Floating-Point Support for Embedded FPGA Platform with 6502 Soft-Processor

Sait Izmit
Delft Center for Systems and Control
Delft University of Technology
sizmit@control-lab.dcsc.tudelft.nl

Abstract— **This paper presents a framework for the porting of the SoftFloat floating-point implementation package to an FPGA with 6502 soft core and evaluates the performance of the package on this platform by testing a 2^{nd} order Butterworth Low Pass Filter. Within the framework, 8051, PIC and FPGA architectures are explained and the three platforms are compared for control applications in order to explain why FPGA with soft core is chosen as a platform. The IEEE 754 - *Standard for Binary Floating-Point Arithmetic* is explained for Single Precision Floating-Point Format in order to give the reader an insight in the representation of floating-point numbers, special values, floating-point arithmetic, rounding modes and exceptions. The porting details and execution times for available floating-point functions are presented. To evaluate the performance of the package, an experiment is carried out using a 2^{nd} order Butterworth Low Pass Filter. The design of the Butterworth Low Pass Filter and its implementation in microcontrollers using direct form realization is presented. Fixed-point arithmetic is introduced and a methodology is presented for the choice of the number of integer and fraction bits. The effects of truncation and filter coefficient perturbation on the filter properties are discussed in detail.**

I. Introduction

8051s, PICs and FPGAs are widely used embedded platforms for control applications. While 8051s and PICs introduce fast and cheap control solutions, they are limited by supplier-defined functionalities such as the integer size and the number of interrupts. On the other hand, FPGAs with soft-processors introduce more flexible microcontroller architectures. It is possible to add user-defined functionalities to FPGAs with soft-processors depending on the specifications of the application. Due to its flexible architecture, FPGA with 8 bit core 6502 soft-processor is chosen as platform to evaluate floating-point support.

Floating-point computations are more convenient in scientific and engineering applications. Even though floating-point arithmetic is slower compared to integer arithmetic, floating-points can represent a greater range compared to integers, which help dealing with overflow problems. With their changing precision throughout their range and the rounding modes that are used in the arithmetic operations, sometimes floating-points can introduce higher computational precision . That is why floating-points are studied on FPGA with 6502 soft-processor in this paper.

This paper is written for the *in4073 - Real-Time Embedded Systems* course in Delft University of Technology and the porting of SoftFloat floating-point package to FPGA with 6502 soft-processor is discussed in detail. A case study is also carried out both to test the performance of the floating-point package and also to show the implementation of digital filters in microcontrollers using fixed-point arithmetic.

This paper is organized as follows, Section 2 compares 8051s, PICs and FPGAs for control applications. Section 3 explains IEEE 754 - *Standard for Binary Floating-Point Arithmetic* for Single Precision Floating-Point Format. Section 4 explains the structure of the SoftFloat Floating-Point Implementation Package, porting details for FPGA with 6502 soft-processor and available floating-point functions. In Section 5, a case study is explained in detail for the comparison of integer and floating-point calculations on the FPGA platform using a 2^{nd} order Butterworth Low Pass (BLP) Filter. Conclusions are drawn in Section 6.

II. 8051, PIC and FPGA Platforms

In this section, the advantages and disadvantages of three different 8 bit microcontrollers for control applications are discussed.

A. 8051 Microcontrollers

The 8051 family consists of 8 bit Harvard architecture microcontrollers, which were first developed by Intel in 1980 [4]. They became very popular in the 1980s and the early 1990s for control applications.

One of the most important advantages of the 8051 is its Boolean processor [4]. This means that the instruction set of the microcontroller is specialized in dealing with binary inputs and outputs. This makes the 8051s suitable for control applications.

The second advantage of the 8051 microcontroller is its speed. According to [4], most available versions use oscillator frequencies varying between 3.5 and 33 MHz. However, each machine cycle is also divided into 12 oscillator periods. 8051s are *Complex Instruction Set Computers* (CISC), which execute microcodes as their instructions. Microcodes can be regarded as small programs, which perform series of actions. Therefore, by dividing the machine clock into oscillator periods, the 8051s can execute instructions in one or more machine cycles. [4] also states that the 8051 has four separate register sets. When an interrupt occurs, the microcontroller does not need to store the register contents in the stack. Only the program counter is stored. This property greatly reduces the interrupt latency, resulting in a faster performance.

The 8051 microcontrollers are available for different prices with different features. The designer can choose among different 8051 versions for the necessary features depending on the complexity of the application. Today, enhanced 16 bit 8051 versions with A/D converters, PWM and I2C capabilities are available in the market.

As most of the 8 bit microcontrollers, 8051s are poor in implementing complex designs that require high sampling frequencies and complex computations. 8051s may fail performing the required tasks when sampling frequency increases or the software implementation of the task becomes too complex.

The 8 bit integer size of 8051s can introduce limitations on the system. First of all, overflows and underflows can easily occur in the computations. Second, in many applications, the precision introduced by 8 bit integers is not enough for good computational performance. The 8051 microcontrollers do have the floating-point support but it is not implemented in the hardware. The floating-point functions are available as software macros, which result in poor performance. This is a common problem with most of the 8 bit microcontrollers. Therefore, it is not feasible to use 8051s for applications that require a lot of floating-point computations.

As a conclusion, the 8051 microcontrollers were very popular in 1980s for control applications due to their optimized instruction sets in one-bit operations, their fast performances and their low-costs. However, the 8051s are limited in handling complex designs and high precision computations, which does not make them very suitable for every control application.

B. PIC Microcontrollers

Compared to the 8051 microcontrollers, which contain Boolean processors, PICs are 8-16 bit Harvard Architecture RISC microcontrollers [5]. RISC stands for Reduced Instruction Set Computer. According to [6], with RISC approach, less number of instructions with simpler formats and fixed lengths are available that permit fast hardwired decoding. This simplifies the instruction pipelining in the microcontroller, reducing the execution times of all the instructions in average to one clock cycle. Compared to the 8051s, which usually have around 111 complex instructions [4], the PICs support only 35 simple RISC instructions [5]. The disadvantage of the RISC approach is that less number of simpler instructions result in longer programs [6]. However, since most of the instructions are register-based, access to main memory is also reduced, increasing the speed of the microcontroller [6].

Just like the 8051 microcontrollers, it is possible to find wide variety of PICs for different prices with different features on the market. However, more advanced features are available for the PIC microcontrollers compared to 8051s. Today, some models include LCD drivers, peripherals for I2C, SPI, IS2 and motor control, and support for USB, Ethernet, CAN, LIN and IRDA. These futures make PIC microcontrollers highly suitable for real-time control applications. However, just like the 8051s, PICs are limited in implementing complex designs. Their 8 bit integer size introduces the same problem with 8051s in high precision computations. It is possible to find software floating-point macros for PIC microcontrollers, which are poor in performance.

As a conclusion, just like 8051s, PICs are general purpose microcontrollers with a different architecture. They introduce fast and low-cost real-time control problem solutions. However, they are not suitable for every control application just like the 8051 microcontrollers due to their drawbacks in handling complex designs and high precision computations.

C. Field Programmable Gate Arrays

A Field Programmable Gate Array, FPGA, has its roots from the Complex Programmable Logic Device, CPLD, of the mid 1980s. According to [8], an FPGA is a semiconductor device composed of logic components connected with programmable interconnections, which allow the users to implement multi-level logic.

FPGAs introduce a very different approach in designing systems compared to general purpose integrated circuits like 8051s and PICs. As [7] points out, standard part solutions like 8051s and PICs, deliver supplier-defined functionality at low cost. On the other hand, application specific integrated circuits (ASICs) deliver user-defined functionality at high cost. FPGAs combine the advantages of the both approaches and deliver user-defined functionality at low cost.

According to [8], the programming circuitry of FPGA slows down the connection paths, resulting in a slower system performance compared to ASICs. Even though FPGAs are cheaper then ASICs, they are still expensive compared to solutions like 8051s and PICs.

It is possible to use FPGAs for control applications in two ways. The first one is to implement the control algorithm using a Hardware Description Language. This approach results in even faster performance compared to 8051 and PIC microcontrollers.

The second approach, which is the focus of this paper, is using a soft-processor implemented by a Hardware Description Language to transform the FPGA platform into a microcontroller. Then the designer can use this microcontroller to implement control algorithms. This approach has both advantages and disadvantages. An FPGA with soft-processor has slower performance, consumes more energy and introduces higher-cost solutions compared to 8051 and PIC microcontrollers.

Despite its disadvantages, FPGA with soft-processor introduces a more flexible microcontroller architecture. The functionalities of the microcontroller are not supplier defined as it is in the 8051s and PICs, instead they are design parameters to be defined by the user. Depending on the features of the soft-processor more complex designs can be handled by the FPGA compared to 8051 and PIC microcontrollers. Another important advantage is that computations with greater range and higher precision can be executed since integer size is a design parameter for the soft-processor. Just like 8051s and PICs, floating-point support in software can also be introduced to the system, as it is explained in Section 4. However, different than most of the 8 bit microcontrollers, the floating-point support can also be implemented in VHDL as a part of the soft-processor. Such an implementation can result in better floating-point performance compared to 8051s and PICs.

As a conclusion, 8051 and PIC microcontrollers are only suitable for simpler control applications due to their limitations on handling complex designs and executing high precision computations. An FPGA with soft-processor introduces a more flexible microcontroller architecture and enables more complex applications that require higher computational precision. However, FPGA platform introduces a slower performance and a more expensive solution compared to 8051s and PICs.

III. Floating-Point Numbers

In this section, the Single Precision Floating-Point Format is explained according to the IEEE 754 - *Standard for Binary Floating-Point Arithmetic*.

A. Floating-Point Representation

According to [2], floating-Points are divided into two: the basic format and the extended format. The basic format consists of the single precision format, which is represented with 32 bits and the double precision format, which is represented with 64 bits. Only the single precision floating-point format is discussed in this paper.

The basic representation of numbers in computer based systems is the integer representation. When a 32 bit 2's complement integer is concerned, the MSB is used for the representation of the sign and the remaining 31 bits are used for the representation of the magnitude. Therefore the integer has a range from $-2,147,483,648$ to $+2,147,483,647$. Furthermore, the integer representation has a fixed precision through out its range.

In some applications, 32 bit integers may result in overflows and underflows. Sometimes the precision is not enough to give good computational results. For example, in some digital filtering applications, if the filter coefficients can not be represented with enough precision, the filter may become unstable. Floating-points introduce a greater range with the same number of bits used to represent the numbers. Further, the precision of the floating-point numbers change throughout its range. Therefore, depending on the application, floating-points are used to overcome the range problem of the integers and sometimes they introduce higher precision.

As [5] states, basically a floating-point number is represented according to,

$$n = b^e \times m \tag{1}$$

where e is the exponent, m is the mantissa or the significant and b is the base number or the radix. According to [9], on the computer level, the single precision floating-point number consists of a 32 bits representation. Bits 0 - 22 are used to represent the mantissa or the significant, bits 23 - 30 are used to represent the exponent and the bit 31 is used to represent the sign of the floating-point number.

The floating-point representation increases the range of the number dramatically compared to the integer representation. Further, the precision is not fixed throughout the range anymore. Instead, the floating-point numbers are very close to each other around zero and the precision decreases as the numbers get further away from the origin.

There are two types of floating point representations: the normalized floating-points and the denormalized floating-points. According to [9], they are represented using the sign s, which is equal to zero for positive numbers and equal to one for negative numbers, the fraction f, which is equal to

$$f = (b_{22}^{-1} + b_{21}^{-2} + ... + b_{0}^{-23}), \tag{2}$$

and the unbiased exponent e, which is equal to

$$e = E - 127, \tag{3}$$

where E is the biased exponent. While $E_{max} = 255$ is reserved to represent special values $\pm\infty$ and NaN, $E_{min} = 0$ is reserved to represent ± 0 and denormalized floating-point numbers. Further, the unbiased exponent e is defined in order to make negative exponents possible without using a sign bit for the exponent. [9] represents the normalized floating-points and the denormalized floating-points according to,

Normalized:

$$(-1)^s 2^e \times 1.f \tag{4}$$

Denormalized:

$$(-1)^s 2^{-126} \times 0.f \tag{5}$$

Denormalized floating-points are used to represent numbers very close to zero when gradual underflow occurs. Denormalized floating-points and gradual underflow are discussed in Section III-E.

B. Range, Overflow and Underflow

It is very easy to compute the range for different representations using Equations 4 and 5 from the previous section. The results can be observed in Table I.

TABLE I
RANGES OF DIFFERENT REPRESENTATIONS

Representation	Range
Normalized FP	$\pm 2^{-126}$ to $\pm(2-2^{-23})2^{127}$
Denormalized FP	$\pm 2^{-149}$ to $\pm(2-2^{-23})2^{-126}$
Integer	-2^{31} to $2^{31}-1$

Combining the normalized and the denormalized floating-point numbers one can come up with the specifications in Table II for *IEEE 754 Single Precision Floating-Point Numbers*.

TABLE II
FLOATING POINT SPECIFICATIONS

Effective Range	$\pm(2-2^{-23})2^{127}$
Negative Overflow	$n < -(2-2^{-23})2^{127}$
Negative Underflow	$n > -2^{-149}$
Positive Underflow	$n < -2^{-149}$
Positive Overflow	$n > (2-2^{-23})2^{127}$

C. Rounding Modes

There are four rounding modes that are specified in IEEE 754. Some guarding bits are used in arithmetic operations in order to make rounding possible.

The first method is *Round to Nearest Even*. It is also the default rounding method that is used in IEEE 754. As [2] explains, with this method the number is rounded to the nearest value. If the number is exactly in the middle then it is rounded to the nearest even value. For example, the number 3.4 will be rounded to 3 and the number 4.8 will be rounded to 5. However number 3.5 will be rounded to 4 and 2.5 will be rounded to 2.

The second method is *Round to Zero*. Actually no rounding is made with this method. The excess bits are truncated to produce the result [2]. For example the numbers 3.1, 3.5 and 3.8 are all rounded to 3.

The third method is *Round-Up*. With this method the number is rounded to $+\infty$ [2]. For example, 3.1 becomes 4 and -3.1 becomes -3.

The last method is *Round-Down*. It is the opposite of the previous method. With this method the number is rounded to $-\infty$ [2]. For example, 3.1 becomes 3 and -3.1 becomes -4.

D. Exceptions

According to [2], there are five exceptions that are specified in IEEE 754. The first one is the *Invalid Operation Exception*. When an invalid operation such as zero divided by zero or square root of a negative number is executed then the result is not-a-number, NaN. When the result of an operation is NaN, the *Invalid Operation Exception* is generated.

The infinity result can be generated in two ways. It can be either generated as a result of one of the arithmetic operations in which the result overflows or it can be generated when a nonzero number is divided by zero [2]. To differentiate the two cases, *Numerical Overflow Exception* and *Divide-by-Zero Exception* are defined.

Due to the restrictions on the exponent and the precision, the results sometimes may become inexact. In such cases, the *Inexact Result Exception* is generated. The *Numerical Underflow Exception* is also defined in IEEE 754 in order to be generated when underflow occurs.

E. Gradual Underflow and Denormalized Floating-Point Numbers

Gradual underflow occurs when the number becomes very small such that it can not be represented by the exponent anymore. In such cases, denormalized floating-point numbers are used to represent the gradually underflowed numbers [2]. The minimum value for the exponent is -126 but with denormalized floating-point numbers, it is possible to represent numbers with exponents up to -149. This is done by introducing leading zeros to the significant until the minimum exponent, -126, is reached [2]. An example is given in Table III.

The introduction of leading zeros result in loss of precision. In the worst case, all the bits are shifted by leading zeros, resulting in zero result. Since there are both positive and negative zeros that are defined in floating-point representation, the sign of zero shows the side that underflow occurs.

TABLE III
GRADUAL UNDERFLOW EXAMPLE

Sign	Exponent	Significant
0	-128	1.011000...0
0	-127	0.101100...0
0	-126	0.010110...0

IV. SOFTFLOAT PACKAGE

SoftFloat Floating-Point implementation Package is ported to FPGA with 6502 soft-processor, which has an 8 bit core, for floating-point support for the system. SoftFloat package includes two different source codes in it: one for floating-point implementation using 32 bits integers and one for floating-point implementation using 64 bits integers. Since in 6502 soft-processor only 32 bits long integers are supported, the parts of the source code for 64 bits integers are taken out. The top most two directories of the source code are softfloat and processors. The softfloat directory contains the most of the source code while the processors directory includes target specific header files that are not specific to SoftFloat. For the ported version of the code for 6502, the two directories are merged and all the files are present in one directory.

There are six files included in the SoftFloat source code:

- *6502.h* - It is a target specific file that defines different integer sizes and also some target specific C preprocessor macros.
- *softfloat.h* - It is a target specific file that defines the membership functions of the class that are accessible by the programmer.
- *milieu.h* - It a target specific file that includes the necessary declarations to compile SoftFloat.
- *softfloat-specialize* - It is a target specific file that includes exception handlings and some internal routines for the source code.
- *softfloat-macros* - It is a target independent file that includes some arithmetic functions that are used in some internal routines.
- *softfloat.c* - It is a target independent file that includes the body of the SoftFloat source code.

Some membership functions in the source code are taken out while porting the code to 6502 in order to form a compact version of the SoftFloat floating-point implementation with just the necessary functions. The porting details and the necessary bug fixes can be found in Appendices A and B. The ported version of the code includes two conversion functions, five arithmetic operation functions, one remainder function and six comparison functions. The complete list of the functions is as follows:

Conversion Functions:

- *int32_to_float32* - Integer to floating-point conversion.
- *float32_to_int32* - Floating-Point to integer conversion.

Arithmetic Functions:

- *float32_add* - Addition operation.
- *float32_sub* - Subtraction operation.
- *float32_mul* - Multiplication operation.
- *float32_div* - Division operation.
- *float32_sqrt* - Square root operation.

Remainder Function:

- *float32_rem* - Remainder operation.

Comparison Functions:

- *float32_eq* - Equal to comparison.
- *float32_le* - Less than or equal to comparison.
- *float32_lt* - Less than comparison.
- *float32_eq_signaling* - Equal to comparison in which invalid exception is raised for any NaN input.
- *float32_le_quie* -]Less than or equal to comparison in which invalid exception is not raised for quiet NaN input.
- *float32_lt_quiet* - Less than comparison in which invalid exception is not raised for quiet NaN input.

TABLE IV
ARITHMETIC OPERATION EXECUTION TIMES

	Integer (msec)	FP (msec)	t_int/t_FP
$+$	0.02	$0.35 - 0.44$	$1/20$
$-$	0.02	$0.40 - 0.48$	$1/22$
$\times$	0.07	0.68	$1/9.7$
$\div$	0.1	1	$1/10$

Timing tests are also performed to observe the performance of the SoftFloat package. The execution times for the four arithmetic operations and their comparisons with the corresponding integer operations are presented in Table IV.

TABLE V
EXECUTION TIMES FOR FLOATING-POINT FUNCTIONS

Function	Execution Time (msec)
Integer to FP	0.22
FP to Integer	0.12-0.16
Remainder	0.40-0.48
Square Root	1.18-1.24
EQ	0.10
LE	0.14
LT	0.15
EQ_Signaling	0.09
LE_Quiet	0.15
LT_Quiet	0.15

As expected, the floating-point arithmetic operations are slower compared to the integer arithmetic operations. The execution times for the remaining floating-point functions are presented in Table V.

V. CASE STUDY

To study the performance of the floating-point package, a Butterworth Low Pass (BLP) Filter is implemented and tested with square input signal both in the PC environment and in the FPGA environment with 6502 soft-processor. The design process and the results are presented in the following sections.

A. *Butterworth Low Pass Filter Design*

The design of a 2^{nd} order Butterworth Low Pass (BLP) Filter with 10 Hz cut-off frequency and 5 ms sampling time is presented in this section. The reason for the choice of BLP filter is due to its sensitivity for arithmetic errors. For f_s as the sampling frequency, f_c as the cut-off frequency and N as the order of the filter, by defining,

$$\begin{aligned} &f_r = \tfrac{f_s}{f_c} \\ &\Omega_c = \tan\tfrac{\pi}{f_r} \\ &k = \tfrac{N}{2} - 1 \\ &c = 1 + 2\cos(\tfrac{(2k+1)\pi}{2N})\Omega_c + \Omega_c^2 \end{aligned} \tag{6}$$

the filter coefficients a_0, a_1, a_2, b_1 and b_2 can be calculated according to,

$$\begin{aligned} a_0 &= a\frac{\Omega_c^2}{c} \\ a_1 &= 2a_0 \\ a_2 &= a_0 \\ b_1 &= \frac{2(\Omega_c^2-1)}{c} \\ b_2 &= \frac{1-2\cos(\frac{(2k+1)\pi}{2N})\Omega_c+\Omega_c^2}{c} \end{aligned} \tag{7}$$

such that they satisfy,

$$a_0 + a_1 + a_2 - b_1 - b_2 = 1 \tag{8}$$

For the filter coefficients a_0, a_1, a_2, b_1 and b_2 that are calculated using equations in 7, the transfer function of a 2^{nd} order BLP Filter is presented in Equation 9.

$$H_{y,u}(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}} \tag{9}$$

The filter coefficients for a 2^{nd} order BLP Filter with cut-off frequency of 10 Hz and sampling time of 5 ms are,

$$\begin{aligned} a_0 &= 0.02008336556421 \\ a_1 &= 0.04016673112842 \\ a_2 &= 0.02008336556421 \\ b_1 &= -1.56101807580072 \\ b_2 &= 0.64135153805756 \end{aligned}$$

The resulting bode plot of the filter is presented in Figure 1. It is clear that the filter has a cut-off frequency of 10 Hz.

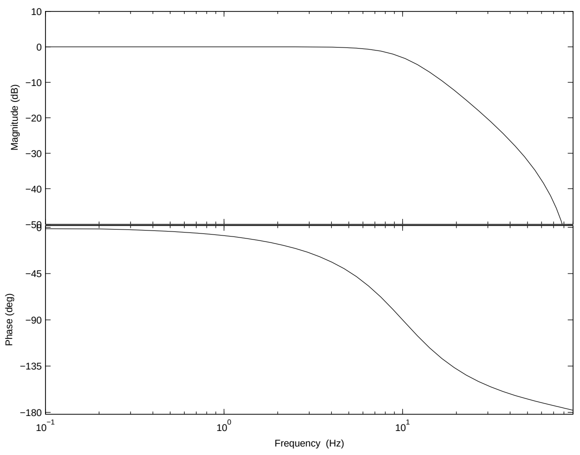

Fig. 1. Bode Plot of the BLP Filter

B. *Fixed Point Arithmetic and the Butterworth Low Pass Filter*

In this section, the representation of transfer functions in computer systems, fixed-point arithmetic and the effects of discrete representation on the digital filter properties will be discussed.

According to [3], the easiest way to realize transfer functions in microcontrollers is to use the *direct form* realization, such that,

$$y(k) = \sum_{i=0}^{m} a_i u(k-i) - \sum_{i=1}^{m} b_i y(k-i) \tag{10}$$

As [3] states, the *direct form* has the advantage of having its state variables as the delayed versions of its input and output signals. This makes the *direct form* realization very easy to implement. However, because the coefficients in the realization are the same as the coefficients in the transfer function, the realization is very sensitive to computational errors especially if the system is of high order or if it has poles or zeros very close to each other. There are other realizations, such as the *ladder realization*, which can avoid this coefficient sensitivity. Nevertheless, the *direct form* realization is used to implement the BLP filter that is designed in the previous section.

Many microcontrollers, such as the FPGA platform with 6502 soft-processor, use integer representation to represent numbers and to do calculations. However, integers can not handle decimals. Decimals consist of an integer part and a fraction part and the integer representation can not differentiate between the two. Therefore, the programmer defines a fixed-point representation to be able to represent the fractions. To do this, some bits of the integer are used to represent the integer part of the decimal and some bits of the integer are used to represent the decimal fraction. Characteristics of the digital filter can be used to decide on the number of fraction and integer bits.

When the transfer function coefficients are implemented using fixed-point representation, the coefficients are perturbed by some amount. This also perturbs the zeros and the poles of the transfer function. Therefore, according to [1], one must choose the number of fraction bits such that the maximum percentage change of the poles relative to the unit circle are less than some percentage ϵ. In other words,

$$\left| \frac{\triangle p_i}{1 - |p_i|} \right| < \epsilon \tag{11}$$

[1] states that the number of fraction bits f, can be chosen such that,

$$f = [-\log_2(\epsilon |1 - |p_i|| \, |p_2 - p_1|)] \tag{12}$$

The poles of the 2^{nd} order BLP filter $H_{y,u}(z)$ designed in the previous section are,

$$p_i \cong 0.780509 \pm 0.179324$$

Having ϵ equal to 0.1%, the Equation 12 becomes,

$$f \cong 13.77$$

for the 2^{nd} order BLP filter. Rounding up f, 14 bits are necessary to represent the decimal fraction in order to guarantee maximum 0.1% change in filter poles with respect to the unit circle. Using 14 bits for the decimal fraction, the coefficients of the 2^{nd} order BLP filter become,

$$\begin{aligned}
a_0 &= 0.02008056640625\\
a_1 &= 0.04016113281250\\
a_2 &= 0.02008056640625\\
b_1 &= -1.56097412109375\\
b_2 &= 0.64129638671875
\end{aligned}$$

The poles of the new transfer function $\tilde{H}_{y,u}(z)$ with the above coefficients are,

$$p_i \cong 0.780487 \pm 0.179266$$

Using Equation 11, the percent change in filter poles is,

$$\left|\frac{\triangle p_i}{1-|p_i|}\right| \cong 0.0312\%$$

which is less than ϵ equal to 0.1%.

The choice of ϵ is also very important. As the poles of the filter perturb, the filter can not generate the same output as the filter with the original coefficients. Therefore, an error e_p is introduced to the output as a result of the perturbations of the filter coefficients. It is possible to calculate the bound on the error e_p according to,

$$\|\triangle y\|_\infty \leq \|g_{e_p,u}\|_1 \|u\|_\infty \tag{13}$$

$\|g_{e_p,u}\|_1$ is the one norm of the impulse response of the system from u to e_p and $\|u\|_\infty$ is the infinity-norm of the input signal u. The one-norm and the infinity-norm of a vector x are defined respectively by,

$$\|x\|_1 := \sum_{i=1}^{n} \|x_i\| \tag{14}$$

$$\|x\|_\infty := max(|x_1|, ..., |x_n|) \tag{15}$$

The transfer function from u to e_p can be computed by,

$$H_{e_p,u}(z) = H_{y,u}(z) - \tilde{H}_{y,u}(z) \tag{16}$$

It is possible to calculate the one-norm of the impulse response from u to e_p numerically. This is done by simulating the system $H_{e_p,u}(z)$ for an impulse input and summing the absolute value of the output e_p at each sampling period [1]. The infinity-norm of u is actually the maximum possible value for the input signal to the system, which is a square signal with magnitude one. The $\|g_{e_p,u}\|_1$ and the $\|u\|_\infty$ for the 2^{nd} order BLP filter are,

$$\begin{aligned}
\|g_{e_p,u}\|_1 &= 0.0003\\
\|u\|_\infty &= 1
\end{aligned}$$

Therefore the maximum amount of error introduced to the output of the system due to the perturbation of the filter coefficients is bounded by 0.0003.

Checking the bound on the range of the calculated variables is a good indication to determine the number of bits necessary for the integer part of the decimal. According to [1], the bound on a particular variable, v, can be calculated according to,

$$\|v\|_\infty \leq \|g_{v,u}\|_1 \|u\|_\infty \tag{17}$$

$\|g_{v,u}\|_1$ is the one-norm of the impulse response of the system from u to v and and $\|u\|_\infty$ is the infinity-norm of the input signal u. For the 2^{nd} order BLP filter, the only calculated variable is the filter output y. It is possible to compute $\|g_{y,u}\|_1$ numerically as it is done for $\|g_{e_p,u}\|_1$. The $\|g_{y,u}\|_1$ and the $\|u\|_\infty$ for the 2^{nd} order BLP filter are,

$$\begin{aligned}
\|g_{y,u}\|_1 &= 1.0931\\
\|u\|_\infty &= 1
\end{aligned}$$

Therefore, the maximum value of the filter output y is 1.0931. As a result, one bit for the integer part of the decimal is enough to implement the filter.

However, internal calculations of the filter should also be considered to avoid overflow. The maximum result that can be internally computed is the result of the multiplication $b_1 y(k-1)$. In the worst case, $y(k-1)$ is equal to 1.0931, which is the maximum value of the filter output y. Therefore the maximum value of the multiplication $b_1 y(k-1)$ is around 1.7. One bit for integer part of the decimal is still enough to implement the filter without overflow.

There is still one more check to be made to ensure that no overflow occurs. In the worst case, when two numbers that have fixed-point representation with one sign bit, n decimal bits and f fraction bits are multiplied, the result has one sign bit, $2n+1$ decimal bits and $2f$ fraction bits. Later the result is shifted f bits to the right to keep the number of fraction bits constant. In the worst case, before shifting bits to the right, the total number of bits that are used must be less than the total number of bits available. If not, either guarding bits should be implemented in the software or the calculations may overflow.

For the 2^{nd} order BLP filter, one sign bit, one decimal bit and 14 fraction bits are used. In the worst case, when two numbers are multiplied, before shifting the result 14 bits to the right, the result has one sign bit, three decimal bits and 28 fraction bits. The total number of bits that are used is equal to 32. Therefore, it can be ensured that with this fixed-point representation, no calculation overflow will occur.

Due to the precision of the fixed-point representation, a truncation error e_t is also introduced to the system. Using fixed-point representation, [3] states that the additions give exact results but truncation error occurs in the multiplications. According to [1], it is possible to calculate the bound on the error e_t introduced to the output as a result of truncation according to,

$$\|\triangle y\|_\infty \leq \|g_{y,e_t}\|_1 \|e_t\|_\infty \tag{18}$$

$\|g_{y,e_t}\|_1$ is the one norm of the impulse response of the system from e_t to y and $\|e_t\|_\infty$ is the maximum error introduced as a result of truncation. It is possible to compute $\|g_{y,e_t}\|_1$ numerically as it is done for $\|g_{e_p,u}\|_1$ and $\|g_{y,u}\|_1$. The transfer function from e_t to y is,

$$H(z) = \frac{1}{1 + b_1 z^{-1} + b_2 z^{-2}} \tag{19}$$

The $\|g_{y,e_t}\|_1$ and the $\|e_t\|_\infty$ for the 2^{nd} order BLP filter are,

$$\|g_{y,e_t}\|_1 = 13.64$$
$$\|e_t\|_\infty = 2^{-f} = 2^{-14}$$

Therefore the maximum amount of error introduced to the system output due to truncation is less than 0.00083 with 14 fraction bits.

C. Results

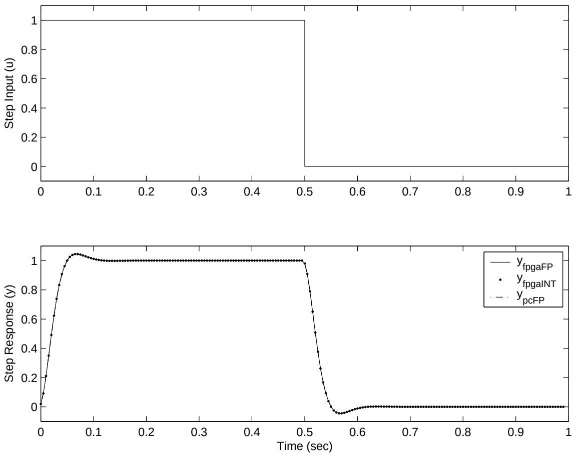

Fig. 2. Step Responses

The BLP Filter designed in Section V-A is tested in FPGA with 6502 soft-processor using the fixed-point representation explained in Section V-B for square input signal. The calculations are made with both using integers and SoftFloat floating-point implementation package. The source code for the two filters can be found in Appendix C. The same filter is also tested in PC using floating-points for comparison. The resulting outputs are ploted in Figure 2 where y_{fpgaFP} stands for the FPGA output calculated using floating-points, $y_{fpgaINT}$ stands for the FPGA output calculated using integers and y_{pcFP} stands for the PC output calculated using floating-points with the original filter coefficients of $H_{y,u}(z)$. In Section V-B, the bound on the output was calculated as,

$$\|y\|_\infty = 1.0931$$

In Figure 2, the maximum value of the output signal is 1.0444 for the floating-point calculations and 1.0445 for the integer calculations, which are less than the calculated $\|y\|_\infty$.

The error between the integer calculations $y_{fpgaINT}$ and floating-point calculations y_{fpgaFP} in the FPGA platform is

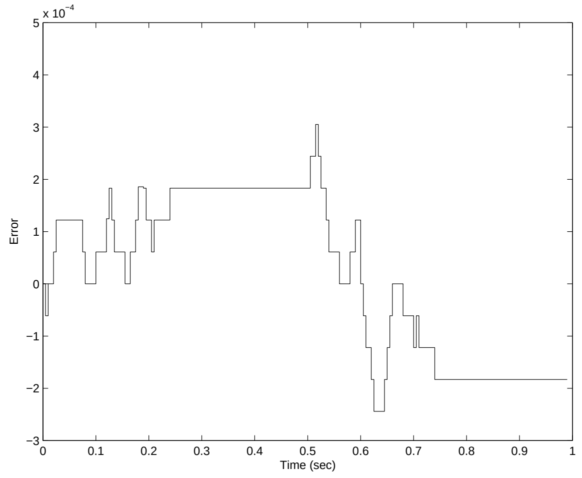

Fig. 3. FPGA Error

presented in Figure 3.

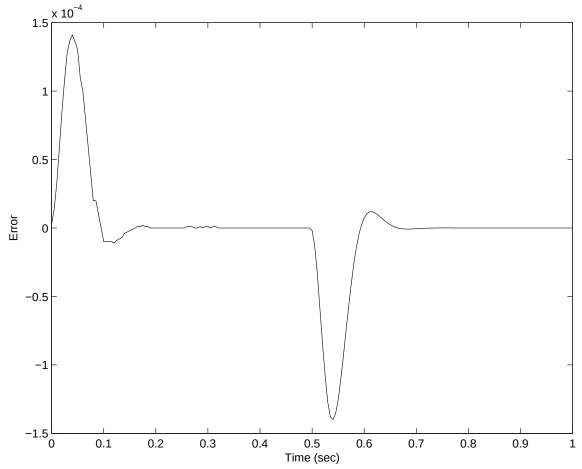

Fig. 4. Error due to Truncation in Filter Coefficients

In Figure 4, the error between $H_{y,u}(z)$ and $\tilde{H}_{y,u}(z)$ is presented, using the outputs calculated in the PC platform. This is to show the error e_p introduced to the system output as a result of the perturbation of the filter coefficients. In Section V-B, the bound on the error was calculated as,

$$\|\triangle y\|_\infty = \|e_p\|_\infty = 0.0003$$

In Figure 4, the maximum error on the system output is equal to 0.00014, which is less than $\|e_p\|_\infty$.

In Figure 5, two error plots are presented. In the above plot, y_{pcFP} is calculated using the original filter coefficients in $H_{y,u}(z)$ in order to show the total error introduced to the system as result of both filter coefficient perturbation and truncations in multiplications. In the second plot, y_{pcFP} is calculated using the truncated filter coefficients in $\tilde{H}_{y,u}(z)$ in order to show only the error e_t introduced to the system output as a result of truncations made in the multiplications. In Section V-B, the bound on the error e_t was calculated as,

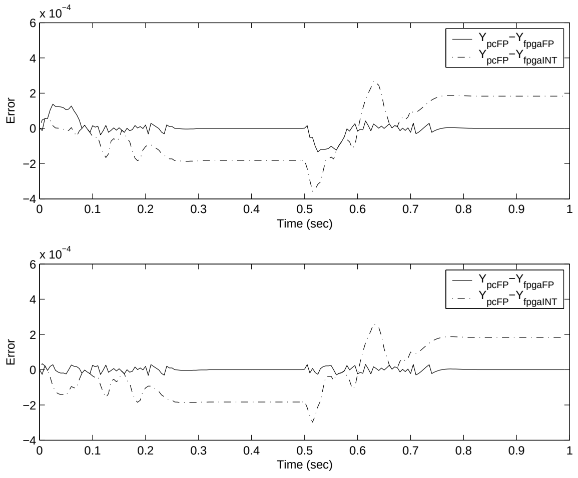

Fig. 5. Error on the FPGA Output

$$\|\triangle y\|_\infty = \|e_t\|_\infty = 0.00083$$

In the second plot presented in Figure 5, the maximum truncation errors introduced to the system outputs are equal to 0.0004 for the integer calculations and 0.0001 for the floating-point calculations. Both of them are less than the calculated $\|e_t\|_\infty$.

The filter is tested for 200 input points. For each sample, the integer calculations took 0.78 ms in the FPGA platform. However, the performance of the SoftFloat floating-point package was 12.56 times slower. For each sample, the floating-point calculations took almost 9.795 ms. Since the sampling time of the BLP Filter is 5 ms, if the filter was tested in real-time, the FPGA platform with 6502 soft-processor would fail to calculate the outputs with floating-point calculations in the given sampling period.

VI. Conclusion

8051s, PICs and FPGAs are compared for control applications. It is concluded that 8051s and PICs are suitable for simple control applications due to their supplier defined functionalities, which limit them in handling complex designs and executing high precision computations. On the other hand, eventhough they are slower and more expensive, FPGAs with soft-processors introduce more flexible microcontroller architectures enabling user defined functionalities depending on the complexity of the application.

It is usually more convenient to use floating-points in scientific and engineering applications. That is why floating-point support is studied on FPGA with 6502 soft-processor in this paper. The IEEE 754 - *Standard for Binary Floating-Point Arithmetic* is discussed for single precision format. The representation of normalized and denormalized floating-point numbers, special values, exceptions, rounding modes and gradual underflow are introduced to the reader. It is emphasized that floating-point implementation is slower compared to integer representation and arithmetic. However, it introduces a greater range and sometimes a higher precision.

The SoftFloat floating-point implementation package is ported on a FPGA platform with 6502 soft-processor. The execution times of the floating-point functions are presented and comparisons with integers are given for the arithmetic functions. The floating-point implementation was much slower compared to integer operations.

A 2^{nd} order Butterworth Low Pass (BLP) Filter is used to test the performance of the SoftFloat package. In the case study, the design of the BLP filter, implementation of filters in microcontrollers and the use of fixed-point arithmetic are explained in detail. A methodology is given in order to choose the number of integer and fraction bits to implement fixed-point arithmetic. Moreover, the effects of truncation and fixed-point arithmetic on the filter properties are discussed.

In the case study, the floating-point package was very slow to implement the 2^{nd} order BLP filter with 5 ms sampling period. The calculations took about 9.795 ms at each sampling period, which means that the filter would fail to calculate filter outputs if it had performed in real-time. However, the performance of the floating-point package was better in terms of the truncation error e_t introduced to the system output compared to the integer computations. Therefore, it can be concluded that, even the performance is better in terms of precision, the SoftFloat floating-point package is very slow on the FPGA platform with 6502 soft-processor. It is not feasible to use the package for periodic applications with high sampling frequencies.

Appendix

A. Porting Details

The most up-to-date information about SoftFloat and the latest releases can be found at the web page [10]. There are six files to be modified in order to port the SoftFloat floating-point package to a FPGA with 6502 soft-processor. Four of these files are target specific and the remaining two are target independent.

1) Target Specific Files:

- *6502.h* - The integer sizes for FPGA with 6502 soft-processor and some C preprocessor macros are defined. BITS64 is not defined since 64-bit integer types are not supported. The compiler does not support explicit inlining. Therefore, this macro is defined to be static. The necessary definitions for this file are,

```
#define LITTLEENDIAN
typedef signed char flag;
typedef unsigned char uint8;
typedef signed char int8;
typedef int uint16;
typedef int int16;
typedef unsigned long uint32;
typedef signed long int32;
typedef unsigned char bits8;
typedef signed char sbits8;
typedef unsigned int bits16;
typedef signed int sbits16;
typedef unsigned long bits32;
typedef signed long sbits32;
#define INLINE static
```

- *softfloat.h* - The following modifications are necessary for this file:

```
typedef !!!bits32 float32;             -> typedef unsigned long float32;
extern !!!int8 float_detect_tininess;   -> extern signed char float_detect_tininess;
extern !!!int8 float_rounding_mode;     -> extern signed char float_rounding_mode;
extern !!!int8 float_exception_flags;   -> extern signed char float_exception_flags;
void float_raise( !!!int8 );            -> void float_raise( signed char );
float32 int32_to_float32( !!!int32 );   -> float32 int32_to_float32( signed long );
!!!int32 float32_to_int32( float32 );   -> signed long float32_to_int32( float32 );
!!!int32 float32_to_int32_round_to      -> signed long float32_to_int32_round
                   _zero( float32 );                        _to_zero( float32 );
!!!flag float32_eq( float32, float32 ); -> char float32_eq( float32, float32 );
!!!flag float32_le( float32, float32 ); -> char float32_le( float32, float32 );
!!!flag float32_lt( float32, float32 ); -> char float32_lt( float32, float32 );
!!!flag float32_eq_signaling            -> char float32_eq_signaling
                 ( float32, float32 );                     ( float32, float32 );
!!!flag float32_le_quiet( float32,      -> char float32_le_quiet( float32,
                            float32 );                                 float32 );
!!!flag float32_lt_quiet( float32,      -> char float32_lt_quiet( float32,
                            float32 );                                 float32 );
!!!flag float32_is_signaling_nan(       -> signed char float32_is_signaling_nan(
                            float32 );                                 float32 );
```

The compiler does not support struct parameter passing. Therefore, it is not possible to port the double-precision floating-point functions to the FPGA platform. The declarations of the following double-precision functions should be removed from the file.

```
typedef struct {!!!bits32 high, low;} float64;
float64 int32_to_float64( !!!int32 );
float64 float32_to_float64( float32 );
```

```
!!!int32 float64_to_int32( float64 );
!!!int32 float64_to_int32_round_to_zero( float64 );
float32 float64_to_float32( float64 );
float64 float64_round_to_int( float64 );
float64 float64_add( float64, float64 );
float64 float64_sub( float64, float64 );
float64 float64_mul( float64, float64 );
float64 float64_div( float64, float64 );
float64 float64_rem( float64, float64 );
float64 float64_sqrt( float64 );
!!!flag float64_eq( float64, float64 );
!!!flag float64_le( float64, float64 );
!!!flag float64_lt( float64, float64 );
!!!flag float64_eq_signaling( float64, float64 );
!!!flag float64_le_quiet( float64, float64 );
!!!flag float64_lt_quiet( float64, float64 );
!!!flag float64_is_signaling_nan( float64 );
```

- *milieu.h* - The only necessary modification for this file is,

```
#include "../../../processors/!!!processor.h"   ->   #include "6502.h"
```

- *softfloat-specialize* - The following functions should be removed from this file:

```
static float32 commonNaNToFloat32( commonNaNT a )
static commonNaNT float32ToCommonNaN( float32 a )
flag float64_is_nan( float64 a )
flag float64_is_signaling_nan( float64 a )
static commonNaNT float64ToCommonNaN( float64 a )
static float64 commonNaNToFloat64( commonNaNT a )
static float64 propagateFloat64NaN( float64 a, float64 b)
```

2) Target Independent Files:

- *softfloat-macros* - The following functions should be removed from the file:

```
INLINE bits32 extractFloat64Frac1              static float64 addFloat64Sigs
INLINE bits32 extractFloat64Frac0              static float64 subFloat64Sigs
INLINE int16 extractFloat64Exp                 float64 float64_add
INLINE flag extractFloat64Sign                 float64 float64_sub
static void normalizeFloat64Subnormal          float64 float64_mul
INLINE float64 packFloat64                     float64 float64_div
static float64 roundAndPackFloat64             float64 float64_rem
static float64 normalizeRoundAndPackFloat64    float64 float64_sqrt
float64 int32_to_float64                       flag float64_eq
float64 float32_to_float64                     flag float64_le
int32 float64_to_int32                         flag float64_lt
int32 float64_to_int32_round_to_zero           flag float64_eq_signaling
float32 float64_to_float32                     flag float64_le_quiet
float64 float64_round_to_int                   flag float64_lt_quiet
```

- *softfloat-macros* - The following functions should be removed from the file:

```
INLINE flag ne64                  INLINE flag lt64
INLINE flag le64                  INLINE flag eq64
INLINE void mul64To128            INLINE void mul64By32To96
INLINE void sub96                 INLINE void add96
INLINE void shift64Right          INLINE void shift64RightJamming
INLINE void shortShift96Left      INLINE void shift64ExtraRightJamming
```

B. Bugs

1 The function *normalizeRoundAndPackFloat32* in *softfloat.c* has a bug with the 6502 soft-processor. When shiftcount is equal to zero, it returns zero result. To avoid this bug, change the original function,

```
static float32 normalizeRoundAndPackFloat32( flag zSign, int16 zExp,
bits32 zSig ) {
    int8 shiftCount;
    shiftCount = countLeadingZeros32( zSig ) - 1;

    return roundAndPackFloat32( zSign, zExp - shiftCount, zSig<<shiftCount );

}
```

with the following,

```
static float32
 normalizeRoundAndPackFloat32( flag zSign, int16 zExp, bits32 zSig )
{
    int8 shiftCount;
    shiftCount = countLeadingZeros32( zSig ) - 1;

    if( shiftCount == 0)
    {
      return roundAndPackFloat32( zSign, zExp, zSig);
    }
    return roundAndPackFloat32( zSign, zExp - shiftCount, zSig<<shiftCount );

}
```

2 The function *roundAndPackFloat32* in *softfloat.c* has a bug with the 6502 soft-processor. Sometimes it changes the sign bit from negative to positive. To avoid this bug, remove the following line:

```
    zSig &= ~ ( ( ( roundBits ^ 0x40 ) == 0 ) & roundNearestEven );
```

C. Digital Filter Source Code

This function is implemented for integer computations in order to multiply two numbers and shift the result 14 bits to the right to keep the number of fraction bits constant:

```
signed long mul(signed long a1,signed long a2){
  signed long ans=(a1*a2);
  return (ans >> 14);
}
```

This loop computes the filter output using direct form realization with fixed-point integer computations:

```
b1=0x000063E7; b2=0x0000290B; b2=b2^0xFFFFFFFF; b2=b2+0x00000001;
a0=0x00000149; a1=0x00000292; a2=0x00000149;
for(h=0;h<200;h++) {
   if(h<100){
       u=(1 << 14);}
   else{
       u=0;}
   y=mul(b1,y_1)+mul(b2,y_2)+mul(a0,u)+mul(a1,u_1)+mul(a2,u_2);
   x_2=x_1;
   x_1=x;
   y_2=y_1;
   y_1=y;
}
```

It is possible to implement the filter in two ways using floating points. First way is to compute the rounded floating-point bit representations of the original filter coefficients and use them to implement the filter. The floating-point bit representations of decimals can be computed using the web page [11]. The second way is to use the same filter coefficients and arithmetic operations that are used for the fixed-point integer computations. Using the first method, the filter coefficients are perturbed less and therefore the filter output is more accurate than the second method. However, second method is used in Section V in order to be able to present the truncation performance of the floating-point arithmetic over integer computations. Otherwise, the filters used for integer and floating-point computations would had different filter coefficients and it would be harder to derive conclusions. This way it is both possible to observe the effects of fixed-point representation and the effects of truncation in multiplication and division from the experiment results. The following loop computes the filter output using direct form realization with floating-point computations:

```
bf1=int32_to_float32(b1); bf2=int32_to_float32(b2);
af0=int32_to_float32(a0); af1=int32_to_float32(a1); af2=af0;
s_14=int32_to_float32(0x00004000); for(h=0;h<200;h++) {
   if(h<100){
       u=u1;}
   else{
       u=u0;}
   sf1=float32_add(float32_div(float32_mul(bf1,yf_1),s_14),
                   float32_div(float32_mul(bf2,yf_2),s_14));
   sf2=float32_add(float32_div(float32_mul(af0,u),s_14),
                   float32_div(float32_mul(af1,u_1),s_14));
   sf2=float32_add(sf2, float32_div(float32_mul(af2,u_2),s_14));
   yf=float32_add(sf1,sf2);
   u_2=u_1;
   u_1=u;
   yf_2=yf_1;
   yf_1=yf;
}
```

References

[1] J. Carletta, R. Veillette, F. Krach, Z. Fang. (2003). *Determining Appropriate Precisions for Signals in Fixed-Point IIR Filters*. DAC.

[2] Intel. (1999). Floating Point Unit. *Intel Architecture Software Developers Manual - Volume 1: Basic Architecture*. http://www.intel.com/design/pentiumii/manuals/243190.htm.

[3] K. J. Astrom, B. Wittenmark. (1997). Realization of Digital Controllers. *Computer-Controlled Systems*, pp. 349-360. Prentice Hall, New Jersey.

[4] Z. Karakehayov, K.S. Christensen, O. Winther. (1999). *Embedded Systems Design with 8051 Microcontrollers*. Marcel Dekker, Inc., New Jersey.

[5] P.H. Staken. (1989). *A Practitioner's Guide to RISC Microprocessor Architecture*. John Wiley & Sons, Inc., New York.

[6] N. Alexandridis. (1993). *Design of Microprocessor-Based Systems*. Prentice Hall, New Jersey.

[7] J.V. Oldfield, R.C. Dorf. (1995). *Field Programmable Gate Arrays - Reconfigurable Logic for Rapid Prototyping and Implementation of Digital Systems*. John Wiley & Sons, Inc., New York.

[8] S.M. Trimberger. (1994). *Field Programmable Gate Array Technology*. Kluwer Academic Publishers, Boston.

[9] R. Mak. (2003). *The Java Programmer's Guide to Numerical Computing*. Prentice Hall, New Jersey.

[10] http://www.cs.berkeley.edu/ jhauser/arithmetic/SoftFloat.html

[11] http://babbage.cs.qc.edu/courses/cs341/IEEE-754.html