Efficient 16-bit Floating-Point Interval Processor for Embedded Systems and Applications

Stéphane Piskorski
Laboratoire de Recherche en Informatique
CNRS – Univ Paris-Sud
F-91405 Orsay Cedex, France

Lionel Lacassagne
Institut d'Electronique Fondamentale
CNRS – Univ Paris-Sud
F-91405 Orsay Cedex, France

Michel Kieffer
Laboratoire des Signaux et Systèmes
CNRS – Supélec – Univ Paris-Sud
F-91192 Gif-sur-Yvette, France

Daniel Etiemble
Laboratoire de Recherche en Informatique
CNRS – Univ Paris-Sud
F-91405 Orsay Cedex, France

Abstract

This paper presents the implementation of a 16-bit interval floating-point unit on a soft-core processor to allow interval computations for embedded systems. The distributed localization of a source using a network of sensors is presented to compare the performance of the proposed processor to those obtained with a general-purpose processor.

1 Introduction

In the last ten years, interval techniques [19, 20] have allowed original solutions for many problems in engineering to be proposed, see, *e.g.*, [10]. One of the main features of interval techniques is their ability to provide *guaranteed* results, *i.e.*, with a verified accuracy, or which are numerically *proved*. Consider for example, a bounded-error parameter estimation problem: the value of some parameter vector has to be estimated from measured data, using a given model structure and bounded measurement errors. In such a context, one may obtain a set which can be proved to contain all values of the parameter vector that are consistent with the model structure, the measured data, and the hypotheses on the noise. Nevertheless, the application of interval techniques in embedded real-time applications is far less developed. The lack of efficient interval hardware support may be a reason for this slower development.

Hardware implementations of interval arithmetic have been mentioned twenty years ago in [16]. Extension of existing hardware platforms have been proposed, *e.g.*, in [26] and [15]. Nevertheless, chip builders were not yet convinced of the usefulness of performing specific adaptation of chips to implement interval analysis. This is why interval analysis is mainly performed by software implementations on general-purpose processors. Interval computations are however quite inefficiently performed on such processors, since the recurrent rounding mode switchings required by interval computations results in also recurrent flush of the processor pipeline [8]. This specific problem led people to study and design dedicated floating-point units (FPU) well suited to double rounding modes (toward $-\infty$ and toward $+\infty$) [15]. Moreover, in many applications, 32-bit FPU are oversized. Measurements, corrupted by errors, do not require to be processed with such an accuracy and in many cases, smaller FPU with reduced precision may fit the application constraints and provide a satisfying accuracy. Thus, for example, 16-bit floating-point computation is an efficient way to tackle both accuracy and dynamic problems encountered in signal and image processing [17], for filtering and convolution-based algorithms.

This paper introduces 16-bit floating-point arithmetic adapted to interval computations. The main idea is inspired by [15], which proposed to implement two 32-bit FPU on the 64-bit FPU of a general-purpose processor. Here, similarly, noticing that a 16-bit FPU is smaller than a 32-bit FPU, two 16-bit FPU (managing the two rounding modes required for interval computations) are shown not being much bigger than a single 32-bit FPU. The main advantage is that no rounding mode switching is required, preventing them from flushing the processor pipeline. The implementation of such a 16-bit FPU is performed on the FPGA based NIOS-II soft processor [2, 6], which allows instructions to be added to its instruction set. Customizable processors represent an opportunity to propose efficient and low-cost on-chip interval applications which may be used in embedded applications.

To compare the performance of 16 and 32-bit FPU, a source localization example using a network of acoustic or electromagnetic sensors is considered. In such network of sensors, power consumption and computational complexity are strong constraints when one is concerned with the increase of operability and autonomy [5]. Distributed interval constraint propagation [12] has been proposed as an efficient and low-complexity solution for source localization using a network of wireless sensors.

Section 2 recalls the distributed source localization problems and sketches the solutions based on interval analysis. In Section 2, the architecture of the 16-bit FPU is presented. Attention is paid to accuracy and dynamic range. Results provided by a 32-bit FPU are compared to those obtained with two 16-bit FPU on realistic simulated data. Section 3 describes the hardware implementation on the two targeted architecture (Pentium4 and NIOS-II) and provides benchmarks for execution time and energy consumption.

2 Source localization with a network of sensors

Localizing a source emitting an acoustic or electromagnetic wave using a set of distributed sensors has received a growing attention in recent years, see, *e.g.*, [25, 24]. The localization technique used depends on the type of information available to the sensor nodes. Time of arrival (TOA), time difference of arrival (TDOA), and angle of arrival (AOA) usually provide the best results [22]. Nevertheless, these quantities are rather difficult to obtain, as they require a good synchronization between timers (for TOA), collaboration between neighboring sensors (for TDOA), or multiple antennas (for AOA). Contrary to TOA, TDOA, or AOA data, readings of signal strength (RSS) at a given sensor are easily obtained, as they only require low-cost measurement devices or are already available, as in IEEE 802.11 wireless networks, where these data are provided by the MAC layer [24].

This paper focuses on source localization from RSS data using a distributed localization technique. Distributed means that each sensor processes its own measurements and exchanges partially processed data with its neighbors. This approach has the advantage of being more robust to sensor failures than centralized localization, where all measurements are processed by a single unit. Distributed approaches have been employed, *e.g.*, in [23], where a distributed version of nonlinear least squares has been presented. When badly initialized, this approach suffers from convergence problems, as illustrated in [7], which advocates projection on convex sets. However, this requires an accurate knowledge of the source signal strength and of the path loss exponent. In [12], a centralized and distributed localization technique based on interval analysis has been proposed. All measurement noises are assumed to be bounded with known bounds. The problem is then cast into a set-inversion problem, which is solved in a centralized fashion using SIVIA [11] and in a distributed manner using interval constraint propagation [10].

The model, hypotheses, and main results of [12] are briefly recalled here, before introducing the implementation of the localization algorithm on a dedicated processor.

Source localization from bounded-error data

The single source localization problem illustrated on Figure 1 is considered. The unknown location of the source

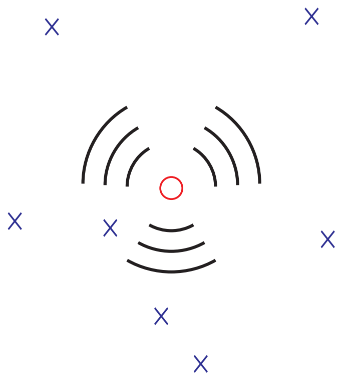

Figure 1. Source (o, unknown location θ) and sensors (x, known location $\mathbf{r}_\ell$)

is denoted by $\theta \in \mathbb{R}^2$; $\mathbf{r}_\ell \in \mathbb{R}^2$, $\ell = 1 \dots L$ represents the known location of the sensors. The RSS by sensor ℓ is denoted by y_ℓ. Using the Okumura-Hata model [21], describing the mean power received by the ℓ-th sensor when a source at a distance $|\mathbf{r}_\ell - \theta|$ emits a wave with power A (measured at a distance of 1 m), one may write

$$y_\ell = y_{\mathrm{m}}\left(\theta, A, n_{\mathrm{p}}, \ell\right) w_\ell,\ \ell = 1 \dots L \tag{1}$$

with

$$y_{\mathrm{m}}\left(\theta, A, n_{\mathrm{p}}, \ell\right) = \frac{A}{|\mathbf{r}_\ell - \theta|^{n_{\mathrm{p}}}}. \tag{2}$$

The noise $w_\ell \in [w]$ is multiplicative, as it is additive in the log domain. In (2), n_{p} is the path-loss exponent. Both A and n_{p} are assumed unknown. The parameter vector to be estimated is thus $\mathbf{p} = (\theta^{\mathrm{T}}, A, n_{\mathrm{p}})^{\mathrm{T}}$ and $y_{\mathrm{m}}\left(\theta, A, n_{\mathrm{p}}, \ell\right)$ will be written as $y_{\mathrm{m}}\left(\mathbf{p}, \ell\right)$.

The aim here is to characterize the *set* $\mathbb{P} \subset [\mathbf{p}]_0$ of all parameter vectors that are *consistent* with the measurements, the RSS model (2), and the noise bounds. The initial search box $[\mathbf{p}]_0$ is assumed to contain the actual parameter value $\mathbf{p}^*$. $\mathbb{P}$ may then be defined as follows

$$\mathbb{P} = \{\mathbf{p} \in [\mathbf{p}]_0 \mid y_{\mathrm{m}}(\mathbf{p}, \ell) \in [y_\ell], \ell = 1 \ldots L\}, \qquad (3)$$

where $[y_\ell] = y_\ell / [w]$ is assumed to contain the actual noise-free RSS by sensor ℓ.

The characterization of $\mathbb{P}$ when all measurements y_ℓ are available at a given location of the sensors network is a classical set-inversion problem which may be solved using set-inversion techniques, like SIVIA [11]. SIVIA provides two *subpavings* (union of non-overlapping boxes) $\underline{\mathbb{P}}$ and $\overline{\mathbb{P}}$, such that

$$\underline{\mathbb{P}} \subset \mathbb{P} \subset \overline{\mathbb{P}}.$$

The accuracy of the description of $\mathbb{P}$ is controlled by a parameter ε, which determines the width of the smallest box in $\underline{\mathbb{P}}$ and $\overline{\mathbb{P}}$.

This approach may be easily extended in a distributed context: each sensor ℓ may evaluate two subpavings $\underline{\mathbb{P}}_\ell$ and $\overline{\mathbb{P}}_\ell$, guaranteed to enclose the set

$$\mathbb{P}_\ell = \{\mathbf{p} \in [\mathbf{p}]_0 \mid y_{\mathrm{m}}(\mathbf{p}, \ell) \in [y_\ell]\},$$

of all values of $\mathbf{p}$ consistent with the ℓ-th measurement. The solution set $\mathbb{P}$ in (3) is then obtained by intersecting the sets $\mathbb{P}_\ell$, $\ell = 1 \ldots L$

$$\mathbb{P} = \bigcap_{\ell=1}^{L} \mathbb{P}_\ell.$$

Performing this intersection is far from being trivial. This requires the transmission of the solution sets $\mathbb{P}_\ell$ between sensors. Such exchanges require many communications, a highly energy consuming task, especially when the desired accuracy (determined by ε) is high, resulting in a large number of boxes stored in each subpaving. The next section describes an alternative approach based on interval constraint propagation (ICP) [10].

Distributed localization: interval constraint propagation

Here, contrary to classical constraint satisfaction problems, the variables and constraints are *distributed* over the sensor network, [3, 1]. However, ICP may still be used at each individual sensor.

At a sensor ℓ, the variables are y_ℓ, θ, A, and n_{p}, their domains are $[y_\ell]$, measured at the sensor, and $[\theta]$, $[A]$ and $[n_{\mathrm{p}}]$, obtained from its neighbors. The variables must satisfy the constraint provided by the RSS model (2), thus

$$y_\ell - \frac{A}{|\mathbf{r}_\ell - \theta|^{n_{\mathrm{p}}}} = 0. \qquad (4)$$

From (4), the *contracted domains* [10] may be written as

$$\begin{aligned}
&[y_\ell'] = [y_\ell] \cap \frac{[A]}{|\mathbf{r}_\ell - [\theta]|^{[n_{\mathrm{p}}]}},\\
&[A'] = [A] \cap [y_\ell'] \, |\mathbf{r}_\ell - [\theta]|^{[n_{\mathrm{p}}]},\\
&[n_{\mathrm{p}}'] = [n_{\mathrm{p}}] \cap \left(\log([A']) - \log([y_\ell'])\right) / \log(|\mathbf{r}_\ell - [\theta]|),\\
&[\theta_1'] = [\theta_1] \cap \left(r_{\ell,1} \pm \sqrt{([A'] / [y_\ell'])^{2/[n_{\mathrm{p}}']} - (r_{\ell,2} - [\theta_2])^2}\right),\\
&[\theta_2'] = [\theta_2] \cap \left(r_{\ell,2} \pm \sqrt{([A'] / [y_\ell'])^{2/[n_{\mathrm{p}}']} - (r_{\ell,1} - [\theta_1])^2}\right).
\end{aligned}$$

In the last two update equations, the set intersecting $[\theta_1]$ and $[\theta_2]$ may consist of two disconnected intervals. In this case, the smallest interval containing the result is evaluated. For each sensor, ICP provides thus a box which is guaranteed to contain the set $\mathbb{P}$ defined by (3).

Using [9], it can be shown that the contraction is optimal with respect to the information available at the ℓ-th sensor. However, when considering all constraints simultaneously, the optimality conditions no longer hold. Cycling through the sensor network, as in [23, 7] improves the estimation. More details about the optimization of sensor communications may be found in [3].

In this distributed approach, a single box has to be transmitted from one sensor to its neighbors, which is much less energy consuming than transmitting a subpaving. To perform all computations at a given sensor, basic interval arithmetic operations have to be implemented ($+$, $-$, $\times$, and $/$), the intersection between intervals, and the square root and square of an interval. Elementary functions $\ln(\cdot)$ and $\exp(\cdot)$ have also to be provided. The next section shows how interval constraint propagation may be implemented on a dedicated FPGA using 16-bit floating-point numbers.

Why 16-bit floating-point numbers?

On one hand, for some applications in image and signal processing (*e.g.*, filtering), 32-bit floating-point (FP) numbers are oversized. On the other hand, fixed-point arithmetic may be inaccurate (*e.g.,* when considering recursive filtering, motion estimation, image stabilization). Moreover, for embedded applications, a 32-bit FPU is larger than a 16-bit FPU and thus consumes more power. As interval computation requires rounding mode switching, designing a customized 16-bit interval FPU is a way to tackle both problems together, as will be seen with the Altera NIOS-II soft processor [2].

F16 format

Some years ago, a 16-bit FP format called *half* has been introduced in the OpenEXR format [4] and in the Cg language [18] defined by NVIDIA. It is currently being used in some NVIDIA and ATI graphical processing units (GPU) for transformation and lightning (3D computations).

The *half* format is justified by ILM, which developed the OpenEXP format, as a response to the demand for higher color fidelity in the visual effect industry:

> "16-bit integer based formats typically represent color component values from 0 (black) to 1 (white), but do not account for over-range value (e.g. a chrome highlight) that can be captured by film negative or other HDR displays. Conversely, 32-bit floating-point TIFF is often overkill for visual effects work. 32-bit FP TIFF provides more than sufficient precision and dynamic range for VFX images, but it comes at the cost of storage, both on disk and memory."

Similar arguments are used to justify the *half* format in Cg language. *Half* format is used to reduce storage cost, while computations are done with 32-bit FP formats, either by the CPU or the GPU.

In the remainder of this paper, this 16-bit FP format will be called F_{16} and the IEEE-754 32-bit single-precision FP format will be called F_{32}. Both formats are represented in Figure 2. F_{16} has a 1-bit sign, a 5-bit exponent with a bias of 15 and a 7-bit mantissa. A number is interpreted exactly as in the other IEEE FP formats. The range of the format extends from 6×10^{-5} to $2^{16} - 2^5 = 65504$.

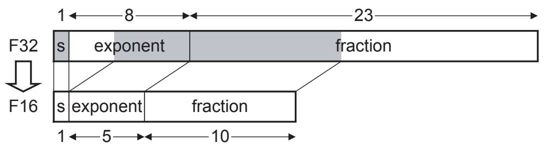

Figure 2. F_{32} and F_{16} floating-point numbers

From an architectural point of view, as many algorithms belong to the class of memory bounded problems, reducing the size of floating-point numbers could have a great impact on performance (just because twice less bytes have to be transfered); for algorithms with a small computational ratio (that is the number of mathematic computations per point divided by the number of memory accesses per point), their speed is limited by the memory bandwidth of the processor.

From an algorithmic point of view, the format can be tuned to an application, by allocating more bits to the mantissa to increase the accuracy, or to the exponent to increase the dynamic range. The bias can be also customized to favor numbers smaller or bigger than the unit. In embedded systems, in order to balance the hardware cost and the algorithms accuracy, lighter FP coding with 15, 14 or even only 13 bits may be considered. Moreover, as F_{16} instructions require less logic than F_{32}, their implementation may lead to faster clock frequencies on an FPGA than F_{32}. Finally with such an approach, custom floating-point formats may be finely tuned to the application and to the embedded constraints.

Moreover, as interval algorithms require from twice up to four times more arithmetic computations that non-interval algorithms, where the speed of arithmetic operators depends on the width of their operands, reducing the number of bits, from 32 down to 16 could be a good trade-off between execution time and accuracy.

2.1 F16 intervals

When running interval algorithms on a RISC processor, even by using a library like PROFIL/BIAS [13, 14] to target real-time implementations (or at least quick implementation), the main problem lies in swapping the rounding mode used for floating-point computations. IEEE 754 defines 4 modes, and interval computations use two of them, towards $+\infty$ and $-\infty$. On most processors, this change forces the processor to flush its pipeline. The deeper the latter is, the bigger the resulting cycle penalty is.

FPGA and soft core processors can be an issue by designing two FPU; the first one is dedicated to computations with rounding towards $-\infty$ the second one towards $+\infty$. With two FPU, changing the rounding mode is no more required. Another important point is that, on a 32-bit architecture, 16-bit numbers and 16-bit operators can be viewed as a natural way of doing parallel computations: two F_{16} numbers are stored inside 32-bit words and define an F_{16} interval number. Such numbers are sent to an interval Floating-Point Unit (*iFPU*) where computations are done in parallel on the two bounds.

The soft core used for this implementation is the NIOS-II from Altera. In this customizable 32-bit RISC processor, new instructions, new data formats, and functions can be easily added in the C language.

Two Altera kits have been used: Cyclone and Stratix II. The Cyclone device has up to 20k Logic Elements and 288 kb SRAM. The Stratix II device has up to 60k logic elements, 2 Mb SRAM, and 36 DSP blocks which can implement 144 18 b$\times$18 b multipliers. In our benchmarks, these DSP blocks introduce a significant difference in the multiplier performance. Both devices can use the NIOS-II softcore CPU. The processor main features are summarized in Table 1. New instructions can be customized and added to the CPU ISA [2], as shown in Figure 3, in the *custom logic* block. The hardware operators for the customized instructions are described with VHDL or Verilog. Two types of operators can be defined. The combinational operators are used when the propagation delay is shorter than the processor cycle time. In that case, the defined interfaces are the 32-bit input data (dataa and datab) and the output data (result). When the propagation delay is greater than the clock cycle time, multi-cycle operations must be used. They have the same data interface than the combinational operators, plus clock and control signals: `clk` (processor clock), `clk_enable`, a global reset (`reset`), a `start` signal (active when the input data are valid) and a `done` signal (active when the result is available for the processor). Since the processor uses a 32-bit data interface, it is natural to define all our instructions as interval instructions: each one operates simultaneously on two 16-bit FP operands. This is a big advantage of using F_{16} operands as it doubles the throughput of operations. Using the customized instructions in a C program is straightforward. Two types of "`define`" are used as the instructions can have one or two input operands:

- `#define INST1(A) __builtin_custom_ini (Opcode_INSTR1, (A))`
- `#define INST2 (A, B) __builtin_custom_inii (Opcode_INSTR2, (A), (B))`

For both kits, the minimal clock frequency is 50 MHz. The maximal clock frequency depends on the complexity of the customized arithmetic instructions, and is higher for the Stratix II kit. Quartus II software has been used to generate the FPGA configuration files form the NIOS-II file and the VHDL code for the customized instructions. All the benchmarks have been compiled with the Altera Integrated Development Environment (IDE), which uses the GCC tool chain. The optimizing option `-O2` has been used in release mode. Execution times have been measured with the `high_res_timer` that provides the number of processor clock cycles for the execution time. We only provide results for the Stratix II kit which is faster than the Cyclone one.

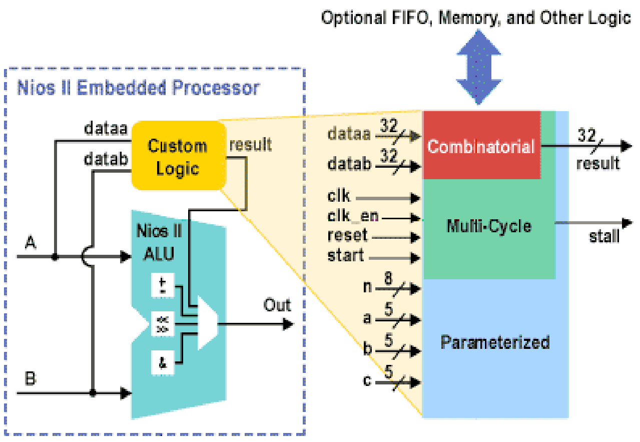

Figure 3. NIOS II architecture

Table 1 presents the fixed and tunable features of the NIOS-II processor. The clock frequency depends on the operator complexity and is usually in the range [100 MHz. . . 200 MHz]. Note that cache size are customizable from 2 and 4 kB up to 64 kB. Cache sizes have been set to the minimum to save space on the FPGA.

fixed features
32-bit RISC processor
branch prediction
dynamic branch predictor
barrel shifter
parameterized features
HW integer multiplication and division
4 KB instruction cache
2 KB data cache
customized instruction

Table 1. NIOS-II main features

3 Application

A network of $L = 5000$ sensors randomly distributed over a field of 100 m×100 m is considered, as well as a single source, such that $\theta^* = (50\,\text{m}, 50\,\text{m})$, $A = 100$. The measurement noise is such that $e = 4$ dBm. Table 2 provides typical measurements with their associated interval. Only the sensors receiving significant amount of power ($y_\ell > 5$) participate to the localization.

number of sensors	measurements
68	$[9.303, 58.698]$
741	$[17.856, 112.664]$
954	$[18.644, 117.640]$

Table 2. Example of measurements

3.1 Qualitative results: F_{32} vs F_{16} accuracy

In order to evaluate the impact of using F_{16} in place of F_{32}, the estimation algorithm has first been simulated with a Pentium M running the distributed constraint propagation algorithms using the PROFIL/BIAS library with F_{32} arithmetic operators and functions (*log*, *exp*, *power*, and *sqrt*) and a F_{16} version of the same library (the size of the mantissa is reduced to 10 bits). When the algorithm converges (which requires between 2 and 3 cycles through the sensor network), a solution box is provided, which is guaranteed to contain the actual position of the source, provided that the hypotheses on the measurement noise were not violated.

Figure 4 represents two histograms (for F_{32} and F_{16}) of the absolute value of the localization error. Figure 5 provides an histogram of the maximum width of projection of the solution box on the θ-plane. For the simulation, 2000 runs with 5000 randomly distributed sensors were considered. The location of each sensor is represented using 16-bit floating-point values, in order to have exactly the same input data. The difference between the F_{32} and F_{16} results thus only comes from the smaller accuracy of F_{16} numbers (10 bits for F_{16} mantissa instead of 23 bits for the F_{32} one).

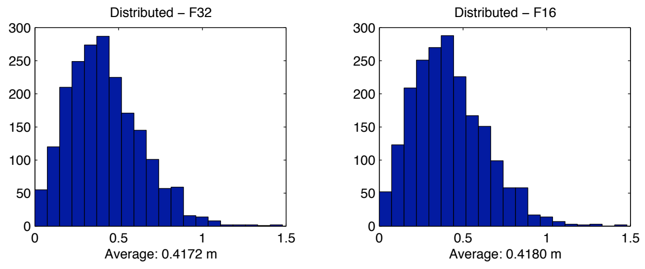

Figure 4. Localisation error: F_{32} vs F_{16}

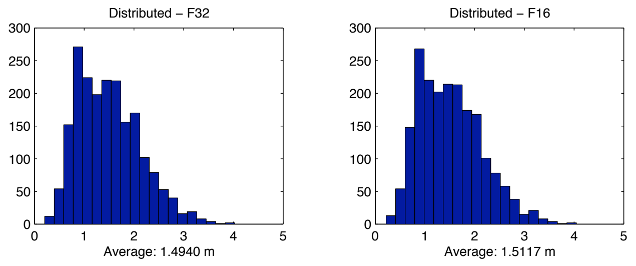

Figure 5. Diameter of the solution box: F_{32} vs F_{16}

The results obtained with F_{16} are very similar to those obtained with F_{32}. The average error and average width differ by less than 1%. For this kind of application, F_{16} provides a satisfying accuracy and dynamic range. The next section gives quantitative results on power consumption and on the required number of logical units on the FPGA.

Quantitative results: F_{16} interval vs F_{32} standard instructions

Table 3 provides a comparison between F_{32} and F_{16} for the number of blocks required for each instruction. The hardware description language (HDL) descriptions of the `DIV`, `POW`, `SQRT`, and `LOG` operators have been taken from [4]. For these operators the rounding mode *to nearest* has been replaced by the two rounding modes towards -∞ and towards +∞.

Note that the 32-bit FPU only implements `ADD`, `MUL`, and `DIV`. Complex instructions like `POW`, `SQRT`, and `LOG` should

Block	F_{16}	F_{32}
Frequency	50 MHz	50 MHz
CPU	2018	1831
32-bit FPU	-	5137
`ADD`	1441	-
`MED`	770	-
`MUL`	811	-
`DIV`	958	-
`POW`	439	782
`SQRT`	431	754
`EXP`	832	1289
`LOG`	898	1898
total	8590	11691

Table 3. Instructions size in block

be added in both cases. The implementation of `POW` is based on iterative calls to `MUL`. For these two instructions, the number of blocks is small, as DSP blocks incorporated on Stratix II FPGAs are used. `POW` and `MUL` respectively use two and four 18×18 DSP blocks. As can be seen in Table 3, an interval F_{16} processor is smaller than a scalar F_{32} processor.

Tables 4 and 5 present a benchmark between a Pentium4 running the F_{32} PROFIL/BIAS library and a NIOS-II with F_{16} iFPU. For NIOS-II, a cycle accurate chronometer is directly available. For the Pentium, two timers are available. The first one is based on the *Query Performance Counter* API which accuracy is better than 1 ms. The second one is a high performance counter (a 64-bit hardware register) that holds the total number of clock cycles elapsed from the boot.

As clock frequencies are very different, we prefer to use a normalized metric like the *cpi*, which is the number of processor cycles per iteration (total number of clock cycles divided by the number of iterations). It is a good metric to estimate the adequation between the algorithm and the architecture.

For embedded systems, the notion of performance does not only implies speed, but also power/energy consumption. Pentium 4 processors are fast but also very power consuming. If we look at the energy (an important metric for embedded systems) we can see that a NIOS2 processor on a StratixII device is twice more efficient. Note also, that because no more rounding mode switch is required, the NIOS2 *cpi* is smaller than the one of a Pentium 4, despite the large number of hardware optimizations present in the latter.

Two localization algorithms have been implemented. The first one (Algorithm 1), presented in this paper (see Section 2), estimates the path loss exponent. The second one (Algorithm 2), assumes that the path loss exponent is known, which requires much less computations. Comparing these algorithms gives an idea of the price to pay (in terms of cycles per iteration) to estimate this additional information. Besides, the first of these two methods leads to data-dependent computation time. In our simulations, a satisfying convergence level was reached after three iterations of the algorithm in the whole sensor field. Before convergence, one iteration takes about 45000 cycles (to compared to the 5000 cycles needed by the second algorithm) and about 9000 cycles after convergence, when fewer additional information can be extracted from the measurements.

	Pentium 4	NIOS-II
frequency	2.4 GHz	50 MHz
time (ms)	11.4	186
cpi	54800	9000 to 45000
power (W)	70 W	1 W
energy (mJ)	798 mJ	186 mJ

Table 4. Algorithm 1, unknown path loss exponent: P4 vs NIOS-II

	Pentium 4	NIOS-II
frequency	2.4 GHz	50 MHz
time (ms)	0.833	26
cpi	4000	2600
power (W)	70 W	1 W
energy (mJ)	58 mJ	26 mJ

Table 5. Algorithm 2, known path-loss exponent: P4 versus NIOS-II

Tables 4 and 5 provide results for Pentium 4 and NIOS-II running the two algorithms for 5 iterations and 100 sensors participating to the localization (500 iterations). Estimating the path loss exponent requires seven times more computing power than not estimating this quantity. Comparing the NIOS-II to the Pentium 4, it can be seen that even for Algorithm 2, involving mathematical functions (finely tuned by Intel inside its processor), NIOS-II is 4.3 times more efficient than Pentium 4.

Conclusion

This paper has presented the evaluation of 16-bit floating-point operators and customizable floating-point formats for embedded systems performing source localization with interval computions.

While the implementation of such type of algorithms has some drawbacks on general-purpose processors (only one FPU with pipeline flushes because of rounding mode switching), the presented customized NIOS-II processor with two 16-bit FPU, one for each rounding mode, tackles these drawbacks. As formats are customizable, interval FPU can be tuned to the application (accuracy and dynamic range) with an efficient energetic implementation. Compared to a classical 32-bit implementation on a RISC computer, the proposed solution is 4.3 times more efficient.

Future works tend to develop high-level tools to perform an automatic design space exploration of the configurations to efficiently implement F_{16} format for interval computation into an FPGA. Right now, customization has been done at the instruction level, by adding new operations adapted to interval computations. Next step is to envision function customization that is to design an hardware accelerator implementing a full iteration of the localization algorithm. Some tools already exist like the Altera C2H compiler that directly *compiles* in hardware a C function. The current version of C2H is only able to compile integer functions, not FP ones. When such a kind of tool will be able to do so, processor customization will be available to non VHDL specialists.

References

[1] Special issue on distributed constraint satisfaction. In B. Faltings and M. Yokoo, editors, *Artificial Intelligence*, volume 161, pages 1–250, 2005.

[2] Altera. *NIOS Custom Instructions, Tutorial*, 2002. http://www.altera.com/literature/tt/tt_nios_ci.pdf.

[3] R. Bejar, C. Fernandez, M. Valls, C. Domshlak, C. Gomes, B. Selman, and B. Krishnamachari. Sensor networks and distributed CSP: Communication, computation and complexity. *Artificial Intelligence Journal*, 161(1-2):117–148, 2005.

[4] J. Detrey and F. de Dinechin. *FPLibrary: A VHDL Library of Parametrisable Floating-Point and LNS Operators for FPGA*. ENS Lyon, 2006. http://perso.ens-lyon.fr/jeremie.detrey/FPLibrary/.

[5] A. Dogandzic, J. Riba, G. Seco, and A. Lee Swindlehurst, editors. *Location is Everythink*, volume 22, 2005.

[6] D. Etiemble, S.Bouaziz, and L. Lacassagne. Customizing 16-bit floating point instructions on a NIOS II processor for FPGA image and media processing. In *Proc. Estimedia*, New York, 2005.

[7] A. O. Hero III and D. Blatt. Sensor network source localization via projection onto convex sets (POCS). In *Proceedings of ICASSP*, 2005.

[8] Intel. Desktop performance and optimization for intel pentium 4 processor. Technical Report 249438-01, 2001.

[9] L. Jaulin, M. Kieffer, I. Braems, and E. Walter. Guaranteed nonlinear estimation using constraint propagation on sets. *International Journal of Control*, 74(18):1772–1782, 2001.

[10] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer-Verlag, London, 2001.

[11] L. Jaulin and E. Walter. Set inversion via interval analysis for nonlinear bounded-error estimation. *Automatica*, 29(4):1053–1064, 1993.

[12] M. Kieffer and E. Walter. Centralized and distributed source localization by a network of sensors using guaranteed set estimation. In *Proceedings of ICASSP*, 2006. submitted.

[13] O. Knüppel. PROFIL - programmer's runtime optimized fast interval library. Technical Report 93.4, Institut für Informatik III, Technische Universität Hamburg-Harburg, Germany, 1993. Available at: ftp://ftp.ti3.tu-harburg.de/pub/reports/report93.3.ps.Z.

[14] O. Knüppel. PROFIL/BIAS – A fast interval library. *Computing*, 53:277–287, 1994.

[15] R. Kolla, A. Vodopivec, and J. Wolff Von Gudenberg. The iax architecture: Interval arithmetic extension, 1999.

[16] U. Kulisch and W. L. Miranker. The arithmetic of digital computer: A new approach. *Siam Review*, 28(1), 1986.

[17] L. Lacassagne, D. Etiemble, and S. Ould Kablia. 16-bit floating point instructions for embedded multimedia applications. In *Proc. IEEE Computer Architecture and Machine Perception*, Palermo, 2005.

[18] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *Proceedings of SIGGRAPH 2003*, 2003.

[19] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.

[20] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, Cambridge, UK, 1990.

[21] Y. Okumura, E. Ohmori, T. Kawano, and K. Fukuda. Field strength ans its variability in VHF and UHF land-mobile radio service. *Rev. Elec. Commun. Lab.*, 16:9–10, 1968.

[22] N. Patwari, J. N. Ash, S. Kyperountas, A. O. Hero III, R. L. Moses, and N. S. Correal. Locating the nodes. *IEEE Signal Processing Magazine*, 22(4):54–69, 2005.

[23] M. G. Rabbat and R. D. Nowak. Decentralized source localization and tracking. In *Proc. ICASSP*, 2004.

[24] A. H. Sayed, A. Tarighat, and N. Khajehnouri. Network-based wireless location. *IEEE Signal Processing Magazine*, 22(4):24–40, 2005.

[25] G. Sun, J. Chen, W. Guo, and K. J. Ray Liu. Signal processing techniques in network-aided positioning. *IEEE Signal Processing Magazine*, 22(4):12–23, 2005.

[26] J. Wolff von Gudenberg. Hardware support for interval computation. In G. Alefeld, A. Frommer, and B. Lang, editors, *Scientific Computing and Validated Numerics*, pages 32–37. Akademie-Verlag, Berlin, Germany, 1996.