FPGA Implementation of a Custom Floating-point Library

Nelson Campos[1], Eran Edirisinghe[2], Shaheen Fatima[1], Slava Chesnokov[3], and Alexis Lluis[4]

[1] Loughborough University, Loughborough, United Kingdom,
`N.C.S.Campos@lboro.ac.uk, S.S.Fatima@lboro.ac.uk`
[2] Keele University, Keele, United Kingdom,
`E.Edirisinghe@keele.ac.uk`
[3] Imaging-CV Ltd, London, United Kingdom,
`slava@chesnokov.org`
[4] ARM Ltd, Manchester, United Kingdom
`alexis.lluis@arm.com`

Abstract. This paper presents FPGA implementation of a floating-point library for high-performance video processing. The library comprises functions including addition, multiplication, division, square-root, exponentiation and logarithm, as well as floating-point to fixed-point and fixed-point to floating-point conversion. We implement a set of composite functions using this library to compute custom floating-point arithmetic using an Artix-7 FPGA. The synthesis of the library using the software toolkit Vivado from Xilinx is performed and compared with the open-source floating-point library Flopoco. The synthesized library has a maximum latency of 18 cycles to process 20 composite floating-point functions in parallel running at a clock frequency of 148.5 MHz.

Keywords: Floating-point arithmetic, FPGA, real-time, VLSI.

1 Introduction

Image and video processing have become popular applications for Field Programmable Gate Arrays (FPGAs). The necessity to meet real-time processing and low-power consumption are contributory factors to this popularity . The fine-grained nature of FPGAs enables them to exploit parallelism, which leads to acceleration of complex tasks. However, the inherent fine-grained features make the programming of FPGAs difficult with a high code density in the implementations. As a result, when implementing hardware architectures, often not only the algorithm is required to be described, but as well the basic arithmetic operations that compose them[26][3].

Fixed-point arithmetic is the preferable data representation when dealing with hardware implementations with the advantage of saving logic resources when compared with floating-point designs. However, the compactness of algorithms resulted from fixed-point format is penalized by the additional numeric manipulations to compensate the quantization error introduced during the conversion from floating-point to fixed-point [21]. When the application requires high precision and dynamic range, floating-point fits better as the data representation, reducing the development time of a product due to its ease of implementation. A customizable floating-point results in efficient implementations controlling the dynamic range and precision with the necessary number of bits in the exponent and mantissa fields, respectively leading to compact architectures and maintaining the features (range and precision) of the standard floating-point format.

MathWorks shows in [18] the advantages of using floating-point over fixed-point representations in FPGA and ASIC designs using the expression $y = \frac{(1-x)}{(1+x)}$. If the floating-point format is applied to compute the value of y, no additional manipulation is required to avoid overflow and the operation executes with high precision and dynamic range with all the data types having the same number of bits. However, to perform the same expression in fixed-point format, the following steps are computed: the division is executed with the product of the numerator with the reciprocal of the denominator, where the reciprocal operation is computed using approximation methods such Newton-Rapshon or lookup tables (LUTs). The numerator and denominator use different data types to control overflows resulted from the subtraction and addition. Although Matlab provides the HDL Coder to generate hardware implementations using the floating-point format, the data representation uses single-precision, which leads to additional resource usage if the application needs less precision and range.

A set of commercial floating-point arithmetic hardware cores are available in the market, which provides high-level customization. Xilinx LogiCORE [25] is a floating-point library, which enables the specification of floating-point functions such as adders and multipliers, logarithms and exponentiation. The exponent and mantissa fields are customized (the exponent bit-width is from 4 to 16 bits; the mantissa from 4 to 64 bits). Similarly, Altera [2] provides floating-point functions with the single-extended precision format (inputs between 43 and 64 bits, the exponent field has at least 11 bits; the mantissa has a minimum of 31 bits). Synopsys [23] also provides floating-point libraries for FPGA and ASIC design flow, with a flexible exponent field ranging from 3 to 31 bits and a mantissa, with width varying from 2 to 256 bits. Another popular state-of-the-art in both industry and academia is FloPoCo (Floating-PointCores) [9], an open-source C++ framework for generating arithmetic cores in VHDL. FloPoCo auto-generates pipelined modules for addition, multiplication, division, square-root, exponentiation, among other functions. The exponent and mantissa fields are freely-chosen by the user, enabling customized floating-point arithmetic according to the application needs.

Our focus in this work is to apply floating-point operations in high-resolution real-time video processing using FPGAs. Many applications require the mathematical operations of division and logarithms. For instance, the detection of variation in images of the same scene taken in different times has a large number of applications in video surveillance, remote sensing and medical diagnosis

[20][7][4][24]. A division of two frames can detect variation in the scene, where each floating-point pixel of the first scene is divided by each floating-point pixel of the second scene. Logarithms are also widely used in image processing applications, for example, in real-time skin segmentation or in the estimation of the likelihood scores estimation for Gaussian Mixture Models [1][10]. Square-roots are also present in many advanced algorithms in real-time image processing[13]. Exponential functions have applications in non-linear image interpolations using B-splines [15]. Although image processing operations using floating-point have straightforward implementations using a high level of abstraction language such as Python, software implementations often do not meet the real-time requirement for high-resolution images. In addition to that, the excessive power consumption justifies embedded applications with FPGAs or ASIC.

A reconfigurable framework in run-time execution is a desirable feature in many applications. Serial buses such as UART, I2C and SPI are the most common protocols used to interconnect embedded systems and provide reconfigurability. The serial connections allow transmission of data in high frequencies with a reduced cost of pin counts with simple implementations of the protocols [19]. The integration between a host computer and processing elements in FPGAs enable the run-time reconfiguration of architectures. This integration is useful when a user-friendly graphical interface (GUI) needs to send a bank of registers to select an operation in an image processing environment [6]. Another application is when there is a need to read a pixel at a specific position in real-time. Therefore, the on-the-fly reconfigurability resulted from the PC-FPGA bridge reduces time in hardware implementations. Moreover, it allows a high-level verification of the functionalities and fast prototyping of the framework.

To summarize, the main contributions of this paper are:

- a multi-precision floating-point library for high-performance video processing comprising functions of addition, multiplication, division, logarithm, square-root, exponentiation, floating-point to fixed-point and fixed-point to floating-point conversion;
- the synthesis of the library using the software toolkit Vivado and the comparison of the results with the open-source floating-point library FloPoCo;
- a graphical interface, which allows the communication of the FPGA with a computer, enabling the writing of floating-point registers and reading of floating-point pixels in real-time.

The rest of this paper is organized as follows. Section 2 provides an overview of the proposed floating-point library and the image quality evaluation of the framework. Section 4 discusses the results of synthesis of the proposed framework. Finally, section 5 concludes this work.

2 Proposed Floating-point Library

This section gives an overview the proposed architectures and implementation of the floating-point library. Subsection 2.1 describes conversion between floating-point and fixed-point. Subsection 2.2 covers the implementation of the floating-point adder and multiplier and subsection 2.3 discusses the implementation and architectures of the functions (division, logarithm, square-root and exponentiation) based on the polynomial approximation.

2.1 Conversion between floating-point and fixed-point

Let x and X be the representations of a floating-point and a fixed-point number, respectively. According to equation 1, x is a function of the sign s, the exponent e and the mantissa m. On the other hand, X is a function of M integer bits, N fractional bits, where a_k and $b_k \in \{0, 1\}$ and X also is represented in 2's complement, i.e., $(-X) = 2^{M+N+1} - X$.

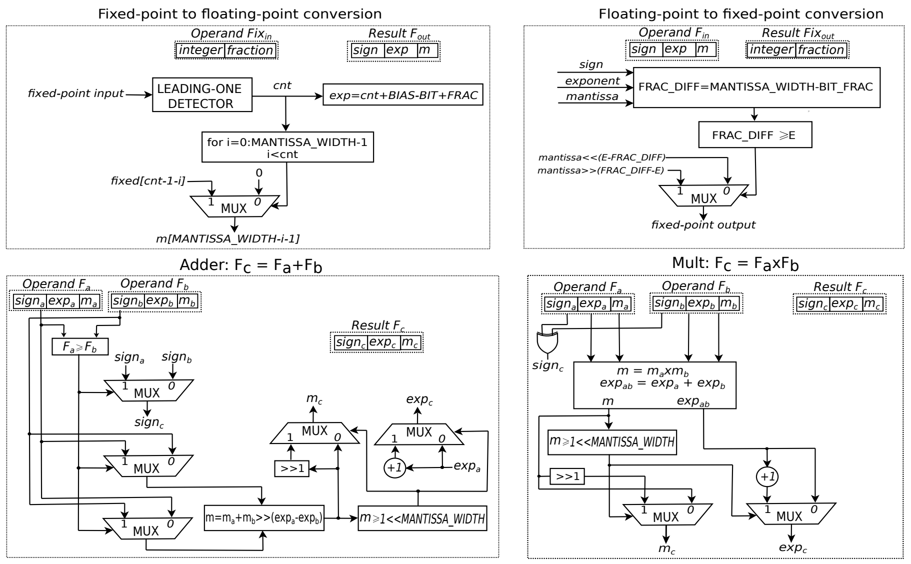


Fig. 1. Architectural descriptions of the floating-point adder, multiplier and conversion between floating-point to fixed-point.

$$\begin{cases} x = (-1)^s \times 2^e \times 1.m \\ X = \sum\limits_{k=0}^{M} a_k 2^{N+k} + \sum\limits_{k=1}^{N} b_k 2^{N-k} \end{cases} \tag{1}$$

Let $\chi : x \to X$ be a function to convert a floating-point to a fixed-point number. χ can be defined as $X = \chi(x) = x \times 2^N$. The implementation of χ is a shift of the mantissa with $e + N$ bits. Similarly, let $\chi^{-1} : X \to x$ be a fucntion to convert a fixed-point to a floating-point number. The determination of χ^{-1} is done by finding the most signifcant bit (MSB) of X using a leading-one detector to compute the exponent. After finding the exponent, the mantissa is computed truncating X to its MSB position. An overview of the architectures to computer χ and χ^{-1} is shown in figure 1. Note that the determination of the signs of x and X is done by checking if the bit $s = a_M$ is zero (positive) or one (negative).

2.2 Floating-point adder and multiplier

Given two floating-point numbers $x = (-1)^{s_x} \times 2^{e_x} \times 1.m_x$ and $y = (-1)^{s_y} \times 2^{e_y} \times 1.m_y$, if $|x| \ \geq |y|, than$x$\pm y$ has the form of the equation 2.

$$x \pm y = (-1)^{s_x} \times \left(1.m_x \pm \frac{1.m_y}{2^{e_x - e_y}}\right) \times 2^{e_x} \tag{2}$$

The steps required to compute $x \pm y$ are described as follows. The first step is to swap x and y if $|y| > |x|$. This guarantees that $|x| \geq |y|$ and that the expression $e_x - e_y$ will always be positive. The next step is to shift the mantissa of the smallest number with the exponent difference and add the shifted mantissa to the mantissa of the bigger number. Finally, a normalisation stage is required to normalise the mantissa (and increment the exponent) of the resulted operation. This is required if the resulted mantissa is smaller than 1. The normalisation stage can be processed using a leading-one detector to find the MSB bit of the mantissa. It is important to note that the hardware implementation of the shift stage can infer barrel shifters and compromise the performance of the floating-point adder [14]. The high-performance computation can be achieved using pipelined stages in addition to the substitution of the barrel shifts with a set of constant shifts executed in parallel.

Consider now the operation of floating-point multiplication defined in the equation 3.

$$x \times y = (-1)^{s_x \oplus s_y} \times 2^{e_x + e_y} \times (1.m_x \times 1.m_y) \tag{3}$$

The floating-point multiplication is straightforward. The sign of the product is the xor operation of the sign operands. The exponent and mantissa of the product consist of the sum and multiplication of the exponent and mantissae of the operands, respectively. Similar to the addition, the normalisation of the mantissa (and the descrease of the exponent) is also required if the resulted mantissa is bigger than 1. The basic architectural description of the floating-point adder and multiplier is also shown in figure 1.

2.3 Polynomial approximation based functions

The implementation of a floating-point divider is conceptually similar to the multiplication operation. According to equation 4, the computation of the quotient's sign also results from a xor operation. The exponent of the quotient is the difference of the operand exponents, and the mantissa is the product of the mantissa's dividend with the reciprocal of the mantissa's divisor[5]. In addition to that, an exception handling deals with division by zero, infinities and not-a-numbers. However, the computation of the reciprocal of the mantissa turns the divider a complex function.

$$\frac{x}{y} = (-1)^{s_x \oplus s_y} \times 2^{e_x - e_y} \times 1.m_x \times \frac{1}{1.m_y} \tag{4}$$

Although a change in the base only adds a constant multiplication, the logarithm in base 2 is the most convenient due to its simplicity. According to equation 5, if the operand is negative, the resulted logarithm is not-a-number, since the logarithm is defined only for positive reals. Otherwise, the result is the logarithm of the mantissa added to the shifted exponent of the operand [16]. This addition results in a fixed-point number and a floating-point to fixed-point conversion converts the final value[5]. Additionally, an exception handling can handle operations with infinities and not-a-numbers.

$$\log_2(x) = \begin{cases} e + log_2(1.m), & \text{if } s = 0 \\ NaN, & \text{if } s = 1 \end{cases} \tag{5}$$

The floating-point exponentiation can be simplified by selecting the base 2 for the power function[5]. The first step for the operation is to convert the floating-point to a fixed point number and break the result into two parts X_I and X_F [11]. As described in equation 6, the integer number X_I is the resulted exponent. The mantissa is computed with the expression 2^{X_F}, if the operand x is positive, or 2^{-X_F} is if $x < 0$. The exception handling is responsible to manipulate underflows, overflows, infinities and not-a-numbers.

$$2^x = 2^{(-1)^s \times (X_I + X_F)} = \begin{cases} 2^{X_I} \times 2^{X_F}, & \text{if } s = 0 \\ 2^{-X_I} \times \frac{1}{2^{X_F}}, & \text{if } s = 1 \end{cases} \tag{6}$$

The floating-point square-root operation is defined in equation 7. The idea is to compute the square-root of the mantissa's operand if its exponent is even or to multiply the mantissa's operand by two and take the square-root if the exponent is odd [8]. The resulted exponent is calculated checking the least significant bit of the exponent's operand and based on this bit, the exponent is decrement by one, followed by a division by two using a shift register[6]. The computation of the mantissa requires the implementation of the square-root operand to compute the number $\sqrt{t}$ in the range $t \in [1, 4)$.

$$\sqrt{x} = \begin{cases} 2^{\frac{e}{2}} \times \sqrt{1.m}, & \text{if } e \text{ is even} \\ 2^{\frac{e-1}{2}} \times \sqrt{2 \times 1.m}, & \text{if } e \text{ is odd} \\ NaN, & \text{if } s = 1 \end{cases} \tag{7}$$

The implementation of the operations in equations 1 to 4 requires the manipulation of transcendental functions[7], which can be computed using polynomial

[5] Again, the base change is realized with a constant multiplication.

[6] If the operand is negative, the resulted square-root is not-a-number.

[7] Even the division, not considered a transcendental function, still requires an approximation.

approximations within a specific range. The accuracy of the implementation is dependent on the degree of the polynomial used in the approximation. Slicing the range of the function into smaller segments leads to improved accuracy[17].

A function $f(x)$ can be approximated by a degree-d, n-segments polynomial y in the range $x_i \leq x < x_f$. The approximated function in the k-th segment is defined according to equation 8, where $\eta = \frac{(x_f - x_i)}{n}$ and the matrix $C_{n \times d}$ contains the coefficients of the polynomial approximation.

$$y(k, d, x) \approx \sum_{n=0}^{d} C(k, n) x^{d-n},\ x_i + k\eta \leq x < x_i + (k+1)\eta \tag{8}$$

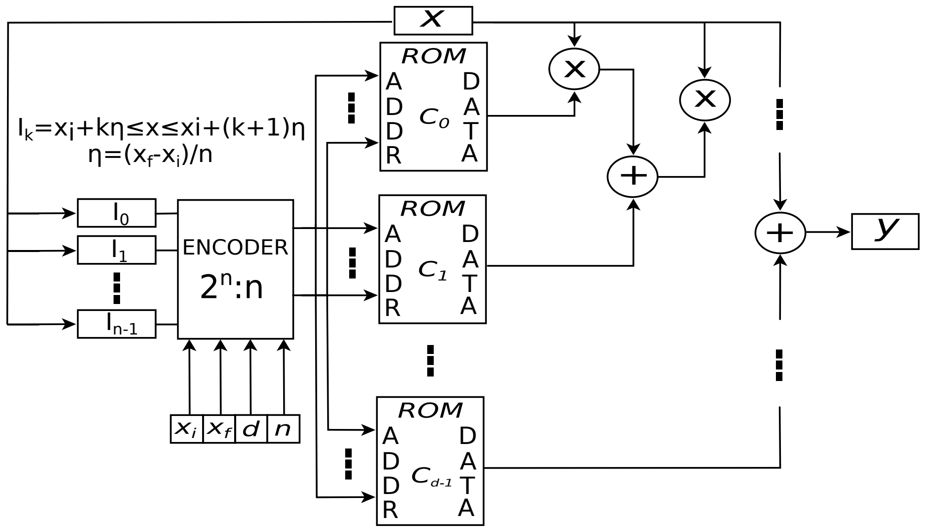


Fig. 2. Parameterizable degree-d, n-segments polynomial approximator

Figure 2 illustrates the hardware implementation of a general polynomial approximator. The architecture is parameterizable in terms of the degree d and the number of segments n. The computation of the approximation uses d additions and d multiplications. Moreover, $n \times (d + 1)$ coefficients are stored in lookup tables to evaluate the approximations.

$$C^{\alpha} = \begin{bmatrix} 0.70986 & -0.9735 & 0.99947 \\ 0.38742 & -0.82214 & 0.98109 \\ 0.23424 & -0.67285 & 0.94441 \\ 0.15228 & -0.55171 & 0.89948 \end{bmatrix}, I^{\alpha} \in [0, 1) \tag{9}$$

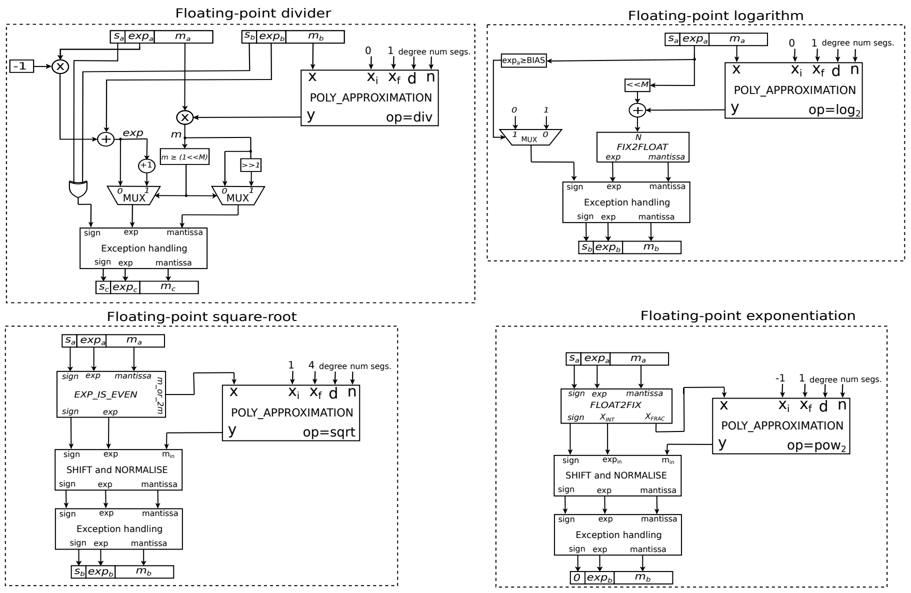


Fig. 3. Hardware implementation of floating-point division, logarithm, square-root and exponentiation using the general polynomial approximator

$$C^{\beta} = \begin{bmatrix} -0.573 & 1.42883 & 0.00028 \\ -0.3829 & 1.33815 & 0.0147 \\ -0.27387 & 1.2312 & 0.03792 \\ -0.20558 & 1.12988 & 0.07564 \end{bmatrix}, I^{\beta} \in [0,1) \tag{10}$$

$$C^{\gamma} = \begin{bmatrix} 0.14315 & 0.62811 & 0.98516 \\ 0.20244 & 0.68584 & 0.9997 \\ 0.28629 & 0.68363 & 1.0004 \\ 0.40488 & 0.56192 & 1.0326 \end{bmatrix}, I^{\gamma} \in [-1,1) \tag{11}$$

$$C^{\delta} = \begin{bmatrix} -0.07913 & 0.64644 & 0.43337 \\ -0.04069 & 0.51676 & 0.54339 \\ -0.02576 & 0.44338 & 0.63378 \\ -0.01816 & 0.39451 & 0.71252 \end{bmatrix}, I^{\delta} \in [1,4) \tag{12}$$

The hardware implementation of the floating-point operations of division, logarithm, square-root and exponentiation is shown in figure 3. Equations 9 to 12 show the coefficients and ranges $I \in [x_i, x_f)$ used in the approximations for the operations reciprocal ($\frac{1}{x}$), logarithm ($log_2(x)$), exponentiation (2^x) and square-root ($\sqrt{x}$), respectively. A degree-2, 4-segments polynomial is chosen in this particular case. Each function incorporates one instance of the polynomial approximator in its datapath, allowing the generation of customizable hardware with little modification. The latency of the operations is a function of the degree of the polynomials. This results from the pipelined additions and multiplications in the polynomial approximator to achieve high-performance computation. The number of segments does not affect the latency, once the ranges are selected using a combinatorial priority encoder[8]. Table 1 shows the latency of each floating-point operation described in this work.

Table 1. Latency and throughput of each floating-point operation

Operation	**Latency**
float2fix conversion	1 cycle
fix2float conversion	1 cycle
addition	6 cycles
multiplication	1 cycle
degree-d approx. for division	d+4 cycles
degree-d approx. for exponentiation	d+4 cycles
degree-d approx. for logarithm	d+3 cycles
degree-d approx. for square-root	d+3 cycles

3 Proposed Framework

This section describes the framework for high-performance video processing on FPGA. The proposed framework consists of FPGA implementations of pixel processing operations in floating-point arithmetic. Composite functions including addition, multiplication, division, logarithm and square-root process pixel at a frequency of 148.5 MHz. Figure 4 shows the block diagram of the framework. The framework contains an HDMI receiver (dvi2rgb) and transmitter (rgb2dvi), the clock wizard[9] (for providing all the required clock signals) and the video framework, which is the core of the floating-point library and the SPI interface to exchange data with the computer and ensure architectural reconfigurability.

The pixel stream is 24 bits wide, 8-bits for each RGB channel broadcasted at 60 fps for a resolution of $1080p$. A fix2float operation converts the incoming pixel to 16-bit floating-point, 10 bits for the mantissa and 5 bits for the exponent. Inside the framework, the floating-point pixel streams into a pipelined arithmetic datapath performing the operations from equations 13 to 16; R, G and B are the red, green and blue channels in floating-point format, c_0, c_1 and c_2 are floating-point parameters of the functions f_2, f_3 and f_4, respectively.

$$f_1(R,G,B) = \frac{max(R,1) \times max(G,1)}{max(R,1) + max(B,1)} \tag{13}$$

[8] Alternatively, the range of x is selected using the $\lceil log_2(n) \rceil$ MSB bits of x.

[9] The dvi2rgb and rgb2dvi are IP-Cores provided by Digilent and the clock wizard is included in the Xilinx Vivado Suite.

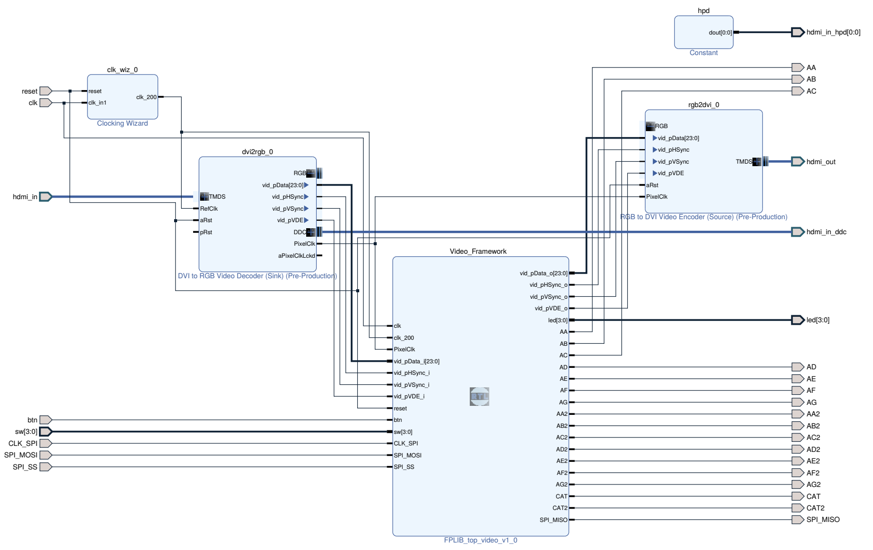


Fig. 4. Block diagram for video framework. The rgb2dvi and dvi2rgb are IP-cores for HDMI interface and the Video_framework is a custom library for floating-point pixel processing.

$$f_2(c_0, R) = c_0 \times log_2(max(R, 1)) \tag{14}$$

$$f_3(c_1, R) = 2^{c_1 \times R} \tag{15}$$

$$f_4(c_2, R) = \sqrt{c_2 \times R} \tag{16}$$

As illustrated in Figure 5, the variables c_0, c_1 and c_2 can be updated on-the-fly through the SPI interface. The values are sent to the FPGA using a user interface written in Python using the library PyQt5 [22]. Each variable is written serially as 2 bytes value using an FTDI interface [12]. The SPI interface is also responsible for selecting the arithmetic operation to process the video frames. There is a total of 20 operations; each one is a variant of equations 13 to 16. Each variant is an expression varying the degree of the polynomial or the number of segments in the approximation, summing 16 floating-point operations. In addition to that, four operations computed using arithmetic primitives generated with the Flopoco framework are also available. Table 2 shows the latency of each implementation of equations 13 to 16, where f_1 to f_4 are the implementations using the proposed floating-point library already discussed in section 2, whilst each function g_1 to g_4 are implementations of those same equations using the Flopoco framework [9].

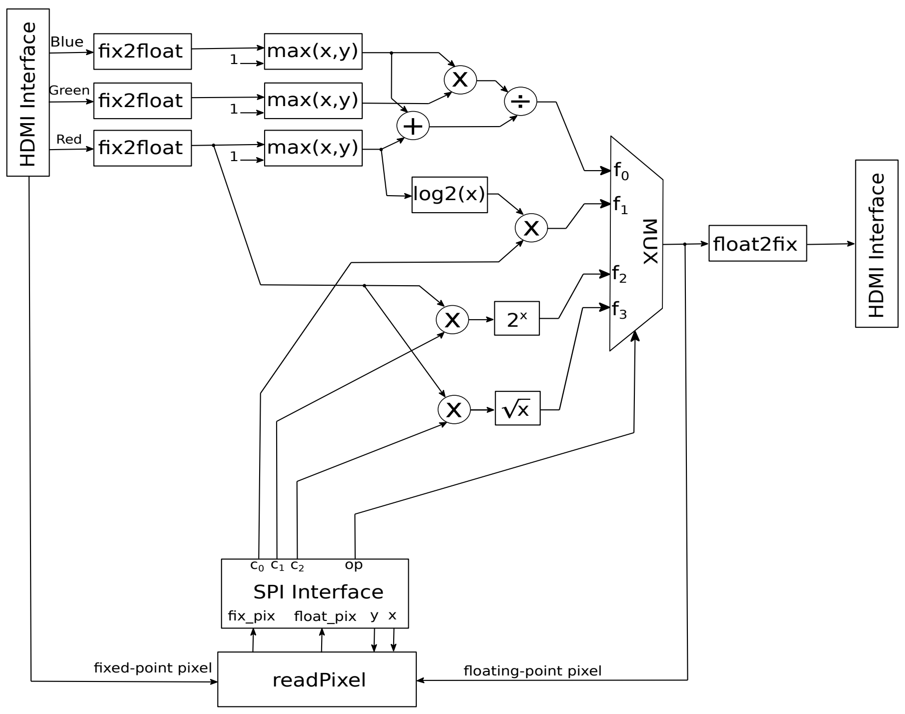


Fig. 5. Pipelined floating-point arithmetic is able to process $1080p$ pixel at a frame rate of 60 fps. In this particular case, the floating-point representation is of 16 bits, 10 and 5 bits for mantissa and exponent, respectively. The four functions f_1 to f_4 run in parallel and the coefficients c_0 to c_2 are set on-the-fly through the SPI Interface.

The user interface is also capable of reading the pixel from a grabbed frame given its spatial position. The user interface sends the x and y position of the pixel in the image as 4 bytes to the FPGA. A module responsible for counting the pixel position writes back the pixel value through the SPI interface. This feature enables the reading of floating-point pixels after image processing operations for numerical analysis in real-time.The SPI and video sync timings of this configuration are illustrated in Figure 6.

As can be seen in Figure 7, the user graphical interface (GUI) is responsible to write the registers c_0, c_1 and c_2, select the arithmetic operation to process the image and also write the pixel coordinates (row and column position) to read a processed pixel in floating-point format. A sample picture is captured using a frame grabber[10] and displayed using the software mplayer.

[10] The DVI2USB frame grabber from Epiphan was used in this experiment. The resolution was set to 1920×1080 with a frame rate of 60 frames per second.

Table 2. Latency and throughput of each floating-point composite function

Operation	Latency	Throughput
$f_1(R, G, B)$	d+10 cycles	1 operation per cycle
$f_2(c_0, R)$	d+5 cycles	1 operation per cycle
$f_3(c_1, R)$	d+5 cycles	1 operation per cycle
$f_4(c_2, R)$	d+4 cycles	1 operation per cycle
$g_1(R, G, B)$	15 cycles	1 operation per cycle
$g_2(c_0, R)$	14 cycles	1 operation per cycle
$g_3(c_1, R)$	10 cycles	1 operation per cycle
$g_4(c_2, R)$	15 cycles	1 operation per cycle

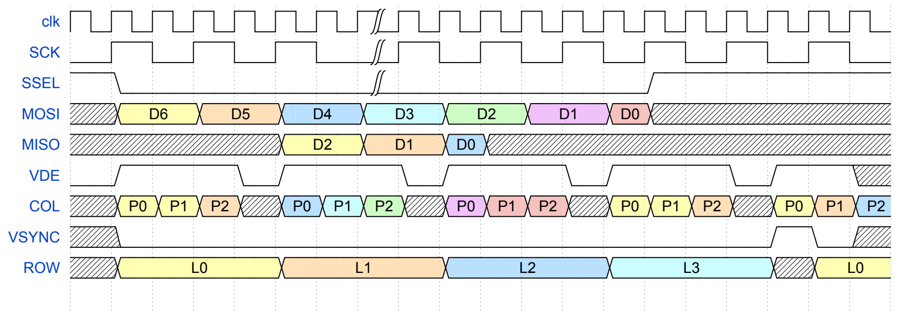


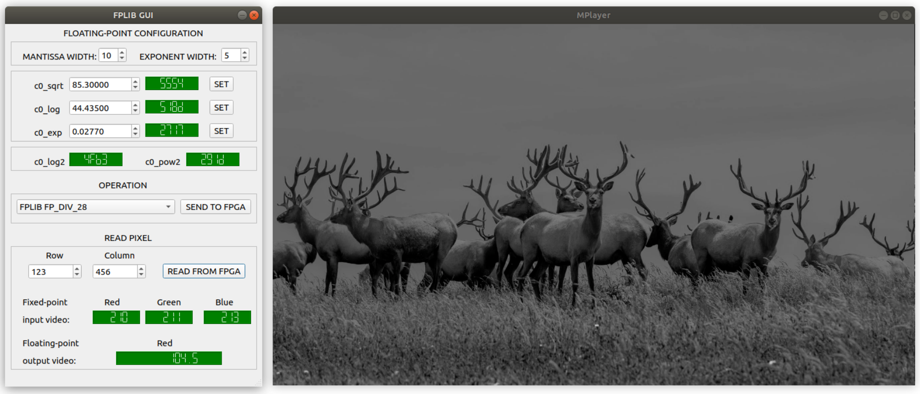
Fig. 6. SPI and Video timings for enabling architectural reconfigurability. Function parameters are sent to the FPGA and pixel values are read from the FPGA using a user interface.

4 Results and discussions

Table 3 shows the synthesis results for functions 13 to 16 using the proposed floating-point library[11].

Each function is composed with one of the following primitives: division, logarithm, square-root and exponentiation. For each of those primitives, the function is approximated with a degree-2 or degree-3 polynomial with 4 or 8 segments in the range where the function is approximated. As can be seen in the table, f_1, f_2, f_3 and f_4 are based on polynomial approximations and an increase in the degree of the polynomial results in a larger number of DSP blocks, once more multiplications are required to compute the approximation. On the other hand, an increase in the number of segments leads to a increased resource usage of LUT and LUTRAM. This is explained with the fact that more coefficients are needed to store the coefficients in the polynomial approximation. Specifically in the case implementation of f_1, the number of DSP blocks increased from 5 to 7 when the polynomial degree varies from 2 to 3, respectively, while the LUTRAM

[11] R, G, B, c_0, c_1 and c_2 are floating-point numbers using the *float16* representaiton. For this experiment, the mantissa and exponent widths were set to 10 and 5 bits, respectively.



MANTISSA WIDTH: 10 EXPONENT WIDTH: 5

c0_sqrt 85.30000 5554
c0_log 44.43500 518d
c0_exp 0.02770 2717
c0_log2 4F63 c0_pow2 291d

FPLIB FP_DIV_28

Row 123 Column 456

Fixed-point input video: Red 210 Green 211 Blue 213

Floating-point output video: Red 104.5

Fig. 7. The User graphical interface to control the communication between PC and FPGA.

showed an increased from 38 to 39 elements. The implementation of g_1, using the Flopoco framework requires 849 LUTs and 543 FFs, which is around double of the resource usage required by f_1, however, no DSP blocks are consumed in this implementation. Similar trends are observed in the implementation of f_4 and g_4. The implementation of f_2 showed a 50% increase in the number of DSP when the polynomial degree rises from 2 to 3, as opposed to g_2, which consumes only 1 DSP block. In addition to an increased number of FF and LUTs, g_2 also inferred 0.5 BRAMs. The implementation of g_3 also consumed 0.5 BRAMs, but no DSP block was inferred, whilst f_3 had an increase of 2 DSPs when the polynomial approximation goes from degree-2 to degree-3. All the experiments were realized using the Vivado 2019.1 targetting the Zybo Z7-20 FPGA board.

4.1 Hardware Acceleration of floating-point software implementations

As already discussed previously, due to the inherent exploration of parallelism on FPGAs, computer-intensive applications such as image and video processing are great candidates for hardware acceleration. The time to process a single frame of resolution 1920×1080 using Python implementation of each function f_1 to f_4 is shown in Table 4.

The hardware implementation process the pixel at a frequency of 148.5 MHz, which gives 6.73 nanoseconds to process each pixel, as each function has a throughput of one clock cycle according to Table 2. The framerate performance of f_1, f_2 and f_3 is nearly half of the 60 fps achieved by the hardware acceleration, and f_4 achieves a framerate of 92 fps, which is 54% higher than the specification. The justification of the acceleration is evident when a set of functions are combined. While the chain of 20 composite operations runs in parallel on the FPGA and still achieve the throughput of one operation per clock cycle, the software processes them in sequence. The software execution of four functions combined

Table 3. Syntehsis results for the hardware implementations of equations 13 to 16. f_1 to f_4 are the FPGA implementations of the equations with a degree-d n-segments polynomial approximations. g_1 to g_4 are the hardware implementations of the same floating-point expressions using the Flopoco framework.

Block name	**LUT**	**FF**	**BRAMs**	**DSP**	**LUTRAM**
$f_1(R,G,B)$, $d=2$, $n=4$	446	216	0.0	5	38
$f_1(R,G,B)$, $d=2$, $n=8$	453	218	0.0	5	39
$f_1(R,G,B)$, $d=3$, $n=4$	463	233	0.0	7	38
$f_1(R,G,B)$, $d=3$, $n=8$	473	231	0.0	7	39
$g_1(R,G,B)$	849	543	0.0	0	32
$f_2(c_0,R)$, $d=2$, $n=4$	183	100	0.0	4	13
$f_2(c_0,R)$, $d=2$, $n=8$	189	100	0.0	4	13
$f_2(c_0,R)$, $d=3$, $n=4$	202	109	0.0	6	13
$f_2(c_0,R)$, $d=3$, $n=8$	208	109	0.0	6	13
$g_2(c_0,R)$	573	498	0.5	1	24
$f_3(c_1,R)$, $d=2$, $n=4$	253	98	0.0	4	22
$f_3(c_1,R)$, $d=2$, $n=8$	305	100	0.0	4	22
$f_3(c_1,R)$, $d=3$, $n=4$	274	113	0.0	6	22
$f_3(c_1,R)$, $d=3$, $n=8$	320	116	0.0	6	22
$g_3(c_1,R)$	386	227	0.5	0	21
$f_4(c_2,R)$, $d=2$, $n=4$	101	50	0.0	4	9
$f_4(c_2,R)$, $d=2$, $n=8$	147	52	0.0	4	9
$f_4(c_2,R)$, $d=3$, $n=4$	120	64	0.0	6	9
$f_4(c_2,R)$, $d=3$, $n=8$	162	67	0.0	6	9
$g_4(c_2,R)$	236	264	0.0	0	13

takes around 0.09 seconds, which gives a framerate of about one-sixth of the 60 fps. On the other hand, if a set of 20 operations are processed using the software model, it accomplishes just around 3.5% of the desired framerate.

5 Conclusions

This paper presented FPGA implementation of a custom floating-point library. The library is a set of functions including addition, multiplication, division, square-root, exponentiation, logarithm, floating-point to fixed-point and fixed-point to floating-point conversion. These operations run at a clock speed of 148.5 MHz. The implementation of the division, logarithm, exponentation and logarithm architecture achieved a compact architecture with a degree-2 4-segments polynomial approximation, saving up to 50% in DSP blocks and 60% in LUTs when compared to the degree-3, 8 segments polynomial approximations. In our future work, we will use the proposed library to process real-time video with custom floating-point pixel arithmetic and we will evaluate the image quality of this library against software implementaitons of image processing algorithms. Compared to the software implementation, the hardware acceleration doubled the framerate of the functions f_1, f_2 and f_3, whereas f_4 is fast enough to run

Table 4. Time to process each function using a Python implementation

Operation	time (seconds)	frames per second
$f_1(R, G, B)$	0.0287	34.81
$f_2(c_0, R)$	0.0285	35.06
$f_3(c_1, R)$	0.0296	33.83
$f_4(c_2, R)$	0.0108	92.53
all functions	0.0966	10.35
all functions $\times 5$	0.4671	2.14

at 90 fps on a Core-i7 computer configuration running at 2.6GHz. However, the speedup of the hardware acceleration is evident when a chain of 20 functions is combined, where the software implementation achieved roughly 2 fps, which is one-thirty of the framerate specification.

References

1. Nikolaos Alachiotis and Alexandros Stamatakis. Efficient floating-point logarithm unit for fpgas. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.
2. Altera. Floating-Point IP Cores User Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altfp_mfug.pdf. [Online].
3. Donald G Bailey. Image processing using fpgas, 2019.
4. Marcel Bosc, Fabrice Heitz, Jean-Paul Armspach, Izzie Namer, Daniel Gounot, and Lucien Rumbach. Automatic change detection in multimodal serial mri: application to multiple sclerosis lesion evolution. *NeuroImage*, 20(2):643–656, 2003.
5. Nelson Campos, Slava Chesnokov, Eran Edirisinghe, and Alexis Lluis. FPGA implementation of custom floating-point logarithm and division. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 295–304. Springer International Publishing, 2021.
6. Tsan-Jieh Chen, Herming Chiueh, Chih-Cheng Hsieh, Chin Yin, Wen-Hsu Chang, Hann-Huei Tsai, and Chin-Fong Chiu. High definition image pre-processing system for multi-stripe satellites' image sensors. *IEEE Sensors Journal*, 12(9):2859–2865, 2012.
7. Robert T. Collins, Alan J Lipton, and Takeo Kanade. Introduction to the special section on video surveillance. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):745–746, 2000.
8. Florent De Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. Multiplicative square root algorithms for fpgas. In *2010 International Conference on Field Programmable Logic and Applications*, pages 574–577. IEEE, 2010.
9. Florent De Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 28(4):18–27, 2011.
10. Bart De Ruijsscher, Georgi N Gaydadjiev, Jeroen Lichtenauer, and Emile Hendriks. Fpga accelerator for real-time skin segmentation. In *Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 93–97. IEEE Computer Society, 2006.

11. Jérémie Detrey and Florent de Dinechin. A parameterized floating-point exponential function for fpgas. In *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.*, pages 27–34. IEEE, 2005.
12. Interfacing FT2232H Hi-Speed Devices. Application note an_114 interfacing ft2232h hi-speed devices to spi bus. 2012.
13. Vijay K Jain and Lei Lin. Square-root, reciprocal, sine/cosine, arctangent cell for signal and image processing. In *Proceedings of ICASSP'94. IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 2, pages II–521. IEEE, 1994.
14. Per Karlstrom, Andreas Ehliar, and Dake Liu. High performance, low latency fpga based floating point adder and multiplier units in a virtex 4. In *2006 NORCHIP*, pages 31–34. IEEE, 2006.
15. Hagai Kirshner and Moshe Porat. On the role of exponential splines in image interpolation. *IEEE Transactions on Image Processing*, 18(10):2198–2208, 2009.
16. Martin Langhammer and Bogdan Pasca. Single precision logarithm and exponential architectures for hard floating-point enabled fpgas. *IEEE Transactions on Computers*, 66(12):2031–2043, 2017.
17. Dong-U Lee, Ray Cheung, Wayne Luk, and John Villasenor. Hardware implementation trade-offs of polynomial approximations and interpolations. *IEEE Transactions on computers*, 57(5):686–701, 2008.
18. MathWorks. You don't always need to convert to fixed point for fpga or asic deployment. uk.mathworks.com/company/newsletters/articles/you-dont-always-need-to-convert-to-fixed-point-for-fpga-or-asic-deployment.html. [Online; accessed 23-December-2020].
19. Daniele Jahier Pagliari, Enrico Macii, and Massimo Poncino. Zero-transition serial encoding for image sensors. *IEEE Sensors Journal*, 17(8):2563–2571, 2017.
20. Richard J Radke, Srinivas Andra, Omar Al-Kofahi, and Badrinath Roysam. Image change detection algorithms: a systematic survey. *IEEE transactions on image processing*, 14(3):294–307, 2005.
21. Balázs Renczes and István Kollár. Roundoff errors in the evaluation of the cost function in sine wave based adc testing. In *20th IMEKO TC4 International Symposium and 18th International Workshop on ADC Modelling and Testing*, pages 15–17, 2014.
22. Vivian Siahaan and Rismon Hasiholan Sianipar. *LEARNING PyQt5: A Step by Step Tutorial to Develop MySQL-Based Applications*. SPARTA PUBLISHING, 2019.
23. Synopsys, Inc. DesignWare Developers Guide. http://venividiwiki.ee.virginia.edu/mediawiki/images/7/78/Dw_developers_guide.pdf. [Online].
24. Christopher Richard Wren, Ali Azarbayejani, Trevor Darrell, and Alex Paul Pentland. Pfinder: Real-time tracking of the human body. *IEEE Transactions on pattern analysis and machine intelligence*, 19(7):780–785, 1997.
25. Xilinx. LogiCORE IP Floating-Point Operator v7.0. https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_0/pg060-floating-point.pdf. [Online].
26. Chi Wai Yu, Julien Lamoureux, Steven JE Wilton, Philip HW Leong, and Wayne Luk. The coarse-grained/fine-grained logic interface in fpgas with embedded floating-point arithmetic units. In *2008 4th Southern Conference on Programmable Logic*, pages 63–68. IEEE, 2008.