Design of fixed-point rounding operators for the VHDL-2008 standard

Nikolaos Kavvadias and Kostas Masselos
Department of Computer Science and Technology
University of Peloponnese
22100 Tripoli, Greece
Email: {nkavv,kmas}@uop.gr

***Abstract*—The contemporary design of sophisticated digital signal processing platforms involves the use of specifications at an increasingly raised abstraction level. This scheme is dictated by the ever growing divide between available circuit complexity and developer productivity. Algorithm developers tend to use very high-level programming languages such as MATLAB in order to rapidly and seamlessly generate low-level design facets such as ANSI C reference implementations and synthesizable HDL code.**

In this paper, a generic and parameterized implementation of fixed-point rounding operators in the VHDL hardware description language is introduced. Most hardware compilation frameworks either lack the support of these operators or provide specialized and non-portable implementations. Further, this is the first time that an implementation for these operators is being proposed, that can take advantage of features only present in the VHDL-2008 standard. Compared to existing fixed-point rounding, the proposed combinatorial designs achieve lower timing by about 30% with similar area demands for the case of signed arithmetic compared to rival designs when realized on FPGAs.

I. Introduction

Current VLSI technology allows the design of sophisticated digital systems with escalated demands in performance and power/energy consumption. The annual increase of chip complexity is 58% [1], consistently reflected by Moore's law [2], while human designers' productivity increase is limited to 21% per annum. The growing gap between the technological capabilities and designer productivity is probably the most important problem in the industrial development of innovative products.

Apart from secondary factors (design reuse, human experience), a dramatic increase in designer productivity, is only possible through the adoption of methodologies and tools that raise the design entry abstraction level, ingeniously hiding low-level, time-consuming, error-prone details. New electronic design automation (EDA) methodologies aim to automate register transfer level (RTL) digital designs from high-level descriptions, a process called High-Level Synthesis (HLS) [3]. The input to this process is usually an algorithmic description in a high-level programming language (Java, C, C++, MATLAB), generating digital circuits in Verilog/VHDL.

Recent EDA tools provide algorithm specification and code generation facilities than enable a quick path for concept to implementation. A popular numerical analysis language, MATLAB [4], and related open-source projects (Scilab [5], Octave [6]) are highly popular among algorithm engineers from many disciplines. These environments are extensible using plug-in mechanisms in order to introduce advanced functionalities to meet user demands. A popular plug-in for developers of digital signal processing (DSP) algorithms [7] especially from the field of communication systems modelling is the Fixed-Point Toolbox [8]. The use of fixed-point arithmetic [9] provides an inexpensive means for improved numerical dynamic range, when artifacts due to quantization and overflow effects can be tolerated. Rounding operators are used for controlling the numerical precision involved in a series of computations; they are defined for inexact arithmetic representations such as fixed-point and floating-point schemes. Commonly used rounding schemes include rounding towards plus (ceiling) or minus (flooring) infinity. To ease algorithm development, the MATLAB programming language defines four additional operators: `fix`, `round`, `nearest`, `convergent` that have different interpretation to `ceil` and `floor`. These operators can be considered as the common denominator of existing rounding schemes found in numerical analysis software in general.

The support of fixed-point arithmetic in several programming language specifications has been included as an afterthought, often due to user demand. As of current, there does not exist a language-independent standard for fixed-point arithmetic. The C99 standard [10] defines fixed-point data types and MATLAB supports such types via an external toolbox. A lightweight ANSI C implementation of fixed-point arithmetic has been sketched in [11]. A complete implementation of a counterpart to the SystemC [12] fixed-point data types is part of the Algorithmic C Data Types suite [13], [14]; this work focuses on offering a plethora of rounding modes. The latest approved IEEE 1076 standard (termed VHDL-2008) [15] adds signed (`sfixed`) and unsigned (`ufixed`) fixed-point data types and a set of primitives for their manipulation [16]. The VHDL fixed-point package provides synthesizable implementations of fixed-point primitives for arithmetic, scaling and operand resizing [17].

In this work, generic and parameterized implementations of all MATLAB-like rounding operators are presented for the first time. These operators have been designed as vendor-independent VHDL source code and are part of a package

that is freely available in the web[1]. The designs are highly-optimized combinatorial functions so that they can be included as atomic operations in modern soft-core microprocessors. Further, this work attempts to complement the existing fixed-point VHDL-2008 package and provide a useful extension of synthesizable rounding operators.

The remainder of this paper is organized as follows. Section II overviews previous research on the subject. In Section III, complete and detailed designs of the fixed-point rounding operators are presented from a hardware point of view. Section IV discusses area and timing characterization of FPGA implementations for the operators across different devices and provides comparisons to third-party implementations, when such are accessible. Finally, Section V summarizes the paper.

II. Related work

Generally, there is lack of open implementations of fixed-point rounding primitives. However, design of such a hardware module library is necessary for the implementation of predefined accelerator units of Embedded MATLAB intrinsic functions. Last years, a number of software compilation and high-level synthesis tools have been developed with the purpose of translating a subset of MATLAB to interpretable bytecode [18], native code (through an ANSI C compiler) [19], or a high-performance hardware architecture [20]–[23].

McLab [18] is an academic project aiming at the development of an extensible compiler toolkit for MATLAB source code. The McLab approach applies JIT (Just-In Time) compilation to McLab IR (intermediate representation) which can be either interpreted (McVM) or compiled to native code. The same project attempts to bridge the semantic differences between MATLAB and the traditional workhorse for scientific computation, Fortran (McFor). Though, certainly an interesting project, the future roadmap does not seem to involve hardware generation.

GraphLab [24], [25] is an unreleased academic project for multi-language high-level synthesis. The supported source languages include a subset of MATLAB. The author acknowledges the need of predefined hardware libraries; a necessity when targeting specifications utilizing black-boxes such as MATLAB. Still, the integrated pre-characterized libraries used in GraphLab do not include a standard-compliant library for supporting fixed-point rounding.

AccelDSP [20] is a discontinued Xilinx [26] product promoted as a designer assist for the task of DSP algorithm compilation to FPGA-oriented hardware. The AccelDSP documentation clearly states that a path to hardware is provided for the fixed-point rounding operators, however neither source code listings nor experimental results can be found, which would prove useful for comparison purposes.

Synopsys Synphony [22] is an ambitious framework for MATLAB-to-hardware compilation that was announced in 2010. Synphony includes a synthesizable fixed-point high-level IP model library, however certain issues remain unclear: a) the interdependence with a MATLAB/Simulink installation, b) the specific MATLAB subset supported (certain dynamic aspects of the language are far from trivial to support), and c) accessible examples so that the capabilities of Synphony can be assessed.

Simulink HDL coder [23] is a mature code generator by MathWorks that focuses rather on block diagram-based than textual specifications. It is offered as an add-on package to the MATLAB environment and utilizes an undisclosed vendor-dependent library for supporting fixed-point arithmetic.

The proposed fixed-point extension library delivers a straightforward solution to the problem of implementing fixed-point rounding operators in hardware. In the form of a vendor-independent package, the relevant VHDL source code can be included in the implementation of third-party tools, e.g. in the development of an RTL VHDL code generation backend. In this paper, we show that the proposed operators' design induces small hardware demands on modern FPGAs. Further, the exact performance in terms of propagation delay and area demands is evaluated across multiple devices in order to highlight performance trends on FPGA processes with smaller geometries.

III. Design of fixed-point rounding operators

This section presents an introduction to the `sfixed` and `ufixed` VHDL-2008 data types, followed by the detailed designs for fixed-point binary rounding.

A. Short primer on fixed-point rounding

Fixed-point arithmetic is a variant of the typical integral representation (2's-complement signed or unsigned) where a binary point is defined, purely as a notational artifact to signify integer powers of 2 with a negative exponent. Assuming an integer part of width $IW > 0$ and a fractional part of $-FW < 0$, the VHDL-2008 `sfixed` data type has a range of $2^{IW-1} - 2^{|FW|}$ to -2^{IW-1} with a representable quantum of $2^{|FW|}$. Correspondingly, `ufixed` has the following range: $2^{IW} - 2^{|FW|}$ to -2^{IW-1}. Actually, the VHDL fixed-point package allows for greater freedom (e.g. $FW < 0$) but in this work the following are assumed:

- The binary point is located between the 2^0 and 2^{-1} powers
- At least one fractional bit is defined (i.e. the fixed-point numbers are not degenerate integers)
- At least one integral bit is defined

According to VHDL-2008, fixed-point signals can be declared as shown in Fig. 1.

The current MATLAB specification defines six distinct cases of fixed-point rounding. These intrinsic functions are considered as black-boxes by third-party tools and no general implementations exist, that are compatible to VHDL-2008. Table I shows the rounding primitives with a short description for each, followed by an illustrative set of numerical examples. Small MATLAB programs have been used for generating

[1] http://www.opencores.org/project,fixed_extensions

```
signal fxp1 : sfixed(4 downto -5);
signal fxp2 : ufixed(7 downto -8);
// using generics
signal fxp3 : sfixed(IW-1 downto -FW);
```

Fig. 1. Partial VHDL description of the index generation unit for NLP=3.

```
...
if (arg'low > 0) then
  y := resize(arg, arg'high, arg'low);
  return y;
end if;
```

Fig. 2. Rounding of a quantized integer.

reference data vectors so that the corresponding VHDL RTL rounding operators can be dynamically verified.

B. The resize primitive

The `resize` function can be used as a rounding and saturation primitive for adjusting the size of fixed-point operands [27]. As inputs, it accepts a fixed-point operand, a left and a right index bound and two additional arguments that specify the rounding (underflow handling) and saturation (overflow handling) mechanisms. While different underflow (rounding to nearest and truncation) and overflow (saturation or wrap-around) schemes are supported, they do not provide the extensive functionality required for emulating MATLAB rounding.

The resize primitive is an integral part of the VHDL-2008 standard, thus this work utilizes it for the optimized implementation of MATLAB rounding schemes. All the operators discussed in section III-C use `resize` with the default overflow/underflow modes (`fixed_round`, `fixed_saturate`) as an essential building block.

C. Implementation of MATLAB-like rounding

In the following implementations, signal y is considered as the output and arg as the input of each operation. Also, it is assumed that an explicit check of the LSB (Least Significant Bit) index takes place. This check can be removed and the corresponding multiplexers (where the result of this comparison is used for selection) can be reduced to simple wiring, if the general case of arbitrary sign of IW, FW is not of interest.

The special case for $IW > 0, -FW > 0$ defines a quantized integer (QI) representation with a resolution of $2^{|FW|}$. The rounding of a QI number is performed in the VHDL implementation as shown in Fig. 2.

Similarly, $IW < 0, -FW > 0$ define a quantized fractional (QF) representation with a resolution of $2^{|FW|}$. The rounding of a QF number is performed as shown in Fig. 3 and it always computes to zero, since $0 \leq arg_{QF} \leq 1 - 2^{|FW|}$.

1) ceil: Both the signed and unsigned ceiling operators can be implemented with the same hardware. To achieve rounding towards $+\infty$, arg is increased by one when arg has a non-zero fractional part, otherwise it is passed directly to `resize` (multiplexer MUX1). The result range (`arg'HIGH downto 0`) is passed directly to the output. The fractional part of the result is selected from either the range (`-1 downto arg'LOW`) or from a zeroed vector, the latter if the LSB index is negative (equal or less than -1). In this way, fixed-point numbers with any FW value can be handled. This runtime check can be omitted if it is guaranteed that arg is not a quantized integer.

```
...
if (arg'high <= 0) then
  y := (others => '0');
  return y;
end if;
```

Fig. 3. Rounding of a quantized fractional.

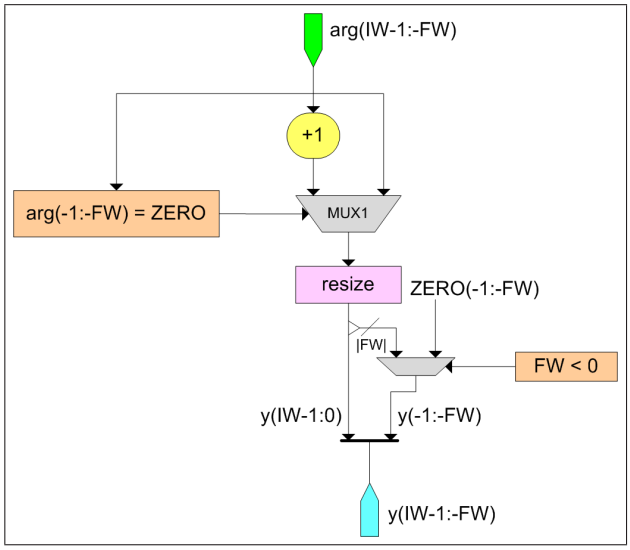


Fig. 4. Implementation of the `ceil` operator.

If necessary, the same approach for handling the fractional part is used in all implementations. Since vector upper and lower bounds ($arg'HIGH$, $arg'LOW$) are known at design compile-time, this structure can be eliminated by the logic synthesis tool. The corresponding RTL design is shown in Fig. 4.

2) fix: Two different designs are needed for implementing the `fix` operator for signed and unsigned arithmetic. In the circuit needed for the `sfixed` variant, if arg is negative then either it is directly resized or it is first incremented by 1, depending on the comparison of its fractional part to zero. The choice of the incremented arg is not needed for the positive case. The corresponding circuit can be seen in Fig. 5.

The implementation of `fix` rounding on unsigned fixed-point numbers involves only the resizing of arg. This is the simplest possible circuit for a rounding operation and is illustrated in Fig. 6.

3) floor: The `floor` operator can be realized with the help of the `fix` design for `ufixed` numbers. `floor` rounds towards $-\infty$ which means that it implements a consistent scheme for either positive or negative values (signed arithmetic) or if considering only magnitude (unsigned).

TABLE I
SUMMARY OF THE FIXED-POINT ROUNDING OPERATORS DEFINED BY THE MATLAB FIXED-POINT TOOLBOX.

Operator	Description (round towards ...)	Values											
		-3.5	-2.5	-1.75	-1.5	-1.25	-0.5	0.5	1.25	1.5	1.75	2.5	3.5
`ceil`	$+\infty$	-3	-2	-1	-1	-1	0	1	2	2	2	3	4
`fix`	zero	-3	-2	-1	-1	-1	0	0	1	1	1	2	3
`floor`	$-\infty$	-4	-3	-2	-2	-2	-1	0	1	1	1	2	3
`nearest`	nearest; ties to greatest absolute value	-3	-2	-2	-1	-1	0	1	1	2	2	3	4
`round`	nearest; ties to $+\infty$	-4	-3	-2	-2	-2	-1	1	2	2	2	3	4
`convergent`	nearest; ties to closest even	-4	-2	-2	-2	-1	0	0	1	2	2	2	4

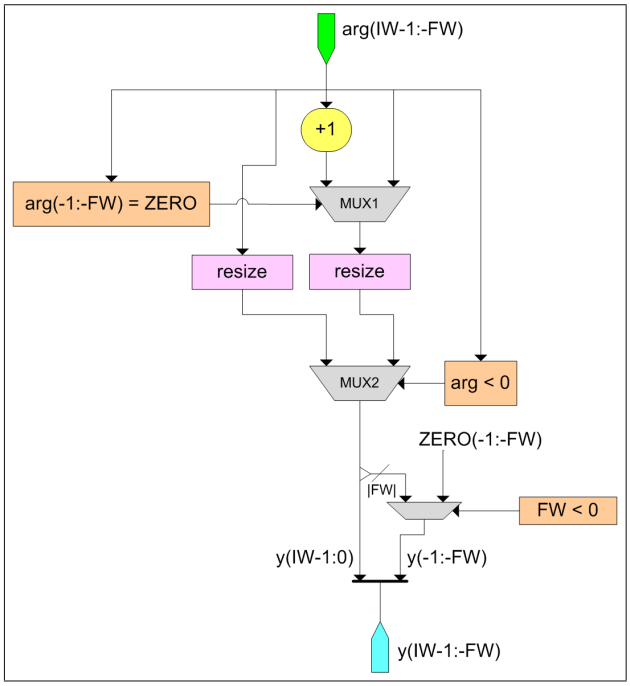


Fig. 5. Implementation of the `fix` operator for signed arithmetic.

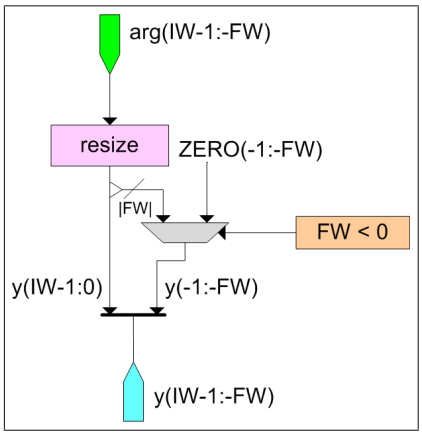


Fig. 6. Implementation of the `fix` operator for unsigned arithmetic.

4) round: Applying rounding-to-nearest with the `round` operator scheme requires comparisons to the value of $\frac{1}{2}$ (ONEHALF). If the fractional part of a positive arg is greater than or equal to $\frac{1}{2}$ then it has to be incremented prior resizing. A negative value for arg is correspondingly resized when its fractional part is strictly greater than $\frac{1}{2}$. The design is shown in Fig. 7.

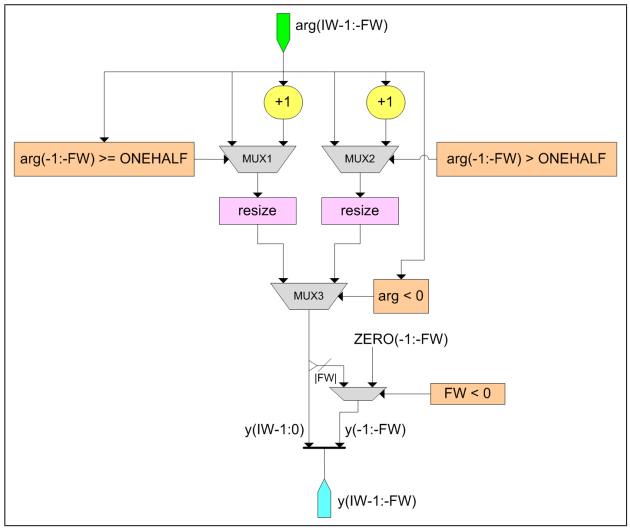


Fig. 7. Implementation of the `round` operator for signed arithmetic.

The unsigned variant of `round` requires a subset of the aforementioned circuit. If deemed necessary, the `round` operators can share a common hardware implementation without any modification.

5) nearest: The signed and unsigned `nearest` operators share a single design. If the fractional part of arg is greater or equal to $\frac{1}{2}$, then its incremented value needs to be resized. This is performed by the `arg(-1) = '1'` comparator which determines whether to round towards zero (choice 0) or $+\infty$ (choice 1). The additional multiplexing shown in Fig. 8 is only required for performing runtime checks of the lower bound for arg and can be automatically eliminated at design elaboration time. The unsigned `nearest` is implemented by the same circuit, since the same behaviour is expected for either positive signed or unsigned arithmetic.

6) convergent: The circuit for `convergent` rounding presents similarities to signed `round` and it is shown in Fig. 9. It requires comparisons to the value of $\frac{1}{2}$ as well as to determine whether the integral part of arg is odd. The latter comparison is needed for applying a rounding towards even integers to resolve a tie. The final result for the integral part is selected from two possible values from multiplexers MUX1 and MUX2. These choices are selected based on the results

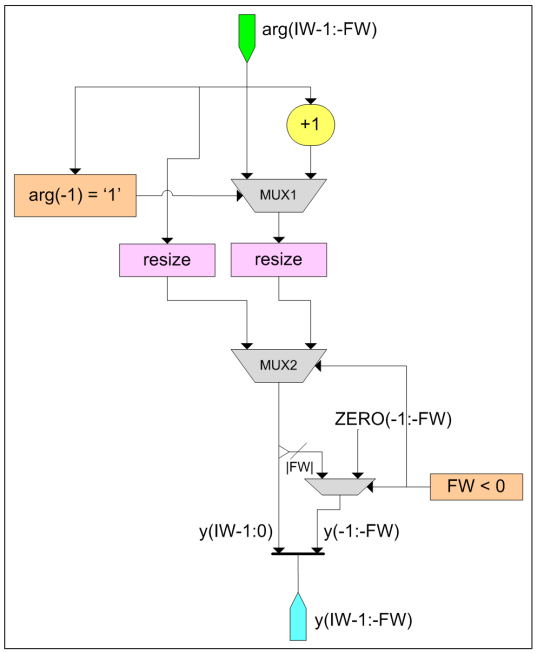


Fig. 8. Implementation of the `nearest` operator.

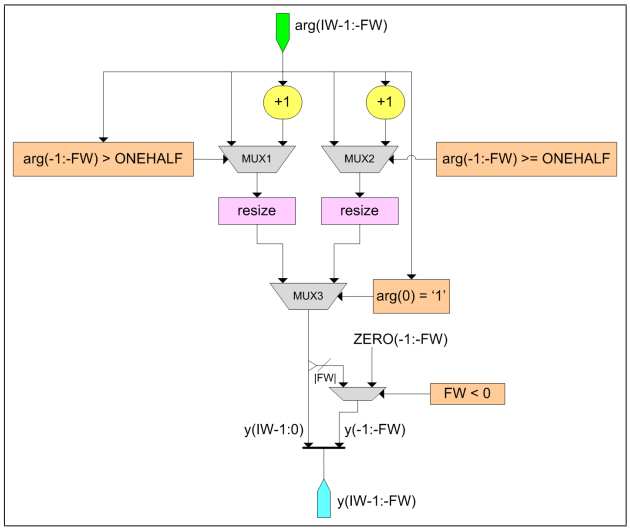


Fig. 9. Implementation of the `convergent` operator.

of comparators CMP1 and CMP2, respectively. CMP1 checks whether the fractional part of arg is greater to the ONEHALF vector which has a constant value of $\frac{1}{2}$, while CMP2 performs a greater-than-or-equal comparison to ONEHALF.

The VHDL source code for `convergent` implementing all possible checks for the signs of the boundary indices of the arg vector, which correspond to checks of the IW and FW parameters is shown in Fig. 10. This version can be even more generalized by avoiding constant indexing of the arg vector. By providing IW, FW as input variables (or in VHDL-2008 as package generics), the LSB (Least Significant Bit) of the integral part (indexed by 0) can be referred to by `arg'HIGH-IW`. Correspondingly, the MSB index (-1) of the fractional part is equivalent to `arg'low+FW-1`.

```
function convergent (arg : sfixed) return sfixed is
  variable result: sfixed(arg'high downto arg'low);
  variable onehalf: std_logic_vector(-arg'low-1 downto 0)
                  := (others => '0');
begin
  if (arg'high <= 0) then
    result := (others => '0');
    return result;
  end if;
  if (arg'low > 0) then
    result := resize(arg, arg'high, arg'low);
    return result;
  end if;
  onehalf(-arg'low-1) := '1';
  if (arg(0) = '1') then
    if (to_slv(arg(-1 downto arg'low)) >= onehalf) then
      result := resize(arg + 1, arg'high, arg'low);
    else
      result := resize(arg, arg'high, arg'low);
    end if;
  else
    if (to_slv(arg(-1 downto arg'low)) > onehalf) then
      result := resize(arg + 1, arg'high, arg'low);
    else
      result := resize(arg, arg'high, arg'low);
    end if;
  end if;
   if (arg'low < 0) then
    result(-1 downto arg'low) := (others => '0');
  end if;
   return result;
end function convergent;
```

Fig. 10. Complete VHDL source code for the `convergent` operator.

IV. Performance Evaluation of the Fixed-Point Rounding Units

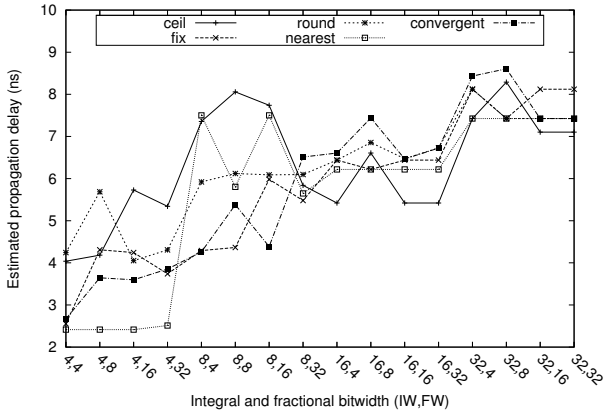
To assess the performance characteristics of the proposed fixed-point rounding units, they are evaluated over the IW, FW parameters for the following value set; $IW(FW)$: $4, 8, 16, 32$ which defines a set cardinality of 16. For each point in the parameter set and for both `ufixed` and `sfixed` variants of the designs, the timing (minimum propagation delay) and area requirements are measured for three representative FPGA processes: the $90nm$ Spartan-3 and Virtex-4 5-input LUT and the $40nm$ Virtex-6 6-input LUT process. The logic synthesis tool used is Xilinx Webpack ISE 12.3i.

Throughout the evaluations, the XC3S200, XC4VLX25 and XC6VLX75T devices have been selected, correspondingly for each one of the processes. All of them rank amongst the smallest devices in their respective family. For example, the maximum capacity of XC4VLX25 is 7,200 slices (which translates to 28,800 6-input LUTs), 96 18-kbit block RAMs (BRAMs) and 48 DSP48E datapath blocks. Both BRAMs and embedded multiplier (Spartan-3)/DSP (Virtex-4/6) blocks remain unused by the rounding logic.

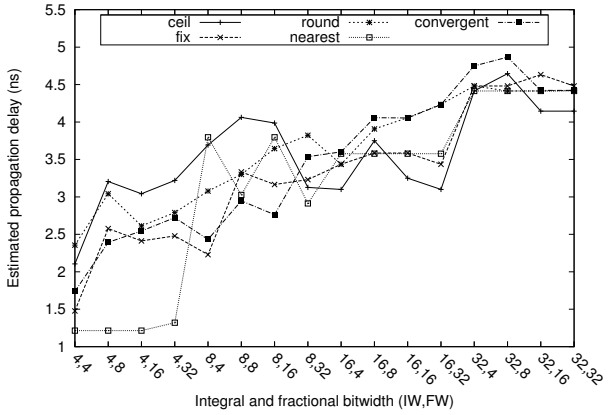
In the following results, the `sfixed fix` and `ufixed fix` and `floor` operators are not shown, since their circuits are optimized to plain wiring by the logic synthesis tool.

A. Speed measurements

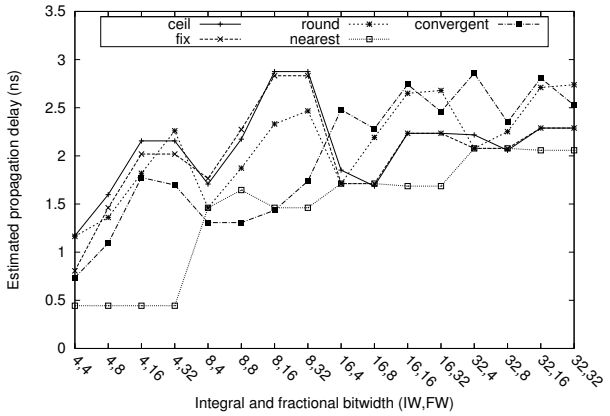
All six rounding operators have been designed in VHDL (as part of the `fixed_extensions` package) and have been synthesized for the three selected devices. Fig. 11 depicts the



(a) XC3S200 Spartan-3 FPGA.



(b) XC4VLX25 Virtex-4 FPGA.



(c) XC6VLX75T Virtex-6 FPGA.

Fig. 11. Estimated propagation delay (ns) for the fixed-point rounding units.

estimated propagation delay for different integral (IW) and fractional part (FW) widths for the `sfixed` versions.

For the examined cases, estimated timings are less than $9ns$ for the largest configuration ($IW, FW = 32, 32$). Average timings vastly improve in newer device generations as expected: the designs are faster by 67% and 44% on the XC6VLX75T device compared to XC3S200 and XC4VLX25, respectively. All circuits have an estimated delay of less than $3ns$ on the Virtex-6 device. `ufixed` designs achieve better timings to their `sfixed` counterparts since they tend to be simpler. This difference ranges from 14.2% (Virtex-4) to 17.8% (Spartan-3). Further, it should be noted that certain results appear non-monotonic; for instance specific $(8, 32)$ operators are faster than their $(8, 16)$ counterparts. This appears to be due to synthesis tool artifacts, provoked by imbalanced LUT input distribution, differences in LUT fan-out loading, routing choices and the usage of specialized primitives. Specifically, for the case of the XC3S200 device, a `MUXF5` wide multiplexer is being utilized for the $(8, 32)$ configuration while this optimization is not attainable for $(8, 16)$.

The `nearest sfixed` operator consistently has the fastest hardware implementation. The slowest `sfixed` operator is either `ceil, fix` or `round` while `convergent` appears with the fastest `ufixed` implementation.

B. Chip area measurements

The chip area requirements are shown in Fig. 12. The Spartan-3 and Virtex-4 devices share a 5-input LUT architecture, with a maximum of ± 1 difference in number of LUTs for any test case.

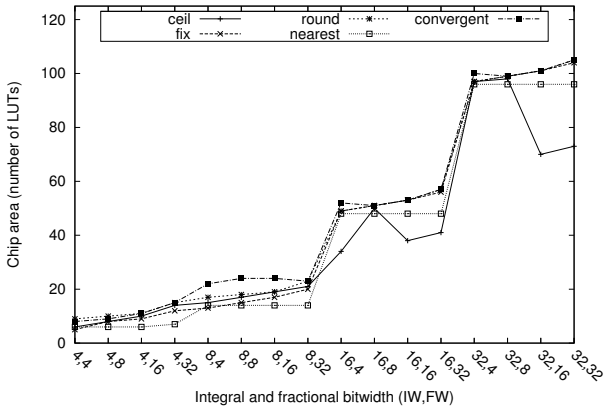
The area requirements for the fixed-point rounding units range from a few to 105 LUTs for the largest integral and fractional widths (Spartan-3/Virtex-4). The `ufixed` circuits require no more than 74 LUTs, or about 27% less than from the `sfixed` ones. It seems that this difference is due to sparsely occupied LUTs; both types of circuits have very similar demands (a deviation of less than 2 LUTs) on the 6-input LUT Virtex-6 device. Thus, it appears these LUTs are much more densely populated.

The `convergent` operator presents the higher area demands irrespective to the arithmetic scheme. `ceil` is the smallest operator on the Spartan-3 and Virtex-4 devices, whereas `nearest` is such for Virtex-6. The latter also applies to the `round` operator for `ufixed` arithmetic since it shares a common design with `nearest` rounding.

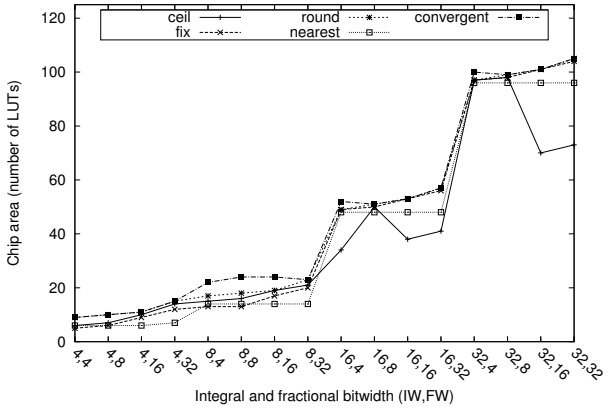
C. Comparison of the proposed rounding units against MATLAB Simulink HDL coder

The proposed units have been compared with the VHDL implementations generated by the MATLAB Simulink HDL coder IP generator (MATLAB 7.8.0, version R2009a). The generator ("hdlcoder" for short) produces optimized hardware realizations of the corresponding MATLAB operators from internal template models using the `numeric_std` IEEE library package. We generalized these models in terms of the IW and FW generic parameters.

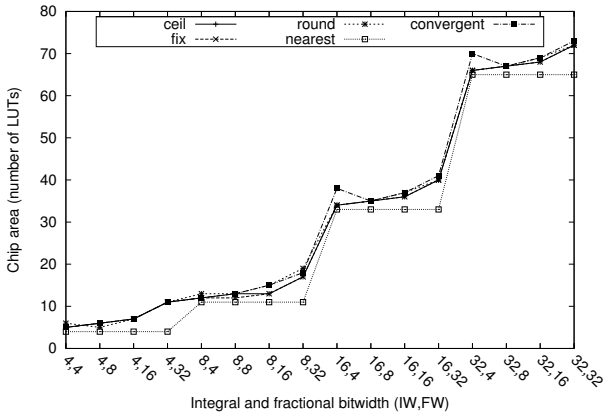
The *hdlcoder* models differ in several aspects from the proposed work. Their characteristics include: a) extensive use of reduction OR trees, b) need for an additional bit of accuracy, c) use of decrement operations, and d) use of several additions with zero-extensions. Due to the architecture of modern FPGAs, reduction ORs prove beneficial due to the exploitation of hardwired, wide multiplexer primitives. On the contrary, in some cases the internal accuracy of computations



(a) XC3S200 Spartan-3 FPGA.

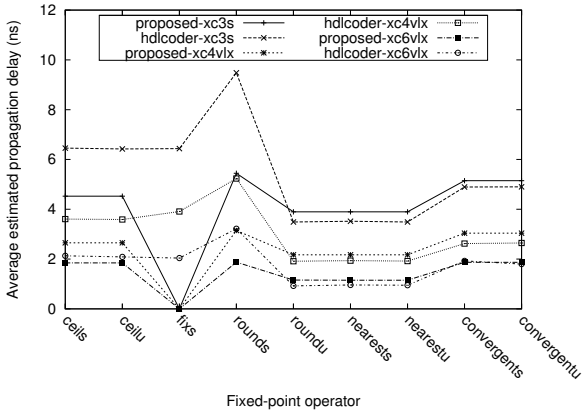


(b) XC4VLX25 Virtex-4 FPGA.

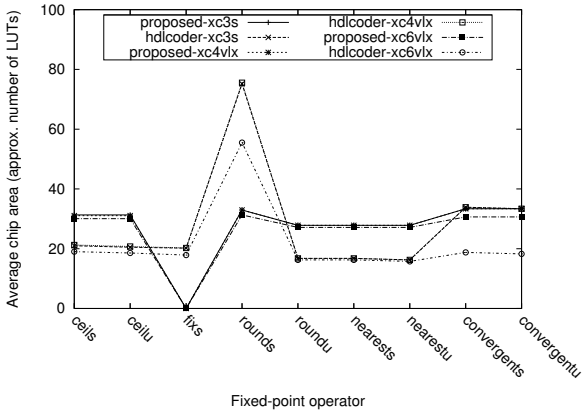


(c) XC6VLX75T Virtex-6 FPGA.

Fig. 12. Chip area in number of LUTs for the fixed-point rounding units.



(a) Average propagation delay (ns) over different FPGA devices.



(b) Chip area in number of LUTs for different FPGA devices.

Fig. 13. Performance metrics for the *proposed* and *hdlcoder* versions of the fixed-point rounding units.

is not carefully optimized. It should be noted, that the *proposed* `fixs` operator reduces to plain wiring, while this is not the case for the HDL coder.

Fig. 13 illustrates the estimated propagation delay and area demand for all cases of signed and unsigned rounding operators. The depicted data points correspond to averages over the same set of (IW, FW) configurations to the previous subsections. To perform a fair comparison, a version of the proposed operators that uses the `resize` primitive of `numeric_std` for the `signed` and `unsigned` vector types was examined, similarly to the *hdlcoder* models.

In general, the *proposed* units for signed arithmetic outperform the Simulink HDL coder models in terms of estimated propagation delay. For the Spartan-3, Virtex-4 and Virtex-6 devices, the *proposed* units are faster by 31.3%, 27.7% and 27.6%, respectively. However, only the *proposed* `ceilu` is faster than its counterpart; *hdlcoder* unsigned units for `round`, `nearest` and `convergent` excel over the *proposed* ones. This is due to the nature of unsigned arithmetic, allowing for case-optimized bit-level manipulations implemented by the *hdlcoder*. For instance, the implementation of unsigned `convergent` by the *hdlcoder* uses a reduction `OR` gate. `nearest` adds a resized arg to its zero-extended sign bit using an additional bit of accuracy, allowing for the elimination of costlier post-processing.

Regarding chip area, the *proposed* signed operators are comparable to the *hdlcoder* models. Specifically, `rounds` requires much less area than its rival. However, the *hdlcoder* unsigned operators consume much less area by an amount of 38.5%-67%, from oldest to newest FPGA device. For the highly capacitive devices of nowadays, the area penalty to be paid is negligible and certainly justified when it is

accompanied by speed improvements.

It is clear that the *proposed* designs of `ceilu`, `ceils`, `fixs`, `rounds` should be chosen for speed while for the remaining cases, the mature *hdlcoder* implementations suffice. Again, this is the first comprehensive work dealing with the important issue of finite-precision (fixed-point) rounding. Additional rounding schemes have been proposed in the past and are summarized in [28], but tend to be ad-hoc and are less frequently used to the well-known MATLAB operators.

V. Conclusion

In this paper, novel schemes for implementing fixed-point binary rounding are introduced. The presented architectures have been described in detail while they have been made freely available in the form of an RTL VHDL package for inclusion in user designs. The implementation is completely vendor-independent using only VHDL constructs compatible with the IEEE standard and the associated libraries.

Thorough experimental measurements have been shown over three representative FPGA devices of different families (Spartan-3, Virtex-4, Virtex-6). Further, the rounding hardware produced by the Simulink HDL coder, an industry standard, has been evaluated for comparison purposes on the same devices. It has been shown that the generic signed-arithmetic hardware of this work, due to its careful design and optimizations achieves much better timing (by 30%) with comparable area demands. It should be noted that with the exception of operator `ceilu`, the HDL coder models perform better for the remaining cases of unsigned arithmetic.

References

[1] International technology roadmap for semiconductors. [Online]. Available: http://www.itrs.net/reports.html

[2] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, no. 8, pp. 114–117, April 1965.

[3] P. Coussy and A. Morawiec, Eds., *High-Level Synthesis: From Algorithms to Digital Circuits*. Springer, 2008.

[4] MathWorks Inc. [Online]. Available: http://www.mathworks.com

[5] Scilab. [Online]. Available: http://www.scilab.org

[6] GNU Octave. [Online]. Available: http://www.gnu.org/software/octave/

[7] S. Roy and P. Banerjee, "An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 886–896, July 2005.

[8] MATLAB Fixed-Point Toolbox. [Online]. Available: http://www.mathworks.com/products/fixed/

[9] R. Yates, "Fixed-point arithmetic: An introduction," Digital Signal Labs, Technical reference, July 7 2009.

[10] *ISO/IEC 9899:TC3 International Standard (Programming Language: C), Committee Draft*, September 2007. [Online]. Available: http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf

[11] S. A. Edwards, "Using program specialization to speed SystemC fixed-point simulation," in *Proceedings of the Workshop on Partial Evaluation and Progra Manipulation (PEPM)*, Charleston, South Carolina, USA, Jan. 2006, pp. 21–28.

[12] *IEEE 1666™-2005: Open SystemC Language Reference Manual*, March 2006.

[13] Algorithmic C data types. [Online]. Available: http://www.mentor.com/products/esl/high_level_synthesis/ac_datatypes

[14] A. Takach, S. Waters, and P. Gutberlet, "Fast bit-accurate C++ datatypes for functional system verification and synthesis," in *Forum on specification and Design Languages (FDL 2004)*, Lille, France, Sep. 2004, pp. 337–345.

[15] *IEEE 1076-2008 Standard VHDL Language Reference Manual*, Jan. 2009.

[16] D. Bishop. VHDL-2008 support library. [Online]. Available: http://www.eda.org/fphdl/

[17] P. J. Ashenden and J. Lewis, *VHDL-2008: Just the New Stuff*. Elsevier/Morgan Kaufmann Publishers, 2008.

[18] McLab: An extensible compiler toolkit for MATLAB. [Online]. Available: http://www.sable.mcgill.ca/mclab

[19] Embedded MATLAB. [Online]. Available: http://www.mathworks.com/products/featured/embeddedmatlab/index.html

[20] Product discontinuation notice: AccelDSP synthesis tool. [Online]. Available: http://www.xilinx.com/support/documentation/customer_notices/xcn09018.pdf

[21] B. L. Gal. GraphLab: high-level synthesis and more. [Online]. Available: http://uuu.enseirb.fr/~legal/wp_graphlab/Home.html

[22] Synopsys Synphony Model Compiler. [Online]. Available: http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/Synphony-Model-Compiler.aspx

[23] Simulink HDL Coder. [Online]. Available: http://www.mathworks.com/products/slhdlcoder/

[24] B. L. Gal and E. Casseau, "Word-length aware DSP hardware design flow based on high-level synthesis," *Integration, the VLSI Journal, Elsevier*, vol. 62, pp. 341–357, March 2011.

[25] E. Casseau and B. L. Gal, "Multi-mode core design based on high-level synthesis," *Integration, the VLSI Journal, Elsevier*, vol. 45, no. 1, pp. 9–21, July 2011.

[26] Xilinx home page. [Online]. Available: http://www.xilinx.com

[27] D. Bishop, *Fixed point package user's guide*. [Online]. Available: http://www.eda.org/fphdl/fixed_ug.pdf

[28] C. Maxfield. An introduction to different rounding algorithms. [Online]. Available: http://www.eetimes.com/design/programmable-logic/4014804/An-introduction-to-different-rounding-algorithms