Sabrewing: A Lightweight Architecture for Combined Floating-Point and Integer Arithmetic

TOM M. BRUINTJES, KAREL H. G. WALTERS, SABIH H. GEREZ, BERT MOLENKAMP, and GERARD J. M. SMIT, University of Twente

In spite of the fact that floating-point arithmetic is costly in terms of silicon area, the joint design of hardware for floating-point and integer arithmetic is seldom considered. While components like multipliers and adders can potentially be shared, floating-point and integer units in contemporary processors are practically disjoint. This work presents a new architecture which tightly integrates floating-point and integer arithmetic in a single datapath. It is mainly intended for use in low-power embedded digital signal processors and therefore the following design constraints were important: limited use of pipelining for the convenience of the compiler; maintaining compatibility with existing technology; minimal area and power consumption for applicability in embedded systems. The architecture is tailored to digital signal processing by combining *floating-point fused multiply-add* and *integer multiply-accumulate*. It could be deployed in a multi-core system-on-chip designed to support applications with and without dominance of floating-point calculations.

The VHDL structural description of this architecture is available for download under BSD license. Besides being configurable at design time, it has been thoroughly checked for IEEE-754 compliance by means of a floating-point test suite originating from the IBM Research Labs. A proof-of-concept has also been implemented using STMicroelectronics 65nm technology. This prototype supports 32-bit signed two's complement integers and 41-bit (8-bit exponent and 32-bit significand) floating-point numbers. Our evaluations show that over 67% energy and 19% area can be saved compared to a reference design in which floating-point and integer arithmetic are implemented separately. The area overhead caused by combining floating-point and integer is less than 5%.

Implemented in ST's general-purpose CMOS technology, the design can operate at a frequency of 1.35GHz, while 667MHz can be achieved in low-power CMOS. Considering that the entire datapath is partitioned in just three pipeline stages, and the fact that the design is intended for use in the low-power domain, these frequencies are adequate. They are in fact competitive with current technology low-power floating-point units. Post-layout estimates indicate that the required area of a low-power implementation can be as small as 0.04mm^2. Power consumption is on the order of several milliwatts. Strengthened by the fact that clock gating could reduce power consumption even further, we think that a shared floating-point and integer architecture is a good choice for signal processing in low-power embedded systems.

Categories and Subject Descriptors: B.2.m [**Arithmetic and Logic Structures**]: Miscellaneous; C.5.4 [**Computer System Implementation**]: VLSI Systems

General Terms: Design, Performance

Additional Key Words and Phrases: Floating-point, integer, datapath, fused multiply-add (fma), multiply-accumulate (mac), low-power, embedded systems, area, digital signal processing, pipeline

Authors' address: T. M. Bruintjes, K. H. G. Walters, S. H. Gerez, B. Molenkamp, and G. J. M. Smit, Computer Architecture for Embedded Systems, Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente; email: t.m.bruintjes@utwente.nl.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 1544-3566/2012/01-ART41 $10.00
DOI 10.1145/2086696.2086720 http://doi.acm.org/10.1145/2086696.2086720

ACM Reference Format:
Bruintjes, T. M., Walters, K. H. G., Gerez, S. H., Molenkamp, B., and Smit, G. J. M. 2012. Sabrewing: A lightweight architecture for combined floating-point and integer arithmetic. ACM Trans. Architec. Code Optim. 8, 4, Article 41 (January 2012), 22 pages.
DOI = 10.1145/2086696.2086720 http://doi.acm.org/10.1145/2086696.2086720

1. INTRODUCTION

One of the trends in state-of-the-art energy-efficient embedded hardware is the multi-processor system-on-chip (MPSoC), fabricated in low-power technology. Such platforms (e.g., Blake et al. [2009], Walters et al. [2011], and ter Braak et al. [2010]) often comprise a small RISC processor, several 'number crunchers' (e.g., digital signal processors) and a network-on-chip (NoC) for energy-lean on-chip communication. These heterogeneous MPSoCs are versatile and efficient, yet their support for floating-point operations is weak or completely lacking. In this field the physical properties (area and power) of floating-point units are difficult to justify. Floating-point is therefore often substituted by fixed-point or emulated in software. However, using fixed-point means compromising dynamic range and software emulation is far from satisfactory in terms of performance.

When fixed-point numbers do not provide sufficient dynamic range and floating-point emulation is too slow, a floating-point unit could be integrated in the MPSoC despite its large area and high power consumption. Besides the obvious drawbacks, it is very likely that this unit will remain idle for long periods of time. During idle cycles a considerable amount of resources are not being used effectively. Because the basic operators in floating-point units are the same as the ones in integer ALUs, the floating-point hardware could in essence be used to perform integer operations. The ability to schedule the floating-point unit for integer instructions would boost the integer performance and increase the overall utilization of the hardware. A combined integer and floating-point architecture does not necessarily have to replace its existing counterparts. Dedicated architectures will always be slightly more energy and area efficient when fully utilized. However, when both floating-point and integer arithmetic is required and the hardware is not fully utilized, an architecture that combines both will be advantageous for engineers that have to deal with tight area and energy constraints.

Bringing floating-point arithmetic, merged with integer functionality, to the embedded hardware domain presents some challenges. First of all the design needs to be lightweight (both small and low-power), which is a rarely-seen property in floating-point hardware. Second, even though the basic components in floating-point and integer units are almost identical, actually sharing them is not trivial. To address these issues we present Sabrewing[1], a lightweight but powerful architecture that combines floating-point fused multiply-add (FMA) [Montoye et al. 1990] with integer multiply-accumulate. Multiply-add ($A \times B + C$) instructions are particularly interesting for digital signal processing (DSP). Not only do they increase performance, they also produce more accurate results due to the elimination of an intermediate rounding operation. Another characteristic property of Sabrewing is the small number of pipeline stages. A typical floating-point datapath is deeply pipelined to achieve high throughput. However, experience has taught that designing efficient compilers for deep pipelines is difficult [Hennessy and Patterson 2006], especially if the datapath is to be shared between floating-point and integer instructions. For this reason, the depth of the pipeline is kept to three stages, while each individual stage is delay optimized in order to minimize the impact this has on the overall performance of the design.

[1]Sabrewings are hummingbirds, small and light birds that symbolize the properties of our architecture.

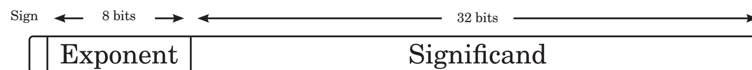


Fig. 1. Sabrewing (41-bit, bias-127) floating-point format.

A fully synthesizable VHDL description of Sabrewing is freely available for download under BSD license.[2] The VHDL is parametrized such that the design can be configured at design time. Besides choosing the number of bits used to represent floating-point and integer numbers, users are able to enable/disable full IEEE-754 support for infinity and NaN by simply specifying this in a configuration file. The only restriction of the design is that that the number of bits used for the significand should match the numbers of bits used for the integers. In the default configuration 32 bits are used for the significands and integers, eight bits for the exponents and infinity and NaN are disabled for reasons that will be explained later.

The remainder of this article is organized as follows. Section 2 discusses the high-level design considerations for a datapath that combines integer with floating-point arithmetic. An overview of previous work related to this subject is presented subsequently. Then Section 4 explains the design principles needed for an efficient FMA datapath, followed by a short analysis of what integer functionality can be migrated to this datapath. The actual Sabrewing architecture is presented in Section 6 and the article is concluded after an evaluation of the physical properties of a 65nm CMOS implementation of Sabrewing in Sections 7, 8 and 9.

Note that throughout the article, N_{sig} indicates the significand of floating-point operand N and N_{exp} the exponent.

2. ARCHITECTURAL DESIGN CONSIDERATIONS

The two most influential design aspects of a processor's execution unit are the way numbers are represented and what operations can be performed on these numbers. We first cover these two aspects before discussing any details of merging floating-point and integer logic.

2.1. Number Representation

In the default configuration, Sabrewing performs operations on 41-bit floating-point numbers. As shown in Figure 1, the format closely resembles IEEE-754 single precision [IEEE 2008], but uses 32 bits for the significand rather than 23. In this configuration, integer arithmetic is based on 32-bit two's complement notation. The reason for this combination is that 32 bits are commonly used to represent integers in existing hardware and software. In addition, our philosophy is that since 32-bit arithmetic components are needed for a full 32-bit integer datapath, we could just as well use that processing power for additional floating-point precision.

The IEEE formats include reserved representations for infinity, NaN, zero and subnormal numbers. In the same default configuration, Sabrewing does not support infinity, NaN and subnormals. Such deviations are not uncommon [Mueller et al. 2005; Texas Instruments 2011; Analog Devices 2011]. Besides the obvious performance gained by reducing the design's complexity, certain applications also benefit from not having to deal with the special cases defined in IEEE-754. Infinity, for example, behaves substantially different from the maximum representable number. A digital filter is much easier to implement in floating-point when the input simply saturates (interpreting infinity as a large number) rather than needing to deal with infinity in a special clause. The outcome of the filter will not change. For this reason, NaN and infinity are by

[2]http://caes.ewi.utwente.nl/sabrewing.

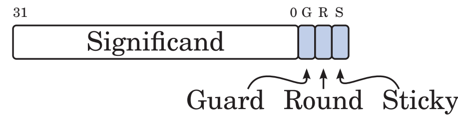


Fig. 2. Guard, round and sticky-bit determine the round direction.

default disabled, but can be enabled at design time. Subnormal support is a different matter. Because supporting subnormal numbers has an enormous impact on the area of Sabrewing, we feel it is justified not to implement them at all. In effect, these deviations from the standard mean that subnormal numbers are forced to zero, infinity becomes a very large number and NaN is not recognized (and also never produced). Numerically, the proposed format behaves the same as single precision, but with more precision. In Section 7 we will discus the impact of full IEEE-754 support on the performance and area of Sabrewing.

2.1.1. Rounding. The reballotted IEEE-754 (2008) standard defines five rounding modes. Due to very limited use and to reduce complexity, it was decided to support only the original four IEEE-754 (1985) rounding modes in Sabrewing. This means round to nearest ties away from zero is not available. Rounding is implemented using a *guard*, *round* and *sticky-bit*, as shown in Figure 2. These additional bits indicate what the rounding direction should be after the result has been normalized. The guard and round bits are basically a 2-bit extension of the datapath. The sticky-bit acts as a control-bit to indicate whether or not the result is inexact (i.e., 1's are lost due to shifting or truncation) and does not participate in the floating-point operation itself.

2.2. Instructions

The need for floating-point signal processing in energy-efficient embedded systems has been the main driver for the development of Sabrewing. Because many DSP applications (e.g., driver Fast Fourier Transform, discrete cosine transform or digital filters such as FIR and IIR) greatly benefit from multiply-add operations, the main class of instructions implemented by Sabrewing are multiply-additions. Despite the fact that the majority of DSP algorithms is based on multiply-add, it should of course be possible to execute separate multiply and add instructions as well. These instructions are available as derivatives of multiply-add ($A\times B+0$ and $A\times 1+C$ respectively), which helps keep the datapath structure as regular as possible and ultimately benefits the multiply-add performance and area of Sabrewing. Input operands can be positive and negative, which provides the means to perform subtraction and multiply-subtract as well. In addition, the full range of comparisons ($>$, $<$ and $=$) is implemented, supporting both integer and floating-point numbers. Finally, two directions of signed integer shifts are supported, because they are relatively easy to extract from a floating-point datapath. Other basic operations such as division and square root are not directly implemented in hardware because they would defeat the purpose of a lightweight floating-point solution. Moreover, they can be provided relatively easily as software routines, using FMA instructions [Markstein 2004]. Considering that these instructions are rarely used, software emulation is feasible, as demonstrated in the Intel Itanium [Robison 2005; Cornea et al. 2003] and CELL processor [Mueller et al. 2005]. This subject is however not part of this work. An overview of the instruction support with corresponding latencies is shown in Table I. Note that every cycle a new instruction can be issued and that all floating-point instructions require three cycles to complete. All integer instruction need two cycles. Even though this is a suboptimal solution from a hardware perspective (Section 6), the latency consistency is convenient for the compiler/scheduler.

Table I. Sabrewing Instructions

Operation	Operands		latency (# cycles) integer	floating-point
Multiply-Add	{A,B,C}	: floating-point or integer	2	3
Compare (>,=,<)	{A,B}	: floating-point or integer	2	3
Shift (≫,≪)	{A,n}	: integer and natural	2	n/a
Multiply	{A,B,0}	: floating-point or integer	2	3
Add	{A,1,C}	: floating-point or integer	2	3

3. RELATED WORK

The FMA concept is slowly gaining popularity [Cornea et al. 2003; Mueller et al. 2005; Wittenbrink et al. 2011; Butler et al. 2011]. While FMA was initially an extension of the existing floating-point datapaths that implemented multiplication and addition separately, we see that the conventional floating-point adders and multipliers are being replaced with FMA units in more recent processors. While believe that now x86 general-purpose processors [Butler et al. 2011] are making the transition to FMA, it will become even more important. IBM became the pioneer of FMA [Montoye et al. 1990] when they introduced the System/6000. Successive publications mostly describe variations of the System/6000's FMA datapath. The work presented in Quinnell et al. [2007] for example, shows how the overall latency of FMA instructions can be reduced by distinguishing multiple data flows for addition, based on the exponent difference of the input operands. Other examples include Jessani and Putrino [1998] where the datapath is modified such that multiple passes allow double precision instructions to be executed on a single precision datapath and Huang et al. [2007] where a double precision FMA datapath is modified to support two-way single precision SIMD instructions. To our knowledge FMA has not yet found its way to the low-power (embedded/DSP) processors.

Merging floating-point and integer hardware is seldom seen, yet the idea is not entirely unexplored. Thapliyal et al. [2006] propose to replace the standard 18×18-bit multiplier in FPGAs with a 24×24-bit multiplier, which could then be used for single precision floating-point multiplication. However, with this approach the multiplier is assigned to integer or floating-point duty at design time, not both. The actual potential of combined integer and floating-point hardware was first recognized by Palacharla and Smith [1995]. They evaluate its feasibility and indicate that about 40% of the integer operations can be migrated to a typical floating-point datapath. In response to this work, Solihin and colleagues present a reconfigurable adder that can switch between integer and floating-point functionality at runtime [Lavenier et al. 2000; Solihin et al. 2001]. Some examples of merged floating-point and integer execution hardware can be found in the industry as well. Mueller et al. [2005] hints that the CELL processor is allegedly capable of performing integer multiply-additions in the synergistic processing element's single precision floating-point cores. Unfortunately no details are discussed except for the fact that such operations require preprocessing, resulting in a latency penalty. In the SPARC V2 architecture, the multiplier of the floating-point execution datapath has been isolated to serve as the main multiplication element for both integers and floating-point significands. However, the majority of the SPARC's floating-point hardware is still used exclusively for floating-point purpose. Architectures that do combine floating-point and integer hardware are still immature and this shows there is need further research to optimize and explore the potential benefits.

The main contributions of this work are the advances in the concept of merging floating-point and integer by applying the technique in a new scope (FMA) and at a much larger scale. The rationale is to bring floating-point hardware support to small, low-cost and energy efficient embedded systems. With Sabrewing we demonstrate the feasibility of such a lightweight floating-point solution.

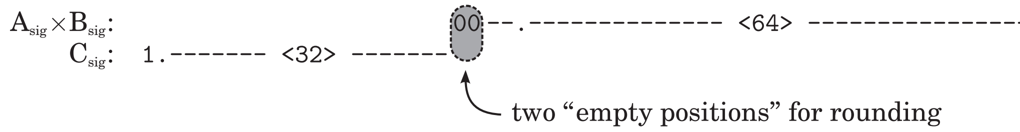


Fig. 3. Positioning of $A_{sig} \times B_{sig}$ and C_{sig} prior to alignment.

4. FUSED MULTIPLY-ADD

When limited to few pipeline stages (in this case three), designing a reasonably performing FMA datapath is not trivial. This section shows a number of optimizations used in the Sabrewing architecture to achieve good performance without compromising the pipeline constraint and the area limitations of embedded systems. Because we feel there is a serious lack of literature describing FMA in detail, we go into much greater depth than most articles would. The principles discussed here are generally applicable to any sign-magnitude floating-point format. However, all examples assume the 41-bit representation presented in Section 2 (Figure 1).

4.1. Alignment

Floating-point addition requires that the smallest operand is shifted to the right during alignment. However, in case of FMA ($A \times B + C$) the product ($A \times B$) first needs to be computed before the smallest operand can be identified, which causes a major bottleneck. If C_{sig} is positioned entirely in front of the product, as shown in Figure 3 [Jessani and Putrino 1998], alignment can be implemented in parallel with multiplication. By positioning C_{sig} entirely in front of $A_{sig} \times B_{sig}$, and imagining the radix points to be fixed, C_{sig} will always be the largest operand and therefore $A_{sig} \times B_{sig}$ does not have to be evaluated first. Although this principle can be found in the literature [Jessani and Putrino 1998; Schwarz 2006; Mueller et al. 2005], the details are lacking.

For a basic floating-point addition ($P+Q$), the alignment shift is given by the absolute difference between the exponents: $|P_{exp} - Q_{exp}|$. However, multiplying two 33-bit significands (32 bits and the *hidden-bit*) yields a 66-bit significand, with two bits instead of one bit to the left of the binary point (Figure 3). The FMA exponent datapath must take this into account by incrementing the value of the intermediate exponent. To place C_{sig} entirely in front of the product, the exponent also needs to be adjusted by the amount of bits used for the significand (32 in this case) plus the hidden-bit. Finally, as shown in Figure 3, two more positions may have to be taken into account for rounding. During fused multiply-add operations, the guard and round bits can be placed between C_{sig} and $A_{sig} \times B_{sig}$. This has the advantage of no overflowing significands. The above-mentioned adjustments can be combined into a single implementation-specific offset. This offset should also include the bias of the IEEE-754 notation, which normally accumulates during the addition of A_{exp} and B_{exp}. In this particular case, the alignment shift is:

$$\begin{aligned} \text{Shift} &= A_{exp} + B_{exp} - C_{exp} + 36 - 127 \\ &= A_{exp} + B_{exp} - C_{exp} - 91. \end{aligned} \quad (1)$$

And the corresponding exponent is:

$$\text{Exponent} = A_{exp} + B_{exp} - 91. \quad (2)$$

4.1.1. Sticky-Bit. During alignment, precision may be lost due to the shift. For rounding purpose, a *sticky-bit* keeps track of precision loss (Section 2.1.1). Whenever a `1` is shifted out of range, the sticky-bit is asserted to `1` and stays `1` during the entire computation.

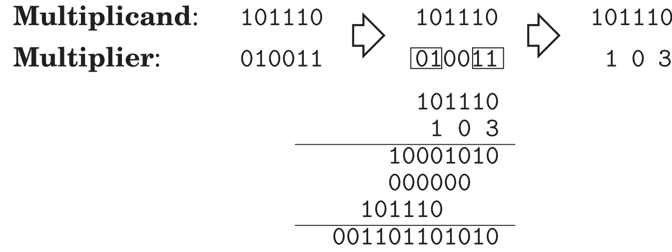


Fig. 4. Radix-4 multiplication.

4.2. Multiplication

Although multiplication can be performed in parallel with alignment, the multiplier design itself continues to be decisive for the latency and area of the datapath. Sabrewing uses a combination of modified *Booth encoding* [Booth 1951] and a *Wallace tree* structure [Wallace 1964] for its single-cycle multiplier. Two steps can be distinguished in single-cycle multiplication. Generating partial products and summing these partial products. Booth encoding helps reduce the number of partial products while a Wallace tree efficiently accumulates them.

4.2.1. Booth Encoding. Booth [1951] states that M×`00111110` ($M\times(2^5+2^4+2^3+2^2+2^1)$) equals M×`010000(-1)0` ($M\times(2^6-2^1)$). The number of additions in the latter is obviously less, however, this basic Booth encoding only applies to multi-cycle multiplication. For single-cycle multiplication, Booth encoding only works when radix-4 multiplication [Parhami 2000] is used. In radix-2 multiplication, the multiplier bits generate partial products that are either 0× or 1× multiples of the multiplicand. In radix-4 multiplication, two multiplier bits generate partial products that are 0×, 1×, 2× or 3× multiples of the multiplicand as shown in Figure 4.

Note that 3× is not a nice regular multiple (i.e., cannot be obtained by merely shifting). However, with Booth encoding this multiple can be obtained by (4×)-(1×), which is again regular. Radix-4 multiplication reduces the number of needed adders from $n + 1$ to $\lceil(n + 1)/2\rceil$. Hence, by using modified Booth encoding the area of the multiplier is reduced significantly without negatively affecting its latency. In addition Booth encoding enables us to support both two's complement and unsigned input in the multiplier.

4.2.2. Wallace Tree Structures. Carry propagation in adders is among the longest latencies found in most arithmetic circuits. Therefore, carry-save adders (CSA) are used to accumulate the partial products. CSAs are a special type of adder that immediately output their carries instead of propagating them. An n-bit CSA yields an n-bit sum and an n-bit carry that can be converted to a regular n+1 bit binary number by adding the sum and carry as usual. This seemingly cumbersome way of adding numbers is very efficient for accumulation. Instead of propagating m carries for a series of m additions, only one carry propagation is required at the bottom of the multiplier array.

By placing the CSAs in a regular tree structure, a Wallace tree is obtained (Figure 5). As opposed to a standard multiplier array, the depth of a tree ($\lceil\log_2(m)\rceil$) is much smaller and therefore a much better latency can be achieved.

4.3. Addition

Addition and multiplication are elegantly *fused* when an additional CSA is inserted in the multiplier tree, as shown in Figure 5. However, because IEEE-754 floating-point representation is sign-magnitude, this only works for positive operands. To support

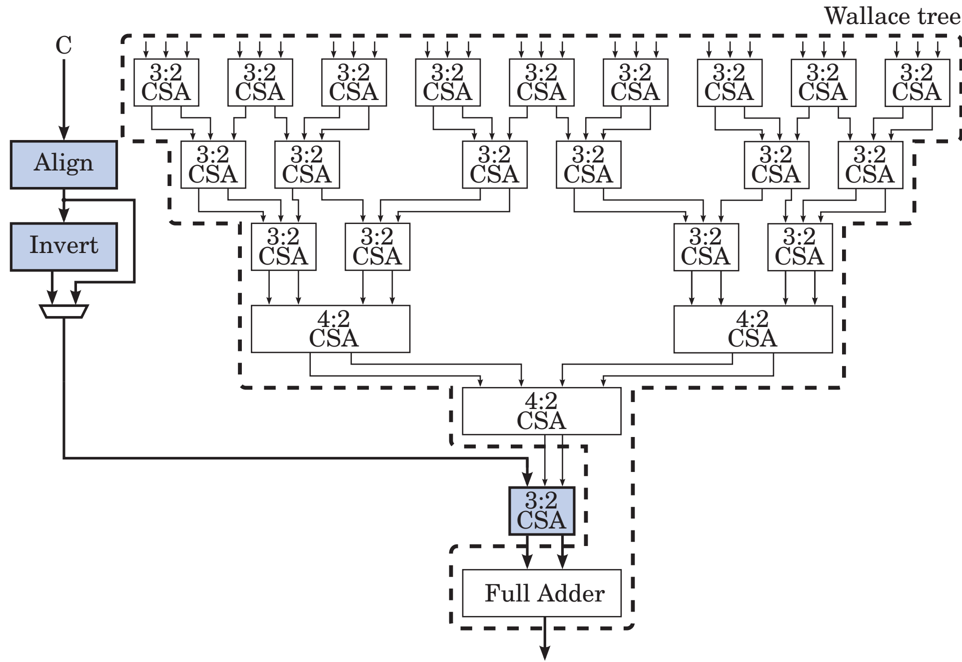


Fig. 5. Addition merged with the (Wallace) multiplier tree.

effective subtraction (e.g., $-A\times B+C$ or $A\times B-C$), additional measures have to be taken. Internally converting all input to two's complement notation will work but is not efficient. *End-around carry addition* [Vassiliadis et al. 1989] requires much less hardware and does not contribute to the critical path since only C needs to be converted to one's complement, which overlaps with the delay of the multiplier.

The principle of end-around carry addition applies to the magnitude of the operands ($|A\times B|$ and $|C|$); the sign bits are merely used for control. If $|A\times B| > |C|$, the magnitude is given by: $|R| = (|A\times B| - |C|) = |A\times B| + \overline{|C|} + 1$. This means that $|C|$ is inverted (converted to one's complement) before being added to $|A\times B|$, and after addition the result must be incremented. An important property of end-around carry addition is that the carry out (C_{out}) is always 1 in this case. If $|A\times B| < |C|$, the resulting magnitude is obtained by: $|R| = \overline{|A\times B| + \overline{|C|} + 0}$, and the carry out will always be 0.

In a more generalized form, the magnitude can be found by: $\Sigma = |A\times B| + |C|^* + C_{out}$, where $|C|^*$ equals $|C|$ for effective addition and $\overline{|C|}$ for effective subtraction. The sign bits combined with C_{out} provides enough information to determine if the result needs to be re-complemented after addition: $\Delta = \overline{C_{out}} \wedge$ "effective operation". When Δ is 1, the result is re-complemented, otherwise it remains unchanged: $|R| = \Sigma \oplus \Delta$.

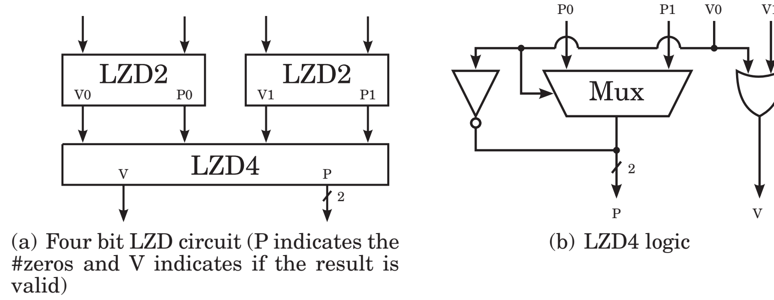
The sign-bit is defined as:

$$R_{sign} = (A_{sign} \oplus B_{sign}) \oplus \text{effective operation} \wedge \overline{C_{out}} \tag{3}$$

except when the result is exactly zero [Bruintjes 2011].

4.4. Normalization

After addition, the intermediate result is most likely not normalized. A normalized result is obtained by counting the number of leading zeros in the significand and shifting it to the left by that amount. Typically the number of zeros can only be counted



(a) Four bit LZD circuit (P indicates the #zeros and V indicates if the result is valid)

(b) LZD4 logic

Fig. 6. Leading zero detection.

after the addition has been performed, resulting in another bottleneck. To tackle this problem, *leading zero anticipation* (LZA) is used to predict the number of leading zeros.

4.4.1. Leading Zero Anticipation. There are different techniques to predict the number of leading zeros [Schmookler and Nowka 2001]. Some aim for a prediction that is always exact, others settle for a close approximation. LZA describes a boolean relation between the input of the adder (A,B) and a string f of indicators: $0^k1^*x^*$ (where $k \geq 0$ and x is 1 or 0) in which position f_i indicates whether that position will be a 1 or 0. The LZA [Schwarz 2006] used in Sabrewing predicts the leading zeros with one position uncertainty. The value of each position is given by:

$$f_i = \begin{cases} \overline{T_0}T_1 & (i = 0) \\ T_{i-1}(G_i\overline{Z_{i+1}} \vee Z_i\overline{G_{i+1}}) \vee \overline{T_{i-1}}(Z_i\overline{Z_{i+1}} \vee G_i\overline{G_{i+1}}), & (i > 0) \end{cases} \quad (4)$$

where

$$\begin{aligned} T_i &= A_i \oplus B_i \\ G_i &= A_i\ B_i \\ Z_i &= \overline{A_i}\ \overline{B_i}. \end{aligned}$$

When position f_i is the least significant bit (LSB), position f_{i+1} represents the carry-in of the adder. This approximate LZA requires much less hardware than exact LZA [Schmookler and Nowka 2001]. The downside is that a correction may be needed.

Handling the LZA misprediction. The position of the leading 1 according to Equation (4) is either exact or it is located one position to the left of the actual leading 1. Since the error is so consistent, it is easily corrected. In case of a misprediction, we can simply shift the significand one more position to the left.

4.4.2. Leading Zero Detection. The prediction obtained from LZA needs to be encoded into a binary number to drive a left-shifter for actual normalization. Two different techniques to count the number of leading zeros exist. One is based on monotonic string encoding, the other on a hierarchical counting tree. The monotonic string method [Hayes et al. 1985] is typically faster while the counting tree has the advantage in terms of area and energy-efficiency. Because LZA takes leading zero detection (LZD) off the critical path, counting trees are attractive for an approach like Sabrewing.

A well known LZD circuit is the one from Oklobdzija [1994]. This tree is built up from small 2-bit LZD circuits (Figure 6(a)) that are combined into a 4-bit LZD circuit (Figure 6(b)), that can be used to create 8-bit LZDs and so on. The technique does not directly apply to input that is not a power of two, implying that a 128-bit LZD will be needed to count the number of leading zeros in the 101-bit LZA result (Figure 8).

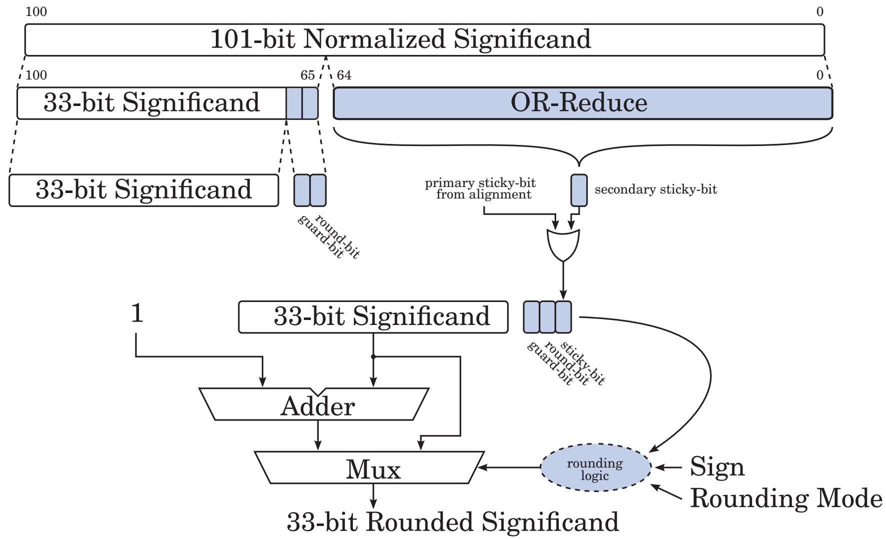


Fig. 7. IEEE-754 compliant fused multiply-add rounding.

ALGORITHM 1: IEEE-754 round to nearest ties to even.

Input: guard-bit $\boldsymbol{G}$, round-bit $\boldsymbol{R}$, sticky-bit $\boldsymbol{S}$, the normalized result truncated up to the original input size plus the hidden-bit $Significand_{normalized}$, and the least significand bit of the normalized result $\boldsymbol{LSB}$.
Output: The rounded significand $Significand_{rounded}$.
if $(\boldsymbol{G} = 0)$ **then**
    $Significand_{rounded} \leftarrow Significand_{normalized}$
else if $(\boldsymbol{R} = 1 \vee \boldsymbol{S} = 1)$ **then**
    $Significand_{rounded} \leftarrow Significand_{normalized} + 1$
else if $(\boldsymbol{LSB} = 0)$ **then**
    $Significand_{rounded} \leftarrow Significand_{normalized}$
else
    $Significand_{rounded} \leftarrow Significand_{normalized} + 1$
end

Because this is not very area efficient, a modified LZD algorithm that performs better on input that is not a power of two has been devised for Sabrewing [Bruintjes 2011].

Sticky-Bit. In order to reduce the obtained 101-bit significand to the original input size (32 bits), the 66 LSBs will have to be truncated (the MSB will become a hidden-bit and the guard and round-bit will be removed after rounding). For similar reasons as explained in Section 4.1.1, these bits are OR-reduced into the sticky-bit to keep track of any precision that is lost.

4.5. Rounding

After truncation, the significand is either incremented (rounded up), or remains unchanged (rounded down) based on the values of the guard, round and sticky-bit. Once the sticky-bit has been determined, IEEE-754 compliant rounding requires little effort. Several pattern matching algorithms [Woo-Chan et al. 1996] such as the one for *round to nearest even*, listed in Algorithm 1, can be used to determine the rounding direction. An overview of the rounding process is schematically depicted in Figure 7.

4.6. Pipelining

Based on a combination of the design principles described in this section, fused multiply-add can be implemented in a fairly well balanced three stage pipeline (Figure 8).

5. INTEGER OPERATIONS AND FLOATING-POINT HARDWARE

A number of fundamental arithmetic operations reside in every floating-point datapath: shifts, addition, subtraction and multiplication. These are the integer operations that can be incorporated in the floating-point datapath, without noteworthy increase of silicon area or latency. The FMA datapath following from Section 4 is now reviewed to show which parts lend themselves to be modified for integer purpose.

Alignment requires several small adders/subtractors (Equation (1)) to determine the shift, and a right-shifter to actually align the operands. The adders and subtractors (and the entire exponent datapath for that matter) are not particularly useful for integer operations because they accommodate only 9-bit input (8-bit exponent plus one overflow bit). The shifter can be used for a shift-right instruction, although integers benefit most from an arithmetic shift while floating-point alignment is based on a logical shift.

The Booth encoded multiplier is directly usable for two's complement integer operation. Both full adders can in principle also directly be used for two's complement additions. However, full utilization of the multiply-accumulate potential of the FMA datapath requires a number of modifications.

Major components in the normalization circuit are the leading zero anticipator, the leading zero detector, and a left-shifter. The function of leading zero anticipation is too specific to be of any use for operations other than normalizing floating-point data. Leading zero detection is also not particularly useful outside the floating-point context, although some fixed-point instruction sets do include a leading zero count instruction. The normalization shifter on the other hand, provides the means to implement a left oriented shift. Since arithmetic shift-left is exactly the same as logical shift-left, no complications arise here. The rounding circuitry includes an incrementor, but since this does not provide any additional functionality over addition, re-using it will not yield a direct advantage worth the effort.

Although this synopsis may give the impression that merging integer and floating-point datapaths is trivial, this is certainly not the case. For example, the basic integer instructions all require at most one out of three pipeline stages. Multiply-accumulate, however, requires both the first and second stage. Performance-wise it would be best to implement the instructions with variable latency. However, this will lead to complicated instruction scheduling due to *pipeline-hazards*. For consistency we have chosen to implement all floating-point instructions in three clock cycles and integers instructions in two cycles (Section 2). A pipeline flush is required when changing from floating-point to integer operation. The penalty is however just one clock cycle because the pipeline is three stages deep. The next section describes the Sabrewing architecture, our approach to combine integer and floating-point hardware for low-power DSP.

6. THE SABREWING ARCHITECTURE

The entire (default) Sabrewing datapath (configuration) is depicted schematically in Figure 8. To give an impression of the achieved level of reuse, all (partially) shared components are colored.

I/O Interfaces. The input of the datapath consists of a 5-bit opcode and three 64-bit operands. Each 64-bit input is subdivided into two 32-bit words ("left" and "right"), accommodating the floating-point sign and exponent, and significand/integer

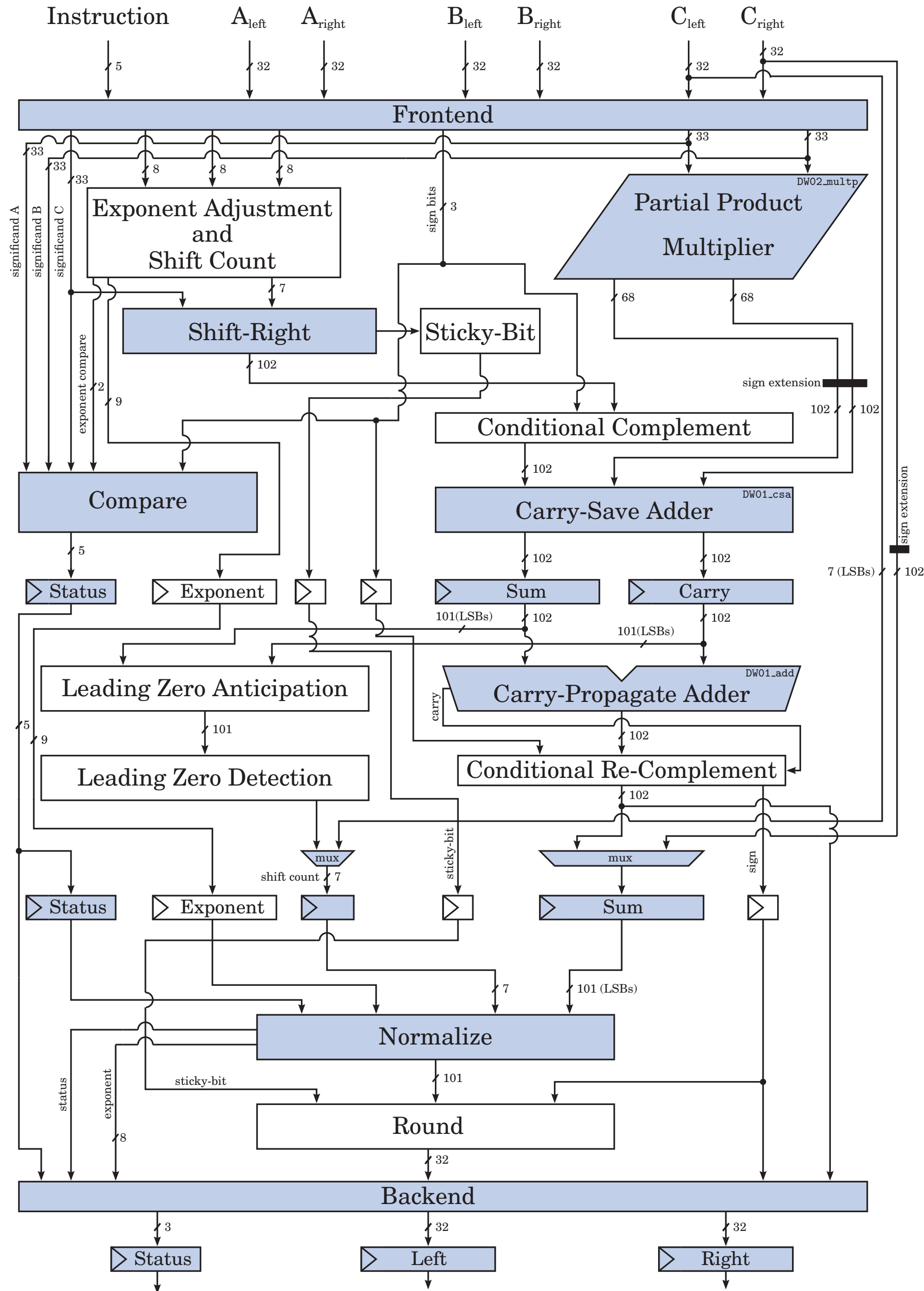


Fig. 8. Default Sabrewing datapath overview (colored blocks indicate shared hardware).

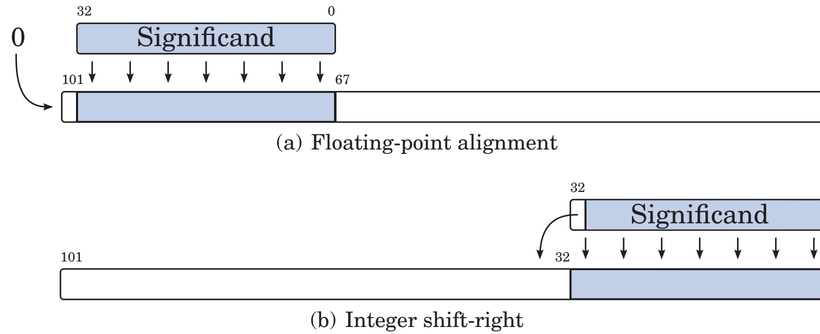


Fig. 9. Input mapping to right-shifter.

respectively. A 5-bit opcode is sufficient to encode all instructions from Section I and possible future extensions. The reason for making all data interfaces 32-bit (even though the exponent and sign require only 9 bits) is that this is convenient for memory access and it allows the datapath to be integrated in existing infrastructures more easily. The output of the datapath consists of three status bits acting as the exception flags, and two 32-bit data ports. In case of integer multiplication or multiply-accumulate, the higher order bits of the 64-bit result are mapped to “left” and the lower order bits to “right”.

Frontend. The frontend performs three preparatory tasks. First each opcode is translated into control signals that configure the datapath. Second, all floating-point significands are extended with an MSB of `1` to make the *hidden-bit* explicit, while integers are sign extended to match the width of the new 33-bit significand. Last, each operand is checked for zero, and when the Sabrewing is configured to do so, also for infinity and NaN.

Exponent Datapath and Alignment Shifter. Equations (1) and (2) show how the shift for alignment and corresponding exponent can be found. An actual implementation requires one adder and two subtractors. The exponent is manipulated with 9-bit arithmetic, such that the intermediate results will never overflow. Underflow can not be prevented as easily. In order to detect (temporary) underflow, a 9-bit comparator is used to determine if the intermediate exponent is smaller than the bias. If this is the case then the exponent will underflow, which is registered for later use. Actually raising the exception flags is postponed until after rounding, because normalization could undo overflow or underflow. To save area, the comparator is multiplexed with primary input A_{left} and B_{left} to facilitate floating-point compares.

The input of the alignment shifter is extended by one bit, of which the value depends on the type of input: `0` for floating-point and `1` for integer. This allows the shifter to be implemented as an arithmetic shift-right, such that it can serve both integer and floating-point operations. To aid datapath regularity, the input to the shifter is mapped as shown in Figure 9(a) and 9(b). By mapping integer input to the LSBs, the shifter output can be routed through the adder $(A \ll n) = 0 \times 0 + (A \ll n)$ to match the two cycle latency of the other integer instructions, without using additional registers. As shown in Figure 8, the output of the adder is sent directly to the backend of the datapath such that the third stage is bypassed for all integer arithmetic and shift-right.

Sticky-Bit. The primary sticky-bit (caused by alignment) is implemented as a 66-bit OR-gate reduction tree, fed with the bits from C that are shifted beyond the range of the datapath.

Comparator. In the comparator we face similar problems as encountered in the shifter. Two's complement comparators do not operate correctly on unsigned operands and vice versa. The solution is again to perform an input extension, such that synthesis tools will infer only one (signed) comparator. Note that this modification only needs to be applied to the significand comparator. Moreover, the full range of compares ($>$, $=$ and $<$) require only implementations of $>$ and $=$, since $<$ can easily be derived.

Multiplication and Addition. The heart of Sabrewing consists of the *partial product multiplier*, a Wallace tree without the final carry-propagate adder; the complementers that invert the addend in case of subtraction; an additional CSA and the final carry-propagate adder.

Modified Booth encoding is used to support both floating-point (unsigned) and integer (signed) multiplication. Although other (possibly more efficient) techniques exist for signed multiplication [Parhami 2000; Sjalander and Larsson-Edefors 2008], Booth has the advantage that it is widely adopted and particularly well supported in the Synopsys design flow. In our prototype (Section 7), Synopsys DesignWare components [Synopsys 2011] were used to implement parts of the core functionality. The DW02_multp IP implements the partial product multiplier, DW01_csa the carry-save adder and DW01_add the main adder. A few complications arise when Booth encoded Wallace trees are used for integer and floating-point multiplication. For FMA, sign suppression needs to be applied to prevent false carry out during end-around carry addition [Schwarz 2006]. Because we have no control over the internals of DW02_multp, the problem was solved otherwise. By explicitly including sign bits during end-around carry addition, we can prevent false carry out from occurring in most cases (not for input that is zero). To share the multiplier between integer and floating-point operands in general, it should be noted that the output of a Booth multiplier is always signed, regardless of how the input is read, and that the output is actually two bits wider than twice the input. Both complications are easily solved by sign extension. Since the aligned C operand including sign-bit is already 102 bits wide, the 68-bit output of the multiplier should be sign extended. End-around carry addition will now work correctly for floating-point, for integers the complementers must be disabled and C needs to be sign extended from 101 to 102 bits.

The partial products, now including the aligned C, are added with DW01_csa of which the carry in is forced to `0` and the carry out can be ignored. Due to inclusion of the sign bits, end-around carry addition does not work when the input is zero. A correct result is again obtained by disabling both complementers when the input is zero. Two sequential adders implement end around carry addition (Figure 10), because both sum and sum+1 (Section 4.3) have to be known. Since the final addition takes place in stage two, a non-critical stage of the pipeline (see Section 7), the adders are placed serially (instead of parallel) for a marginal area improvement.

The modifications needed to incorporate integer multiply-accumulate in a floating-point FMA datapath are actually modest, yet not self-evident. More details can be found in Bruintjes [2011].

Leading Zero Anticipation and Leading Zero Detection. The LZA and LZD hardware is not particularly interesting for integer use. LZA has been implemented exactly as shown in Equation (4) and LZD by a variant of Oklobdzija's counting tree [Oklobdzija 1994], which requires less area and does not affect latency.

Normalize. The actual normalization circuit contains a left-shifter which is useful for integer shift-left instructions. Arithmetic shift-left is exactly the same as logical shift-left, sharing this shifter is straightforward. To obtain a two cycle latency, the sum

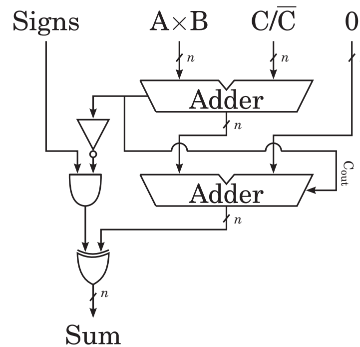


Fig. 10. End-around carry addition.

and shift count registers have been multiplexed rather than the input of the shifter itself.

Round. Floating-point rounding (Figure 7) is implemented in 34-bit arithmetic, according to the pattern matching logic described in Woo-Chan et al. [1996]. The 34 bits prevent the significand from overflowing during the increment. If the carry out is `1`, the overflow is easily corrected by adjusting the exponent and using only the 33 MSBs as the significand. Since the functionality of the 66-bit primary adder supersedes that of a 34-bit incrementor, the incrementor was not modified to support integer operands. Rounding is disabled when the last stage is used for integer shift-left instructions.

Backend. The backend of the datapath performs the final exception checks and formats the output. During a floating-point multiply-add operation, underflow and overflow can occur in multiple places: during alignment, normalization and rounding. It is possible that an underflowed or overflowed intermediate result can perfectly be represented after being normalized or rounded. Because the exponent datapath is implemented with 9-bit arithmetic, overflow occurs only when the rounded result is found to be outside the representable range. The backend then asserts both the status bits and the output to the corresponding IEEE-754 representation. Note that 32-bit integer multiply-accumulate will never overflow due to the exceptionally wide datapath. The most positive result is still well within the 64-bit output range, which also holds for the most negative result. For convenience, the status registers are therefore used to indicate if the integer result requires more than 32 bits. Last, the sign-bit as defined by Equation (3) does not always apply when the result, or one of the operands, is zero, infinity or NaN. The backend performs a correction in these cases.

7. REALIZATION

To show the feasibility and advantages of Sabrewing, a VHDL structural description has been implemented in FPGA and ASIC technology. FPGA mapping was carried out using Synopsys Synplify, targeting a Virtex-5 LX330T device. The ASIC prototype is based on a fairly standard IC design flow, using Synopsys DesignCompiler for synthesis and Cadence Encounter for layout generation. Both the low-power high voltage threshold (LPHVT) and general-purpose standard voltage threshold (GPSVT) libraries from STMicroelectronics' 65nm CMOS technology have been evaluated.

7.1. FPGA

Despite the fact that an FPGA mapping of Sabrewing was never a major objective of this work, it was carried out to verify compatibility with the technology. We encountered

Table II. 200MHz LPHVT (pre-layout) Area Distribution

Component(s)	Area (μm^2)	Percentage (%)
Sticky-Bit OR-Reduction	238	0.7
Backend	381	1.2
Exponent Datapath	432	1.3
Frontend	454	1.4
Comparator	517	1.6
Round	862	2.6
Carry-Save Adder	951	2.9
Leading Zero Detection	965	2.9
Residual Logic	1506	4.5
EAC Recomplement	1726	5.2
Main Adder	1852	5.6
Leading Zero Anticipation	2384	7.2
Alignment Shift & Complement	2723	8.2
Normalize	4193	12.6
Non-Combinatorial (Registers)	4223	12.7
Partial Product Multiplier	9750	29.4
Total	33157	100.0

no problems, although the design flow is restricted to Synopsys tools because of the DesignWare IPs. On a Virtex-5 LX330T, the design operates at 96MHz and occupies less than 2% of the device's resources (3683 LUTs).

7.2. ASIC

Because the potential advantages of Sabrewing matter most in ASIC technology, we present the ASIC implementations more elaborately. Every design is based on the default Sabrewing configuration (see Section 2) which means no support for NaN, infinity and subnormals. Scan-chains are included and automatic retiming is applied to optimize the pipeline balance. Since power consumption is a major concern and limitation for embedded systems, the LPHVT library is the main target technology in this survey.

7.2.1. Timing. The critical path is located in the first stage, rippling through the multiplier as expected. Implemented in LPHVT, the datapath can be clocked up to 667MHz while 1.35GHz is the limit in GPSVT technology. Although such frequencies can be achieved when desired, the clock will most likely be constrained to 200MHz (i.e., the frequency currently achieved in MPSoC DSP-cores [ter Braak et al. 2010] that are candidate for Sabrewing integration). The pipeline is fairly well balanced by itself, however, register retiming still yields an improvement of almost 100MHz in LPHVT and over 200MHz in GPSVT. Implementing IEEE-754 compliant infinity and NaN support (see Section 2) incurs a penalty of just 10MHz, when retiming is not taken into account. Support for subnormal numbers has a much larger impact. Even though subnormal support was not fully implemented like infinity and NaN, our initial evaluations show that at least 90MHz must be traded off to prenormalize the input operands.

7.2.2. Area. The area of the default Sabrewing configuration, fully placed and routed with the LPHVT library, is approximately 0.07mm^2 at 667MHz; or 41K equivalent gates. The GPSVT implementation, running at 1.35GHz, occupies close to 0.08mm^2 (54K gates). However, when the clock frequency is brought back to 200MHz, both the LPHVT and GPSVT areas are just 0.04mm^2 (22K and 20K gates respectively). A detailed pre-layout area distribution of the 200MHz LPHVT implementation is shown in Table II. Roughly the same distribution can be found in the other implementations. The residual logic is a good indication of the overhead caused by merging floating-point with integer arithmetic. Since these area figures are based on pre-layout results, they

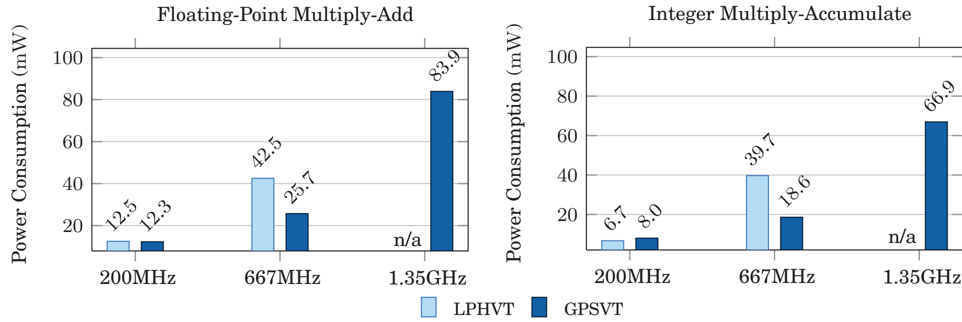


Fig. 11. Power consumption based on a 4-tap FIR filter in floating-point (left) and integer (right.)

should be considered as lower bounds. Supporting infinity and NaN has no notable impact on the area. Subnormal support will increase the area by at least 30%.

7.2.3. Power Consumption. Taking the parasitics of the design layout into account, the power consumption of Sabrewing is summarized in Figure 11. This power is based on a 4-tap FIR filter under normal conditions (1.2/1.1V operating voltage and 25°C temperature). An input signal with a large amplitude was chosen to cause high internal switching activity, but more importantly, to ensure that the dynamic range of floating-point is being used. The filtering was performed both in floating-point and integer (fixed-point) arithmetic. As expected, the power consumption of the floating-point filter is higher. Close inspection shows that high clock frequencies result in power inefficiency in the integer path. Integer operations consume almost the same amount of power as floating-point operations at 667MHz because of the aggressive synthesis optimizations needed to satisfy the timing constraint. At this frequency there is also a large difference between LPHVT and GPSVT. The 667MHz frequency pushes the design to its limits in LPHVT, which means many more transistors are required. In addition the voltage threshold of GPSVT is 0.1V lower. Because of this observation one might be tempted to use GPSVT. However, leakage is several orders of magnitude higher in GPSVT. At 200MHz, the leakage is 117nW in LPHVT while it is 286μW in GPSVT. The total power consumption is almost the same for GPSVT and LPHVT. Leakage then becomes an important factor, especially in combination with clock gating.

Clock Gating. To reduce/eliminate power inefficiency, clock gating can be applied to the datapath. In the 667MHz low-power design, automatically inserted clock gates save approximately 4.5mW in the FMA case and a little over 3mW for integer MAC. We expect that much better results can be obtained by manual clock gate insertion, or by providing the EDA tools optimization directions based on switching activity.

8. EVALUATION AND FUTURE WORK

A realization of the Sabrewing architecture has demonstrated that it has attractive physical properties. This section will provide an indication of how it compares to similar floating-point solutions. At the end of this section we also discuss possible improvements and further research planned for Sabrewing.

8.1. Comparable Low-Power Floating-Point DSP Solutions

It is difficult to make a fair assessment of our new architecture because it is still somewhat unique (Section 3). However, to put Sabrewing in the right perspective, some properties of commercially available low-power floating-point DSPs have been

Table III. Low-Power Floating-Point DSP Overview

	TI TMS320C674x	AD SHARC 2147x
Clock Frequency	200–456MHz	200–266MHz
Floating-Point Support	IEEE SP/(DP) MUL and ADD *- No subnormals* *- Four rounding modes* *- DP not native*	IEEE SP/40-bit MUL and ADD *- No subnormals* *- Only two rounding modes*
Pipelining	Two stage SP MUL and ADD → *four-cycle MAC*	Single stage MUL and ADD → *two-cycle MAC*
Power Consumption	141.2mW at 200MHz 187.8mW at 266MHz 377.8mW at 456MHz	235.9mW at 200MHz 310.0mW at 266MHz

listed in Table III. The low-power DSPs of Texas Instruments [Texas Instruments 2011] and Analog Devices [Analog Devices 2011] are believed to be the current state-of-the-art.

Although both Texas Instruments (TI) and Analog Devices (AD) claim to be IEEE-754 compliant, thorough inspection of their documentation reveals that this is not entirely true. Like Sabrewing, some shortcuts were taken in both architectures. Neither support subnormal numbers and SHARC only supports half of the rounding modes. Concerning the physical properties, Sabrewing appears to be a step ahead of both DSPs. Either measured in raw frequency or instructions per cycle, both TI and AD are outperformed by Sabrewing. And even though it is impossible to isolate the power consumption of just the arithmetic datapath in these DSPs, their total power consumption shows that Sabrewing is at least within 10% thereof. Unfortunately no information about area is provided. This is why we implemented a reference design to do baseline comparisons.

8.1.1. A Baseline Conventional Approach. Our baseline circuit offers functionality comparable to Sabrewing, but is implemented in a conventional way (no sharing of integer and floating-point hardware). The integer datapath is designed by straightforward VHDL coding and the floating-point datapath is based on a synthesizable version [Bishop 2011] of the new IEEE-1076 VHDL floating-point support [IEEE 2009]. Because the same 65nm process technology and tool flow are used, no assumptions have to be made and a direct comparison is possible. There is however one caveat. Although Bishop [2011] provides very similar floating-point multiply-add functionality, the results it produces do not adhere to IEEE-754 (about 8% of the tests in IBM's FPGen [IBM Haifa Research Lab 2011] fail) and only one rounding mode can be chosen per instance (round to nearest was chosen here). This implementation is not ideal but unfortunately the best that is currently available to us. Sabrewing does produce IEEE-compliant results and is therefore a much more complete design. This has notable impact on its performance and to an even greater degree its area. That is why we expect that the improvements made by Sabrewing are better than portrayed by this comparison.

8.2. Performance Area and Energy Comparison

Compared to the baseline, Sabrewing shows improvement in all areas: clock frequency, silicon area, and power consumption. Table IV shows the differences in detail. The performance difference can be explained by the many delay optimizations applied to the Sabrewing datapath (Section 4). The difference in area is mostly a result of eliminating integer specific logic. The reduced power consumption results mostly from a lower transistor count. No clock gating was applied for this comparison. Note that although the area of Sabrewing is larger at maximum frequency in GPSVT, it should be taken into account that the clock frequency is also practically twice as high.

Table IV. Sabrewing Compared to a Conventional Solution

		Baseline	Sabrewing	Improvement
Maximum Clock Frequency	LPHVT	425MHz	667MHz	57%
Maximum Clock Frequency	GPSVT	713MHz	1.35GHz	89%
Area at 200MHz	LPHVT	51790μm^2	43640μm^2	19%
Area at Max. Frequency	LPHVT	76210μm^2	66653μm^2	14%
Area at Max. Frequency	GPSVT	65215μm^2	78086μm^2	-
Integer Power at 200MHz	LPHVT	24.9mW	6.7mW	74%
Floating-Point Power at 200MHz	LPHVT	37.3mW	12.5mW	67%
Normalized Power at Max. Frequency	LPHVT	220μW/MHz	64μW/MHz	71%

In addition to the baseline comparison, we have compared Sabrewing to a dedicated and area optimized integer MAC. This MAC was implemented in LPHVT with the clock frequency constrained to 200MHz. According to our expectations, the area of the MAC is 11858 μm^2 which is about 27% of the area that Sabrewing occupies. The dedicated MAC consumes approximately 4.3mW to perform the FIR filter that we used to determine Sabrewing's power consumption (Section 7.2.3). This confirms that inefficiencies still reside in the current Sabrewing design. We expect that applying clock gating will be adequate to remedy these inefficiencies. Note that, despite the comparison being made here, Sabrewing is not meant to compete with its dedicated counterparts.

8.3. IEEE Compliance

There exists some controversy over what is IEEE-754 compliance. Many manufacturers and researchers label their work as IEEE compliant while in fact they just implement a subset of the functionality defined in IEEE [2008]. We would therefore not call Sabrewing IEEE compliant, but it does produce IEEE compliant results. In order to guarantee this, extensive testing was done using a floating-point verification framework from IBM [IBM Haifa Research Lab 2011]. In contrast to many others, IBM's floating-point test is contemporary and not randomly generated. It specifically targets the error prone parts of the datapath and includes many test cases for FMA. IBM is pioneer in the field of FMA and they have many years of experience in testing such datapaths. The fact that Sabrewing passes all of IBM's test cases leads us to say that we are confident the results produced by our design are compliant with IEEE-754.

8.4. Future Work

The feasibility and advantages of combined floating-point and integer hardware have been demonstrated in this article. An important question that remains to be answered is what the architectural implications of this approach are. The first major challenge is instruction scheduling. Further research will focus on scheduling techniques to determine how beneficial combining floating-point and integer arithmetic will be in practice. Exploration of additional hardware features and optimizations is also planned. Although modified Booth multiplication suits the Sabrewing design well, the Baugh-Wooley algorithm might yield even better results Sjalander and Larsson-Edefors [2008]. In terms of features we think vectorized (SIMD) integer instructions and flexible rounding will be interesting for Sabrewing. The Sabrewing datapath is very wide because of the parallel alignment optimization. This makes is suitable to perform SIMD instructions with smaller (integer) operands. By flexible rounding we envision a technique that can resize the extended floating-point number shown in Figure 1 to IEEE single precision at runtime. This could be done by performing the rounding operation on the 23 MSBs of the significand instead of the complete 32-bit significand. Such a rounding technique will be advantageous for industrial applications

that require operations the be performed exactly as in the official IEEE single precision exchange format. Whether these extension are worthwhile or not remains to be seen as they will increase the area and complexity of the circuit and thus partially eliminate the purpose of Sabrewing, which is that it is lightweight.

8.4.1. SoC Integration. The next step will be to integrate Sabrewing in a Xentium® [Recore Systems 2011] integer/fixed-point DSP core running at 200MHz. Various design choices in Sabrewing were made to ensure that the architecture is not only innovative, but also compatible with existing hardware. We therefore expect that integration will require little effort. Once integration is completed, we will use the Xentium® as a vehicle to do further testing.

9. CONCLUSION

In this article we presented Sabrewing, a new combined floating-point and integer arithmetic architecture designed primarily for low-power embedded digital signal processors. Many DSP applications will map nicely to this new architecture because it offers various floating-point and integer multiply-add operations. In addition the number of pipeline stages is kept low (three) so that developing efficient compilers for this architecture will not be a very difficult task.

It was shown that a prototype supporting 41-bit floating-point and 32-bit integer operands, implemented in 65nm low-power technology and operating at 200MHz, has an area of just 0.04mm^2. This design's power consumption ranges from about 6.7mW to 12.5mW based on a 4-tap FIR filter performed in integer (fixed-point) and floating-point, respectively. Sabrewing can be clocked up to 667MHz in low-power technology and even at 1.35GHz when implemented in general-purpose technology.

By tightly integrating integer functionality in a fused-multiply-add floating-point datapath, at least 19% less area is required compared to a conventional approach where the same functionality is implemented in two separate datapaths. Compared to the same conventional baseline, a power reduction of 67% can be observed. The amount of area overhead created by augmenting the FMA datapath with integer functionality is less than 5%. Considering all these aspects, we conclude that the Sabrewing architecture will be a valuable asset for hardware platforms that require floating-point arithmetic but are on a tight area and power budget.

REFERENCES

ANALOG DEVICES. 2011. SHARC ADSP-21478/ADSP-21479 (Rev. 0). Datasheet. http://www.analog.com/static/imported-files/data_sheets/ADSP-21478_21479.pdf.

BISHOP, D. W. 2011. VHDL-2008 support library. http://www.vhdl.org/fphdl.

BLAKE, G., DRESLINSKI, R. G., AND MUDGE, T. 2009. A survey of multicore processors. *IEEE Signal Process. Mag. 26,* 6, 26–37.

BOOTH, A. D. 1951. A signed multiplication technique (Part 2). *Quart. J. Mech. Appl. Math 4*, 236–240.

BRUINTJES, T. M. 2011. Design of a fused multiply-add floating-point and integer datapath. M.S. thesis, University of Twente, the Netherlands. http://eprints.eemcs.utwente.nl/20466/.

BUTLER, M., BARNES, L., SARMA, D. D., AND GELINAS, B. 2011. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro 31,* 2, 6–15.

CORNEA, M., HARRISON, J., AND TANG, P. T. P. 2003. Intel itanium floating-point architecture. In *Proceedings of the Workshop on Computer Architecture Education,* Held in Conjunction with *the 30th International Symposium on Computer Architecture*. (WCAE'03). ACM, New York, NY.

HAYES, W. P., KERSHAW, R. N., BAYS, L. E., BODDIE, J. R., FIELDS, E. M., FREYMAN, R. L., GAREN, C. J., HARTUNG, J., KLINIKOWSKI, J. J., MILLER, C. R., MONDAL, K., MOSCOVITZ, H. S., ROTBLUM, Y., STOCKER, W. A., TOW, J., AND TRAN, L. V. 1985. A 32-bit VLSI digital signal processor. *IEEE J. Solid-State Circuits 20,* 5, 998–1004.

HENNESSY, J. L. AND PATTERSON, D. A. 2006. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, Chapter A, A2–A77.

HUANG, L., SHEN, L., DAI, K., AND WANG, Z. 2007. A new architecture for multiple-precision floating-point multiply-add fused unit design. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, Los Alamitos, CA, 69–76.

IBM HAIFA RESEARCH LAB. 2011. FPgen: A deep-knowledge coverage-driven floating-point test generator. https://www.research.ibm.com/haifa/projects/verification/fpgen.

IEEE. 2008. IEEE 754-2008, standard for floating-point arithmetic. IEEE-STD, 3 Park Avenue, New York, NY 10016-5997.

IEEE. 2009. *IEEE Standard VHDL Language Reference Manual*. IEEE-STD, 3 Park Avenue, New York, NY 10016-5997.

JESSANI, R. M. AND PUTRINO, M. 1998. Comparison of single- and dual-pass multiply-add fused floating-point units. *IEEE Trans. Computers 47,* 9, 927–937.

LAVENIER, D., SOLIHIN, Y., AND CAMERON, K. 2000. Integer/floating-point reconfigurable ALU. In *Proceedings of the 6th Symposium on New Machine Architectures (SympA'6)*.

MARKSTEIN, P. 2004. Software division and square root using goldschmidt's algorithms. In *Proceedings of the 6th Conference on Real Numbers and Computers*. 146–157.

MONTOYE, R. K., HOKENEK, E., AND RUNYON, S. L. 1990. Design of the IBM RISC System/6000 floating-point execution unit. *IBM J. Res. Develop. 34,* 1, 59–70.

MUELLER, S. M., JACOBI, C., OH, H.-J., TRAN, K. D., COTTIER, S. R., MICHAEL, B. W., NISHIKAWA, H., TOTSUKA, Y., NAMATAME, T., YANO, N., MACHIDA, T., AND DHONG, S. H. 2005. The vector floating-point unit in a synergistic processor element of a CELL processor. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17).* 59–67.

OKLOBDZIJA, V. G. 1994. An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis. *IEEE Trans. (VLSI) Syst. 2,* 1, 124–128.

PALACHARLA, S. AND SMITH, J. E. 1995. Decoupling integer execution in superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO'28)*. IEEE Computer Society Press, Los Alamitos, CA, 285–290.

PARHAMI, B. 2000. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Oxford, UK.

QUINNELL, E., SWARTZLANDER, E. E., AND LEMONDS, C. 2007. Floating-point fused multiply-add architectures. In *Proceedings of the of the 41st Asilomar Conference on Signals, Systems and Computers (ACSSC'07).* 331–337.

RECORE SYSTEMS. 2011. Xentium technology. http://www.recoresystems.com/technology/xentium-technology.

ROBISON, A. D. 2005. N-bit unsigned division via n-bit multiply-add. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17).* 131–139.

SCHMOOKLER, M. S. AND NOWKA, K. J. 2001. Leading zero anticipation and detection—A comparison of methods. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic.* 7–12.

SCHWARZ, E. M. 2006. Binary floating-point unit design. In *High-Performance Energy-Efficient Microprocessor Design*, Series on Integrated Circuits and Systems, Springer, 189–208.

SJALANDER, M. AND LARSSON-EDEFORS, P. 2008. High-speed and low-power multipliers using the baugh-wooley algorithm and HPM reduction tree. In *Proceedings of the 15th IEEE International Conference on Electronics, Circuits and Systems (ICECS'08).* 33–36.

SOLIHIN, Y., CAMERON, K., LUO, Y., LAVENIER, D., AND GOKHALE, M. 2001. Mutable functional units and their applications on microprocessors. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*. IEEE Computer Society, Los Alamitos, CA, 234–239.

SYNOPSYS. 2011. DesignWare building block IP overview. http://www.synopsys.com/dw/buildingblock.php.

TER BRAAK, T. D., BURGESS, S. T., HURSKAINEN, H., KERKHOFF, H. G., VERMEULEN, B., AND ZHANG, X. 2010. Online dependability enhancement of multiprocessor SoCs by resource management. In *Proceedings of the International Symposium on System-on-Chip*. IEEE, 103–110.

TEXAS INSTRUMENTS. 2011. TMS320C6742 fixed/floating point digital signal processor (Rev. C). datasheet. http://www.ti.com/lit/ds/symlink/tms320c6742.pdf.

THAPLIYAL, H., ARABNIA, H. R., AND VINOD, A. P. 2006. Combined integer and floating point multiplication architecture (CIFM) for FPGAs and its reversible logic implementation. In *Proceedings of the 49th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS'06).* Vol. 2, 438–442.

VASSILIADIS, S., LEMON, D. S., AND PUTRINO, M. 1989. S/370 Sign-magnitude floating-point adder. *IEEE J. Solid-State Circuits 24,* 4, 1062–1070.

WALLACE, C. S. 1964. A suggestion for a fast multiplier. *IEEE Trans. Electron. Computers 13,* 1, 14–17.

WALTERS, K. H. G., GEREZ, S. H., SMIT, G. J. M., BAILLOU, S., RAUWERDA, G. K., AND TRAUTNER, R. 2011. Multicore SoC for on-board payload signal processing. *Adaptive Hardware and Systems*. To appear.

WITTENBRINK, C. M., KILGARIFF, E., AND PRABHU, A. 2011. Fermi GF100 GPU architecture. *IEEE Micro 31,* 2, 50–59.

WOO-CHAN, P., SHI-WHA, L. E. E., OH-YOUNG, K., TACK-DON, H. A. N., AND SHIN-DUG, K. I. M. 1996. Floating point adder/subtractor performing IEEE rounding and addition/subtraction in parallel. *IEICE Trans. Inf. Syst. 79,* 4, 297–305.

Received July 2011; revised October 2011, November 2011; accepted December 2011