipelining would allow reaching the same processing rate with integer arithmetic, but the additional latency and complexity is not desirable.

We can compare our implementations to a fixed-point division core provided by Xilinx, using the CoreGenerator tool. The following table gives the results using the above parameters (size of the data) with the same device, and the same timing constraints.

Table 4 – Performances comparison

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Basic</th>
<th>Fast rate</th>
<th>Xilinx’s Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency (cycles)</td>
<td>3</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>(ns)</td>
<td>37.5</td>
<td>35.7</td>
<td>85.9</td>
</tr>
<tr>
<td>Throughput (result/cycle) (rate in MHz)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Complexity (slices)</td>
<td>65</td>
<td>80</td>
<td>98</td>
</tr>
</tbody>
</table>

The overall latency of the fast rate configuration is slightly lower than the basic one. These results produced by the timing analysis tool are somewhat counterintuitive, since the structures are similar.

Our floating-point implementation offers a very interesting low latency. Depending on the throughput required, we can use either the basic or the fast rate configuration, which always produce an overall latency that is much smaller than the Xilinx Core. However, when it is necessary to divide large numbers, such a Core is more effective (smaller, faster), since our look-up table grows exponentially with the input size. Our architecture was developed for applications that need mantissa up to 10 bits.

As an additional reference point, the implementation of an IEEE 754 standard compliant division in the SPARC microprocessor has a latency of 12 clock cycles and can produce one result every 12 cycles for a single-precision division. Of course, there are significant differences between a SPARC and a FPGA (clock rate, custom design, semi-conductor process…), but it confirms that even a powerful processor requires long latency for a division operation.

4. CONCLUSION
Floating-point representation for FPGA based DSP is often considered impractical. By contrast, this paper demonstrates that a suitably adjusted floating-point representation is a good means to effectively implement in hardware effective custom DSP data-paths that maintain good output accuracy, while keeping moderate bit widths. This innovative use of a configurable floating-point format is thus a tool to meet accuracy-complexity-performance targets. It can be used selectively when the complexity of an algorithm starts eroding precision. This is particularly true when an algorithm processes data over wide dynamic ranges, a situation often provoked by the division operator. Our conclusion thus sharply contrasts with the conventional wisdom that floating-point processing necessarily translates to highly complex and slow data-paths.

5. ACKNOWLEDGMENTS
The authors would like to thank Micronet for their financial and technical support.
6. REFERENCES


