Hybrid Hardware/Software Floating-Point Implementations for Optimized Area and Throughput Tradeoffs

Jon J. Pimentel, *Student Member, IEEE*, Brent Bohnenstiehl, *Student Member, IEEE*, and Bevan M. Baas, *Senior Member, IEEE*

***Abstract*— Hybrid floating-point (FP) implementations improve software FP performance without incurring the area overhead of full hardware FP units. The proposed implementations are synthesized in 65-nm CMOS and integrated into small fixed-point processors with a RISC-like architecture. Unsigned, shift carry, and leading zero detection (USL) support is added to a processor to augment an existing instruction set architecture and increase FP throughput with little area overhead. The hybrid implementations with USL support increase software FP throughput per core by 2.18× for addition/subtraction, 1.29× for multiplication, 3.07–4.05× for division, and 3.11–3.81× for square root, and use 90.7–94.6% less area than dedicated fused multiply–add (FMA) hardware. Hybrid implementations with custom FP-specific hardware increase throughput per core over a fixed-point software kernel by 3.69–7.28× for addition/subtraction, 1.22–2.03× for multiplication, 14.4× for division, and 31.9× for square root, and use 77.3–97.0% less area than dedicated FMA hardware. The circuit area and throughput are found for 38 multiply–add, 8 addition/subtraction, 6 multiplication, 45 division, and 45 square root designs. Thirty-three multiply–add implementations are presented, which improve throughput per core versus a fixed-point software implementation by 1.11–15.9× and use 38.2–95.3% less area than dedicated FMA hardware.**

***Index Terms*— Arithmetic and logic structures, computer arithmetic, fine-grained system, floating point (FP).**

I. Introduction

FLOATING-POINT (FP) representation is the most commonly used method for approximating real numbers in modern computers [1]. However, the large area and power requirement of FP hardware limit many architectures to fixed-point arithmetic, for example, software-defined radio architectures [2], Blackfin microprocessors [3], picoChip [4], the Xscale core [5], and massively parallel processor chips such as AsAP [6], [7]. Small chip area is especially critical for many-core architectures, since increasing area per core has a dramatic effect on total chip area and can much more easily reduce the number of cores that will fit on a chip die. There is also interest in adding embedded FP units (FPUs) in FPGAs [8], though most commercial vendors do not offer dedicated hard block FPUs due to the large area overhead [9].

Several approaches have been explored for increasing FP throughput and maintaining low area overhead. Fused and cascade multiply–add FPUs improve accuracy and provide computational speedup [10], [11]; however, they introduce large area [12] and power overhead, which are undesirable for simple fixed-point processors. If blocks of data have similar magnitudes, block FP (BFP) can be useful for increasing SNR and dynamic range [13], [14]. Microoperations have been used to create a virtual FPU, which reuse existing fixed-point hardware to emulate an FP datapath for a very long instruction word processor [14]. Hardware prescaling and postscaling has also been used to reduce the required hardware for FP division and square root [15]. The hardware overhead can be reduced by shortening the exponent and mantissa widths for video coding [16], audio applications [17], and radar image formation [18]. Some speech recognition and image processing applications have been shown to not require the full mantissa width [19]. Custom FP instructions have also been explored for an FPGA to increase FP throughput with lower area overhead than a full hardware FPU [21]. However, Hockert and Compton [21] did not consider modular FPUs built from standalone addition/subtraction and multiplication designs or the throughput when performing the multiply–add operation, nor did they explore the area and throughput tradeoffs of various division and square root algorithms.

This paper presents hybrid FP implementations, which perform FP arithmetic on a small fixed-point processor using a combination of fixed-point software instructions and additional hardware. Hybrid implementations offer alternative area–throughput tradeoffs to full software or full hardware approaches (Fig. 1). They provide higher throughput than full

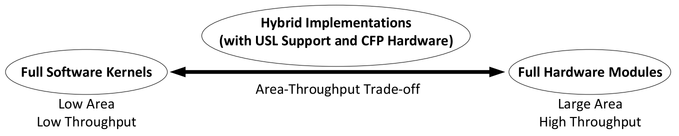


Fig. 1. Hybrid implementations offer alternatives to pure software and pure hardware designs and enable a spectrum of designs with varying levels of chip area and throughput.

Manuscript received October 21, 2015; revised February 20, 2016 and April 28, 2016; accepted June 7, 2016. Date of publication July 12, 2016; date of current version December 26, 2016. This work was supported in part by the National Science Foundation under Grant 0903549, Grant 1018972, and Grant CAREER 0546907, in part by Semiconductor Research Corporation under Grant CSR 1659, Grant GRC 1971, and Grant GRC 2321, and in part by ST Microelectronics.

The authors are with the Department of Electrical and Computer Engineering, University of California at Davis, Davis, CA 95616-5294 USA (e-mail: jjpimentel@ucdavis.edu; bvbohnen@ucdavis.edu; bbaas@ucdavis.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2016.2580142

1063-8210 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

software kernels by including either custom FP-specific (CFP) instructions or unsigned, shift carry, and leading-zero detection (USL) support, which replaces long sections of code, thereby performing the same operation in a fewer cycles. This paper demonstrates that hybrid implementations require less area than conventional full hardware modules by using the existing fixed-point hardware, such as the arithmetic logic unit (ALU) and multiply–accumulate (MAC) unit.

USL support is added to a simple fixed-point processor to determine the area and throughput tradeoffs provided by minimal architectural improvements to the instruction set architecture (ISA). These architectural improvements improve FP throughput without adding FP-specific hardware. The ISA modifications include adding unsigned operation support, leading zero detection, and additional shift instructions. A set of hybrid implementations with USL support is created, which typically require less area than the hybrid implementations with CFP hardware, but offer less throughput.

The main contributions of this paper are as follows.

1) Eight hybrid implementations with CFP hardware and six with USL support.
2) Design and implementation of 38 multiply–add, 8 addition/subtraction, 6 multiplication, 45 division, and 45 square root designs. These designs include full software kernels, full hardware modules, hybrid implementations with USL support, and hybrid implementations with CFP hardware. Three different algorithms for division and three for square root are utilized.
3) Evaluation of the proposed software kernels, hardware modules, and hybrid implementations, and FPUs (i.e., the combination of two or more FP software kernels, hardware modules, or hybrid implementations) in terms of area, throughput, and instruction count when performing FP multiply–add, addition/subtraction, multiplication, division, and square root.

The remainder of this paper is organized as follows. Section II presents background on the FP format, the algorithms utilized, and the targeted architecture. Section III presents the full software kernels and Section IV covers the full hardware modules. Section V discusses the proposed hybrid implementations with USL support. Section VI discusses the proposed hybrid implementations with CFP hardware. Section VII evaluates the software kernels, hardware modules, and hybrid implementations, demonstrates two examples for determining the optimal designs, and presents benchmark results. Section VIII provides insights into the advantages of the hybrid approaches. Section IX compares the results of this paper to previous work. Section X summarizes the findings of this work.

II. Floating-Point Computation Background

A. Floating-Point Format

This work uses the IEEE-754 single-precision format for all FP arithmetic, with values on the normalized value interval $\pm[2^{-126}, (2-2^{-23}) \times 2^{127}]$ [22]. In addition, round to nearest even, the IEEE-754 default rounding mode, and round toward zero are supported for all FP arithmetic. However, in order to reduce overhead, the following features are not supported: exception handling, not a number, ±infinity, denormalized values, and alternative rounding modes. Many applications, such as some multimedia and graphics processing, do not rely on all elements of the standard [20], [23].

B. Floating-Point Arithmetic

The FP operation algorithms are explained below.

1) Addition/Subtraction: This operation begins by determining the smaller magnitude operand, aligning the mantissas of the two operands based on the difference in their exponents, and adding or subtracting the mantissas based on the desired operation and the signs of the operands. The initial exponent is set to the exponent of the larger magnitude operand. The result is then normalized and rounded. The sign bit is determined by the larger input operand.

2) Multiplication: This operation begins by multiplying the mantissas together. The initial exponent is set by adding the operand exponents, the product is then normalized and rounded, and the sign bit is set by XORing the sign bit of both operands together.

3) Multiply–Add: The multiply–add operation performs $a + b \times c$. The unfused multiply–add first calculates $b \times c$, rounds the result, adds the rounded product to the addend a, and then performs a second rounding. The fused multiply-add (FMA) rounds once, after the product is added to the addend.

4) Division: Three algorithms are implemented for division: long-division [24], nonrestoring [25], and Newton–Raphson [1]. Division is typically an infrequent operation [26], [27]; therefore, little area should be allocated. The long-division and nonrestoring algorithms are chosen for their simplicity and low area impact, whereas the Newton–Raphson algorithm is selected for its potentially high throughput [28].

The long-division algorithm first compares the magnitude of the divisor and the dividend. If the divisor is smaller than or equal to the dividend, then it is subtracted from the dividend to form a partial remainder, and a 1 is right shifted in as the next bit of the quotient. Otherwise, they are not subtracted and a 0 is shifted in for the next bit of the quotient [24]. The partial remainder is then left shifted by 1 bit and set as the new dividend. This process continues until all of the quotient bits are determined. The result is then normalized and rounded. The result exponent is calculated by subtracting the input exponents and adding back the bias.

The nonrestoring division algorithm is similar to the restoring algorithm except that it avoids the restoring step for each loop iteration to improve performance [29]. The divisor is first subtracted from the dividend. A loop is executed that first checks if the result is negative or positive, and then left shifts the quotient and the result. If the result is negative, the dividend is added to the result. If the result is positive, the least significant bit (LSB) of the quotient is set to 1 and the dividend is subtracted from the result [30]. This loop iterates until all bits are determined for the quotient [25]. The final step then restores the divisor if the result was negative. The result is then normalized and rounded. The result exponent is calculated in the same manner as long division.

For the Newton–Raphson division algorithm, the reciprocal of the divisor is determined iteratively and then multiplied by the dividend [1]. The divisor and dividend are first scaled down to a small interval. A linear approximation is then used to estimate the reciprocal and minimize the maximum relative error of the final result [31]. This estimation is then improved iteratively. Once this reciprocal is determined, it is multiplied by the scaled dividend to obtain the result, which is then refined by computing residuals at a higher precision [24].

These division algorithms calculate the result's sign by XORing the sign bits of both operands.

5) Square Root: Three algorithms are used for performing square root: digit-by-digit, Newton–Raphson [1], and nonrestoring [32]. Square root is typically an infrequent operation [26], [27]; therefore, little area should be allocated. The digit-by-digit and nonrestoring algorithms are chosen for their low area impact, while the Newton–Raphson method is chosen for providing high throughput since the algorithm converges quadratically rather than linearly [33].

The digit-by-digit algorithm first determines the result exponent. If the unbiased exponent is odd, one is subtracted to make it even and the radicand mantissa is left shifted to account for the change without a loss of precision. The exponent is then right shifted by 1 bit. Solving for the root mantissa then begins by setting the most significant bit (MSB) of the root to one, squaring the root, and subtracting it from the radicand. If the result is negative, the radicand's MSB is set to zero; otherwise, it is left as a one. The next MSB of the root is then set to 1, and the process is continued until all of the root bits are determined. The squaring step is unnecessary for the first iteration of this loop. The result is then normalized and rounded.

The nonrestoring square root algorithm involves a loop where each iteration calculates one digit of the square root exactly and the digits are based on whether the partial remainder is negative or positive [32]. The result exponent is determined by dividing the original exponent by two and adding 63, which is half the bias rounded down. The LSB of the original exponent is then added to this sum.

The Newton–Raphson square root algorithm finds the reciprocal of the square root first using an algorithm similar to Newton–Raphson division [1]: scaling the input, applying the linear approximation [34], and iterating to improve the approximation. The result is determined by multiplying the reciprocal approximation by the original input and corrected via the Tuckerman test [35].

C. Targeted Architectures

This work applies to any fixed-point architecture. Several methods for performing FP operations on a fixed-point datapath are evaluated on the asynchronous array of simple processors architecture (AsAP2). AsAP2 is an example of a fine-grained many-core system with a fixed-point datapath [36] and features 164 simple independently clocked homogeneous programmable processors. Each processor occupies 0.17 mm^2 in 65-nm CMOS technology and can operate up to a maximum clock frequency of 1.2 GHz at 1.3 V [7]. Processors support 63 general-purpose instructions, contain a 128×35-bit instruction memory and a 128×16-bit data memory, and implement a 16-bit fixed-point signed-only datapath including a MAC unit with a 40-bit accumulator. The platform is capable of computing a wide range of applications including audio and video processing [37], [38], as well as ultrasound image processing [39].

III. Full Software Kernels

These full software kernels are coded in AsAP instructions and form a software library consisting of addition/subtraction, multiplication, division, and square root. They are referred to as "full software" because they utilize only general-purpose fixed-point instructions. Since the platform's word size is 16 bits, each value is received on chip as two words. To simplify software computation, these words are split into four to store the following: the sign bit, exponent, high and low mantissa bits. Since these kernels use only the platform's existing fixed-point datapath, they do not add area.

The programs for these kernels are large due to the lack of unsigned ALU instructions and the number of fixed-point instructions required for emulating FP hardware. Computation time for software FP consists primarily of operand comparisons, mantissa alignment, addition, normalization, and rounding.

A. Addition/Subtraction Kernel (Full SW Add/Sub)

Since 222 instructions are required for this kernel, two processors are needed for sufficient instruction memory. The first processor sorts the operands and aligns the mantissas. The second processor adds the mantissas, normalizes, and rounds.

B. Multiplication Kernel (Full SW Mult)

Most of the instructions overheads for this kernel are used for performing mantissa multiplication and rounding. The partial products of the multiplication are created and added using the MAC and aligned using the shifter.

C. Division Kernel Version 1 (Full SW Div Ver. 1)

This kernel uses the long-division algorithm [24]. The loop to determine the quotient requires the greatest number of instructions and involves several shift and subtract operations.

D. Division Kernel Version 2 (Full SW Div Ver. 2)

This kernel uses the Newton–Raphson algorithm [1]. The kernel begins with zero input detection and handling, followed by exponent calculation. The input is then prepared for later calculations. The initial estimate of the reciprocal is calculated, followed by Newton–Raphson iterations. The first input is then multiplied by the reciprocal of the second, and then the result is normalized and rounded. Finally, the LSB is corrected.

E. Square Root Kernel Version 1 (Full SW Sqrt Ver. 1)

This kernel uses the digit-by-digit method. Most of the overhead involves squaring each value being tested.

TABLE I
INSTRUCTIONS USED BY EACH FP DESIGN

	FP Design	Additional Instructions Used
Full Software Modules	*Full SW Add/Sub*	*None*
	Full SW Mult	*None*
	Full SW Div Ver. 1	*None*
	Full SW Div Ver. 2	*None*
	Full SW Sqrt Ver. 1	*None*
	Full SW Sqrt Ver. 2	*None*
Full Hardware Modules	*Full HW Add/Sub*	*FPAdd, FPSub*
	Full HW Add/Sub (32-bit I/O)	*FPAdd32, FPSub32*
	Full HW Mult	*FPMult*
	Full HW Mult (32-bit I/O)	*FPMult32*
	Full HW Div	*FPDiv*
	Full HW Div (32-bit I/O)	*FPDiv32*
	Full HW Sqrt	*FPSqrt*
	Full HW Sqrt (32-bit I/O)	*FPSqrt32*
	Full HW FMA	*FMA*
	Full HW FMA (32-bit I/O)	*FMA32*
Hybrid Implementations with USL	*Hybrid Add/Sub w/ USL*	*ADDU, ADDUC, LZD, SHLC, SHRC, SUBU, SUBUC*
	Hybrid Mult w/ USL	*ADDU, ADDUC, MACCUL, MACUL, MACUH, MULTUL, SHLC*
	Hybrid Div w/ USL Ver. 1	*ADDU, ADDUC, SHLC, SUBU, SUBUC*
	Hybrid Div w/ USL Ver. 2	*ACCSHU, ADDU, ADDUC, MACCUL, MACUH, MACUL, SHRC, SHLC, SUBU, SUBUC*
	Hybrid Sqrt w/ USL Ver. 1	*ACCSHU, ADDU, ADDUC, MACCUL, MACUH, MACUL, SHRC, SHLC, SUBU, SUBUC*
	Hybrid Sqrt w/ USL Ver. 2	*ACCSHU, ADDU, ADDUC, MACCUL, MACUL, SHRC, SHLC, SUBU, SUBUC*
Hybrid Implementations with CFP	*Hybrid Add/Sub w/ CFP Ver. 1*	*BShiftL, FPAdd_SatAlign, FPAdd_Round, LZD*
	Hybrid Add/Sub w/ CFP Ver. 2	*BShiftL, FPAdd_AlignSmall, FPAdd_Round, LZD*
	Hybrid Add/Sub w/ CFP Ver. 3	*BShiftL, FPAdd_Align, FPAdd_Compare, FPAdd_Round, LZD*
	Hybrid Add/Sub w/ CFP Ver. 4	*Shift_LZA, FPAdd_Align, FPAdd_Round*
	Hybrid Mult w/ CFP Ver. 1	*FPMult_NormRndCarry*
	Hybrid Mult w/ CFP Ver. 2	*FPMult_NormRnd*
	Hybrid Div w/ CFP Ver. 1	*FPDiv_LoopExpAdj*
	Hybrid Sqrt w/ CFP Ver. 1	*FPSqrt_Loop*

F. Square Root Kernel Version 2 (Full SW Sqrt Ver. 2)

This kernel uses the Newton–Raphson method, similar to *Full SW Div Ver. 2*, except the first input is multiplied by the reciprocal of the square root. Most of the instruction overhead is from handling multiword values.

IV. FULL HARDWARE MODULES

Full hardware modules offer the highest throughput, but require the most area of the designs implemented. These modules are referred to as "full hardware" because all arithmetic is performed on dedicated FP hardware. Since the target platform has a 16-bit datapath, the FP values are first loaded into FP registers. Each value is stored as two 16-bit words. An entire FP operation is carried out by a single FP instruction and the results are read from the FP registers, 16 bits at a time. The instructions used in each module are shown in Table I.

For comparison purposes, a separate version of each module is created, with a 32-bit word size and datapath. The full hardware modules are discussed as follows.

A. Fused Multiply–Add Module (Full HW FMA)

The full hardware FMA module uses the *FMA* instruction, with a two-cycle execution latency. The design of the module matches that of a traditional single-path FMA architecture, similar to the FMA in the IBM RS/6000 [1], [40]. The addend is complemented if effective subtraction is performed and right shifted by the exponent difference. The multiplier uses radix-4 Booth encoding with reduced sign extension, limiting the widths of the partial products to 28 and 29 bits. The partial products are then compressed using a Wallace tree into carry-save format. A 3:2 carry-save adder then adds these values and the lower 48 bits of the shifted addend. An end-around carry adder with a carry lookahead adder computes the sum. In parallel, a leading zero anticipator (LZA) determines the number of leading zeros for the result, to within 1 place [41], [42]. The result is complemented if the addend is larger than the product. The result is normalized using the LZA count, followed by a possible 1-bit correction and rounding.

Full HW FMA (32-bit I/O) is created for a 32-bit datapath and word size and uses *FMA32* with a two-cycle execution latency. This instruction uses three source operands.

B. Addition/Subtraction Module (Full HW Add/Sub)

This module uses the *FPAdd* and *FPSub* instructions with a two-cycle execution latency each.

Full HW Add/Sub (32-bit I/O) is created for a 32-bit datapath and word size and uses *FPAdd32* and *FPSub32*, each of which has a single-cycle execution latency. If operands are read from a processor's local memory, then a single instruction can perform addition/subtraction.

C. Multiplication Module (Full HW Mult)

This module uses the *FPMult* instruction with a single-cycle execution latency to perform multiplication.

Full HW Mult (32-bit I/O) is created for a 32-bit datapath and word size and uses the *FPMult32* instruction to perform multiplication with a single-cycle execution latency. Assuming operands are read from a processor's local memory, a single instruction can perform multiplication.

D. Division Module (Full HW Div)

This module performs the restoring division algorithm [25] using *FPDiv*. This instruction has a 30-cycle execution latency.

Full HW Div (32-bit I/O) is created for a 32-bit datapath and word size and uses the *FPDiv32* instruction with a 30-cycle execution latency. A single instruction can perform division if operands are read from a processor's local memory.

E. Square Root Module (Full HW Sqrt)

This module uses *FPSqrt* with a 26-cycle execution latency to perform the nonrestoring square root algorithm [32].

Full HW Sqrt (32-bit I/O) is created for a 32-bit datapath and word size and uses the *FPSqrt32* instruction to perform square root operations with a 26-cycle execution latency. A single instruction can perform square root operations.

V. Proposed Hybrid Implementations With Unsigned, Shift-Carry, and Leading Zero Detection Support

To determine the throughput and area achievable by increasing the instruction set, USL support is added to the target platform's ISA. Several ISA modifications are implemented, including adding unsigned operation support, leading zero detection, and additional shift-carry instructions. These extra shift instructions can set a carry flag if data are shifted out. Table I indicates the instructions utilized, each of which has a single-cycle execution latency, except for the MAC instructions, which require two cycles. Each value is split across four 16-bit words. The following instructions are implemented.

1) *SUBU:* Unsigned subtraction.
2) *SUBUC:* Unsigned subtraction instruction with borrow if the carry flag is asserted.
3) *ADDU:* Unsigned addition.
4) *ADDUC:* Unsigned addition with carry in.
5) *SHLC:* Shift left with carry in.
6) *SHRC:* Shift right with carry in.
7) *LZD:* Return number of leading zeros.
8) *MULTUL:* Unsigned multiply that returns the 16 LSBs of the result. The accumulator is not overwritten.
9) *MACCUL:* Unsigned multiply that returns the 16 LSBs of the result. The accumulator is overwritten with the result.
10) *MACUL:* Unsigned multiply-accumulate that returns the lower 16 LSBs of the result.
11) *MACUH:* Unsigned multiply–accumulate that returns the 16 MSBs of the result.
12) *ACCSHU:* Unsigned right shift for the accumulator and returns the 16 LSBs of the result.

Each of the implementations is described as follows.

A. Addition/Subtraction Hybrid Implementation With USL Support (Hybrid Add/Sub w/ USL)

Unsigned addition/subtraction operations increase throughput for sorting operands, calculating the exponent difference, adding/subtracting the mantissas, and rounding. The additional shift instructions and the *LZD* reduce normalization overhead.

B. Multiplication Hybrid Implementation With USL Support (Hybrid Mult w/ USL)

The unsigned multiply–accumulate instructions reduce the overhead for partial product calculation. The additional shift instruction eases normalization and the unsigned addition instructions reduce the instruction count for rounding.

C. Division Hybrid Implementation With USL Support Version 1 (Hybrid Div w/ USL Ver. 1)

This implementation uses the long-division algorithm [24]. Unsigned addition/subtraction and added shift instructions reduce the instruction count for exponent calculation, divisor and dividend mantissa subtraction, rounding, and normalization.

D. Division Hybrid Implementation With USL Support Version 2 (Hybrid Div w/ USL Ver. 2)

This implementation uses the Newton–Raphson division algorithm [1]. The unsigned addition/subtraction, multiply–accumulate, and additional shift instructions reduce the instruction count for calculating the exponent and initial estimate, executing the Newton–Raphson iterations, multiplying the input by the reciprocal, rounding, and correcting the LSB.

E. Square Root Hybrid Implementation With USL Support Version 1 (Hybrid Sqrt w/ USL Ver. 1)

This implementation uses the digit-by-digit method. Unsigned addition/subtraction instructions decrease the instruction count for exponent calculation, root incrementing, radicand and square root subtraction, and rounding. Unsigned multiply–accumulate reduces the instruction count for squaring the root being tested, and the additional shift instructions assist with setting the next radicand bit and alignment.

F. Square Root Hybrid Implementation With USL Support Version 2 (Hybrid Sqrt w/ USL Ver. 2)

This implementation uses the Newton–Raphson square root algorithm [1]. Unsigned instructions ease rounding and the correction of the LSB, calculating the exponent, determining the initial value, and performing the Newton–Raphson iterations. The additional shift instructions help with preparing the input data and the Newton–Raphson iterations.

VI. Proposed Hybrid Implementations With Custom FP-Specific Hardware

Hybrid implementations with CFP hardware are composed of fixed-point software and custom FP instructions operating together on FP workloads [43]. They increase throughput by reducing the bottlenecks of full software kernels and require less area than full hardware modules.

CFP instructions perform operations on data stored in FP registers, and each value is stored as two 16-bit words. Table I indicates the instructions utilized in each implementation, where each instruction has a single-cycle execution latency. Eight implementations are described in the following.

A. Addition/Subtraction Hybrid Implementation With CFP Hardware Version 1 (Hybrid Add/Sub w/ CFP Ver. 1)

With this implementation, fixed-point instructions sort the operands and calculate the exponent difference. The CFP instructions described in the following perform the rest of the operation.

1) FPAdd_SatAlign: The FP registers are loaded with the sorted FP operands. This instruction saturates the exponent difference and then aligns and adds the mantissas. The hidden bits are inserted and the sticky bit is determined. For effective subtraction, the smaller magnitude operand's mantissa is inverted and a one is added. The unnormalized result is stored in an FP register and the 16 MSBs are returned.

2) LZD: Following mantissa addition, *LZD* counts the leading zeros of the result. This count is used to normalize.

3) BShiftL: Using the shift amount determined by *LZD* and the sum stored in the FP registers by *FPAdd_SatAlign* and *BShiftL* shifts left for normalization, adjusts the exponent, and stores the result's 27 LSBs in an FP register.

4) FPAdd_Round: Following normalization, *FPAdd_Round* performs rounding and exponent adjustment. The final result is written to an FP register and the 16 MSBs are output.

B. Addition/Subtraction Hybrid Implementation With CFP Hardware Version 2 (Hybrid Add/Sub w/ CFP Ver. 2)

Operand sorting, exponent difference calculation and saturation, sticky bit calculation, and hidden bit insertion are performed with fixed-point instructions. This implementation utilizes *FPAdd_AlignSmall*, which is described as follows.

1) FPAdd_AlignSmall: This instruction aligns and adds the mantissas using the software calculated shift amount. The rest of the operation uses *LZD*, *BShiftL*, and *FPAdd_Round*.

C. Addition/Subtraction Hybrid Implementation With CFP Hardware Version 3 (Hybrid Add/Sub w/ CFP Ver. 3)

Algorithm 1 displays the pseudocode for this implementation; variables are italicized, comments are in green font, CFP instructions are bolded and in blue font, and all other lines represent operations carried out by fixed-point software. After the input operands are loaded, *FPAdd_Compare* sorts the operands and calculates the saturated shift amount and stores this value in *ExpDiff*. *FPAdd_Align* reads the variable *ExpDiff* to perform the mantissa alignment, possibly complements one of the mantissas, and then adds them. The result is stored in *FPReg1* and the 16 MSBs are stored in *FPreg3*. *LZD* stores the leading zeros count of *FPReg3* in *UpperZeros*. If all bits were zero, then *LZD* counts the leading zeros in the LSBs of *FPReg1*. *BShiftL* then normalizes after adding the leading zeros counts together. *FPAdd_Round* then rounds the normalized result and outputs the 16 MSBs. *FPAdd_Compare* and *FPAdd_Align* are described as follows.

1) FPAdd_Compare: This instruction sorts both operands after they are loaded into the FP registers. The sorted operands are then rewritten into the FP registers. *FPAdd_Compare* also saturates the shift amount since exponent differences greater than 25 result in identical mantissa alignments.

2) FPAdd_Align: This instruction is similar to *FPAdd_SatAlign*, except that alignment shift amount saturation is handled by *FPAdd_Compare*. This instruction reads the sorted operands and then aligns and adds them using the shift amount. The rest of the FP operation is performed using *LZD*, *BShiftL*, and *FPAdd_Round*.

D. Addition/Subtraction Hybrid Implementation With CFP Hardware Version 4 (Hybrid Add/Sub w/ CFP Ver. 4)

This implementation sorts the operands and calculates the hidden bit, sticky bit, and saturated exponent difference using fixed-point instructions. *FPAdd_Align* aligns mantissas, potentially complements one of them, and adds them together. *Shift_LZA* replaces *BShiftL* and *LZD* and is described as follows.

Algorithm 1 Pseudocode of Hybrid Add/Sub w/ CFP Ver. 3

```
while true do
    $FPReg1_{[31:16]} \leftarrow Input$                 \\ Load Operands
    $FPReg1_{[15:0]} \leftarrow Input$
    $FPReg2_{[31:16]} \leftarrow Input$
    $FPReg2_{[15:0]} \leftarrow Input$
    FPAdd_Compare $ExpDiff$                     \\ Sort Operands
    FPAdd_Align $ExpDiff$                       \\ Align & Add
    $LowerZeros \leftarrow 0$                   \\ Initialize 0's count
    LZD $UpperZeros$                            \\ Count 0's in $FPReg3$
    $Temp \leftarrow UpperZeros - 16$
    if $Temp == 0$ then                         \\ Check if all 0's
        $FPReg3 \leftarrow FPReg1_{[31:16]}$
        LZD $LowerZeros$                        \\ Count 0's in $FPReg3$
    end if
    \\ Correct count and add counts together
    $UpperZeros \leftarrow UpperZeros - 4$
    $ShiftAmount \leftarrow UpperZeros + LowerZeros$
    BShiftL $ShiftAmount$                       \\ Normalize
    FPAdd_Round $Output$   \\ $Output \leftarrow FPReg2_{[31:16]}$
    $Output \leftarrow FPReg2_{[15:0]}$
end while
```

1) Shift_LZA: An LZA forms an indicator string to anticipate the leading zeros in parallel with the addition [1]. The leading zero count is then used for normalization shifting, followed by a possible 1-bit correction. The rest of the FP operation is performed using *FPAdd_Round*.

E. Multiplication Hybrid Implementation With CFP Hardware Version 1 (Hybrid Mult w/ CFP Ver. 1)

This implementation performs mantissa multiplication and exponent and sign bit calculation using fixed-point software instructions. *FPMult_NormRndCarry* performs the rest of the operation and is described as follows.

1) FPMult_NormRndCarry: Following mantissa multiplication and exponent calculation, the product is loaded into an FP register. The normalized and rounded mantissa is written back into an FP register, the 16 MSBs are returned, and the carry flag is set. If the carry flag is set, the exponent is incremented in software. Fig. 2(a) shows the hardware for adding this design into the execution stage of the target platform.

F. Multiplication Hybrid Implementation With CFP Hardware Version 2 (Hybrid Mult w/ CFP Ver. 2)

This implementation performs mantissa multiplication in software using fixed-point instructions. *FPMult_NormRnd* performs the rest of the operation and is described as follows.

1) FPMult_NormRnd: Following software mantissa multiplication, the product, exponent, and sign bits are loaded into FP registers. The new sign bit, exponent, and normalized and rounded product are then calculated. The result is written to the FP registers and selectable via a 16-bit multiplexer. Fig. 2(b) shows the hardware for this design.

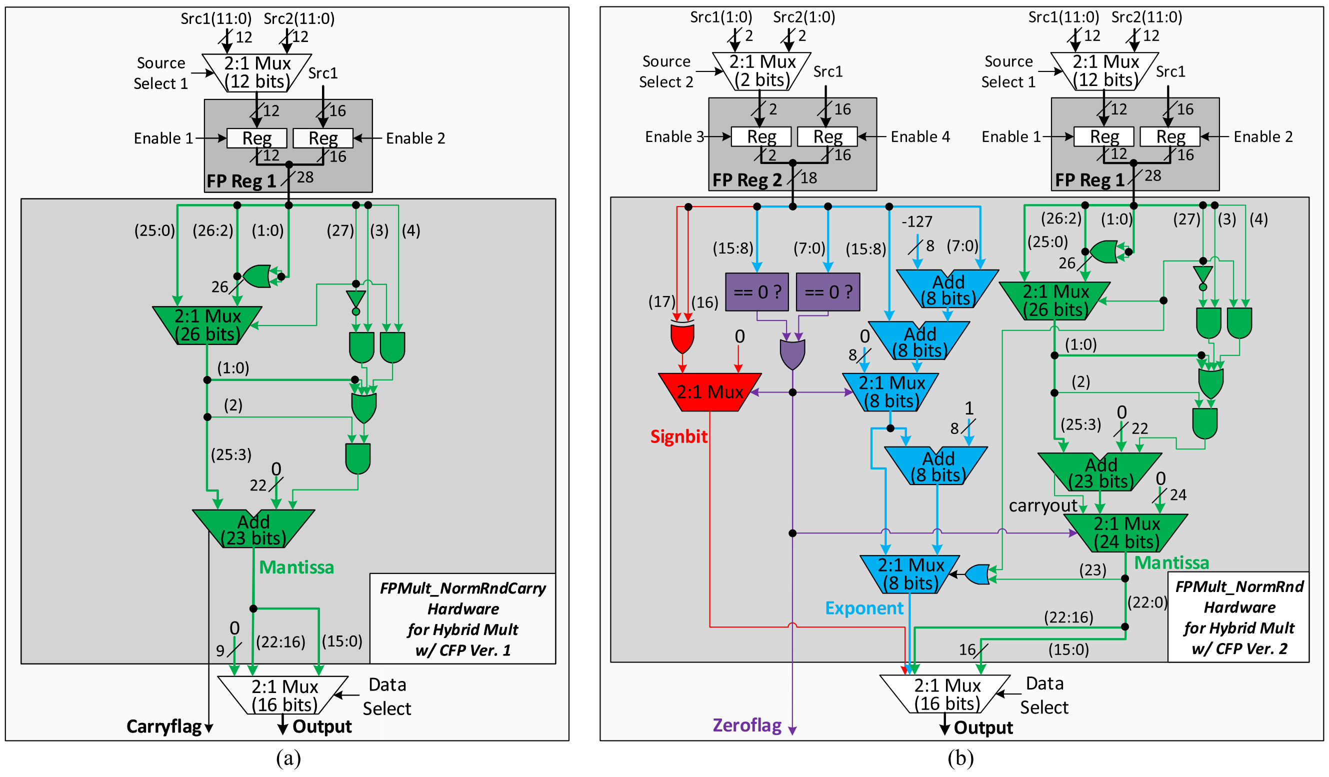


Fig. 2. (a) Hardware to implement the *FPMult_NormRndCarry* instruction for the *Hybrid Mult w/ CFP Ver. 1* implementation. FP Reg 1 is loaded with the product of the mantissa multiplication. The rounded result and carry bit are produced. If the carry flag is set, the exponent is incremented in software. (b) Hardware to implement the *FPMult_NormRnd* instruction for the *Hybrid Mult w/ CFP Ver. 2* implementation. FP Reg 1 is loaded with the product and FP Reg 2 is loaded with the sign bits and exponents of both operands. The sign, exponent, rounded result, and zero flag are then produced.

G. Division Hybrid Implementation With CFP Hardware Version 1 (Hybrid Div w/ CFP Ver. 1)

The nonrestoring division algorithm is used for performing FP division with this implementation [25]. The exponent and sign bit of the result are determined in software. The instruction described in the following performs the rest of the operation.

1) FPDiv_LoopExpAdj: After both inputs and the partially computed exponent are loaded into the FP registers, this instruction performs the division loop, described in Section II-B4, to calculate the final mantissa. The exponent is then adjusted in hardware following normalization and rounding.

H. Square Root Hybrid Implementation With CFP Hardware Version 1 (Hybrid Sqrt w/ CFP Ver. 1)

This implementation implements the nonrestoring square root algorithm [32]. The exponent of the result is determined in software. *FPSqrt_Loop* performs the rest of the operation.

1) FPSqrt_Loop: After loading the input into the FP register, this instruction performs the square root loop, described in Section II-B5, to calculate the final mantissa.

VII. Results and Comparisons

Each implementation is synthesized with a 65-nm CMOS standard cell library using Synopsys DC compiler with a 1.3 V operating voltage and 25 °C operating temperature and clock frequencies of 600, 800, 1000, and 1200 MHz.

For accuracy and performance analysis, FPgen [44], a test suite for verifying FP datapaths is used to include test cases unlikely to be covered by pure random test generation. This testing is supplemented by using millions of pseudorandomly generated FP values on the normalized value interval $\pm[2^{-126}, (2-2^{-23}) \times 2^{127}]$.

With the exception of the full software kernels, each design adds circuitry to the platform processor, the area for this circuitry is referred to as additional area.

A. Individual FP Designs Compared

Fig. 3 plots additional area versus delay for each design using four different target clock frequencies ranging from 600–1200 MHz. Additional area is plotted versus cycles per FLOP times the clock period in nanoseconds. The designs are plotted on separate graphs according to operation type. Since FMA supports both addition/subtraction and multiplication operations, it is plotted in both Fig. 3(a) and (b). Designs for both 16-bit and 32-bit word sizes and datapaths are plotted. The 32-bit I/O designs are smaller than their counterparts because they do not consider the additional area required for a processor with a 32-bit word size and datapath.

As expected, designs providing higher throughput generally require greater area. Regardless of clock period, the dedicated FMA requires the most area. Having a split multiplier and addition/subtraction design requires less area than an FMA

TABLE II
THROUGHPUT, INSTRUCTION COUNT, AND AREA FOR EACH DESIGN

	FP Design	Instruction Count (Static)	Throughput Per Core (MFLOPS)	Average Speedup (Per Core)	Cycles/FLOP (Per Core)	Additional Area (μm^2)	Area Increase (%)
Addition/Subtraction Designs	*Full SW Add/Sub*	222	9.70	1.00x	124	0	0
	Full HW Add/Sub	6	171	17.6x	7	13 621	8.10
	Full HW Add/Sub (32-bit I/O)	1	1200	123x	1	10 402	6.19
	Hybrid Add/Sub w/ USL	127	21.1	2.18x	57	3136	1.87
	Hybrid Add/Sub w/ CFP Ver. 1	40	35.8	3.69x	34	8740	5.20
	Hybrid Add/Sub w/ CFP Ver. 2	78	41.5	4.28x	29	7603	4.52
	Hybrid Add/Sub w/ CFP Ver. 3	**17**	**70.6**	**7.28x**	**17**	**10 821**	**6.44**
	Hybrid Add/Sub w/ CFP Ver. 4	31	44.1	4.55x	28	12 532	7.46
	Full HW FMA	8	150	15.5x	8	55 121	32.8
	Full HW FMA (32-bit I/O)	2	600	61.9x	2	51 771	30.8
Multiplication Designs	*Full SW Mult*	66	17.4	1.00x	69	0	0
	Full HW Mult	6	200	11.5x	6	18 189	10.8
	Full HW Mult (32-bit I/O)	1	1200	69.0x	1	17 258	10.3
	Hybrid Mult w/ USL	54	22.4	1.29x	54	3665	2.18
	Hybrid Mult w/ CFP Ver. 1	52	21.2	1.22x	57	1659	0.99
	Hybrid Mult w/ CFP Ver. 2	**34**	**35.3**	**2.03x**	**34**	**2596**	**1.54**
	Full HW FMA	8	150	8.62x	8	55 121	32.8
	Full HW FMA (32-bit I/O)	2	600	34.5x	2	51 771	30.8
Division Designs	*Full SW Div Ver. 1*	84	1.54	1.00x	777	0	0
	Full SW Div Ver. 2	1032	0.70	0.45x	1719	0	0
	Full HW Div	8	34.3	22.3x	35	6230	3.71
	Full HW Div (32-bit I/O)	3	40.0	26.0x	30	5985	3.56
	Hybrid Div w/ USL Ver. 1	**63**	**4.73**	**3.07x**	**254**	**2957**	**1.76**
	Hybrid Div w/ USL Ver. 2	125	6.24	4.05x	193	5138	3.06
	Hybrid Div w/ CFP Ver. 1	28	22.2	14.4x	54	5706	3.39
Square Root Designs	***Full SW Sqrt Ver. 1***	**114**	**0.80**	**1.00x**	**1500**	**0**	**0**
	Full SW Sqrt Ver. 2	1482	0.46	0.58x	2610	0	0
	Full HW Sqrt	8	37.5	46.9x	32	6553	3.90
	Full HW Sqrt (32-bit I/O)	3	46.2	57.8x	26	6772	4.03
	Hybrid Sqrt w/ USL Ver. 1	60	2.49	3.11x	481	5138	3.06
	Hybrid Sqrt w/ USL Ver. 2	214	3.05	3.81x	394	5138	3.06
	Hybrid Sqrt w/ CFP Ver. 1	20	25.5	31.9x	47	6148	3.66
FMA Designs	*Full HW FMA*	9	133	22.9x	9	55 121	32.8
	Full HW FMA (32-bit I/O)	2	600	103x	2	51 771	30.8

The optimal implementations subject to an area constraint from Fig. 4 are denoted in bold font.
FMA results reported separately for addition/subtraction, multiplication, and fused multiply-add operations.

due to extra circuitry present such as a wider alignment shifter, adder, and normalization shifter, as well as the LZA and end-around carry adder present in the FMA. A large area savings is not observed for most designs when using a longer clock period. In addition, the target platform utilizes a 1.2 GHz clock frequency and a separate clock is not available for the FP circuitry; therefore, the results for the rest of this paper consider a 1.2 GHz clock frequency.

Table II lists the throughput per core, instruction count, and area for each design. Fig. 4 plots the throughput and area for each design. Each implementation is plotted on a separate graph according to operation type. For each of the four plots, full software kernels are found on the left side, along the y-axis. Full hardware modules are located in the bottom right hand of each plot. The hybrid implementations (with USL support and with CFP hardware) are found in the middle of the plots.

We can determine the optimal design subject to an area constraint by selecting an implementation that uses less area than the constraint and requires the fewest average cycles per FLOP. As an example, we consider an area constraint A_{max} equal to 10% of the target platform processor area. For this example, more area is allocated for addition/subtraction and multiplication hardware because division and square root are less frequent operations [26], [27]. As shown in Fig. 4(a), an area constraint is first set for the addition/subtraction design $A_{70\%\ \text{max}}$ equal to 70% of the maximum area constraint. The optimal implementation that requires the least cycles per FLOP while not exceeding the area constraint is denoted by the green arrow as *Hybrid Add/Sub w/ CFP Ver.* 3. Using the remaining area, the area constraint $A_{\text{mult max}}$ is set in Fig. 4(b) and the optimal design for multiplication is *Hybrid Mult w/ CFP Ver. 2*. There remains available area, $A_{\text{div max}}$, for improving division throughput in Fig. 4(c) using *Hybrid Div w/ USL Ver. 1*. Finally, the optimal design for square root is determined in Fig. 4(d) to be a software kernel, *Full SW Sqrt Ver. 1*.

The full hardware designs require the most area and achieve the highest throughput; however, none of these implementations meets the area constraint and the FMA is the largest implementation, increasing processor area by 32.8%. Except for multiplication, the hybrid implementations with USL support require the least area to improve throughput. They can

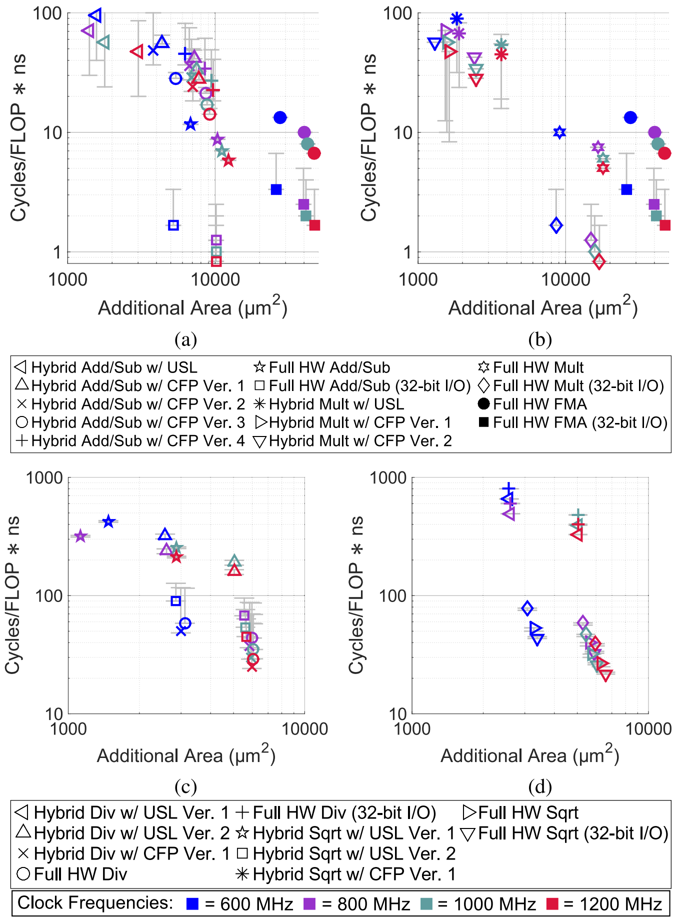


Fig. 3. Result of exploring different cycle times for different FP designs. The markers denote the average cycles per FLOP times the clock period, and the interval bar endpoints for each symbol denote the corresponding minimum and maximum. Cycles per FLOP are scaled by the number of cores required. Results are obtained from synthesis in 65-nm CMOS at 1.3 V and 600–1200 MHz. (a) Addition/subtraction designs. (b) Multiplication designs. (c) Division designs. (d) Square root designs.

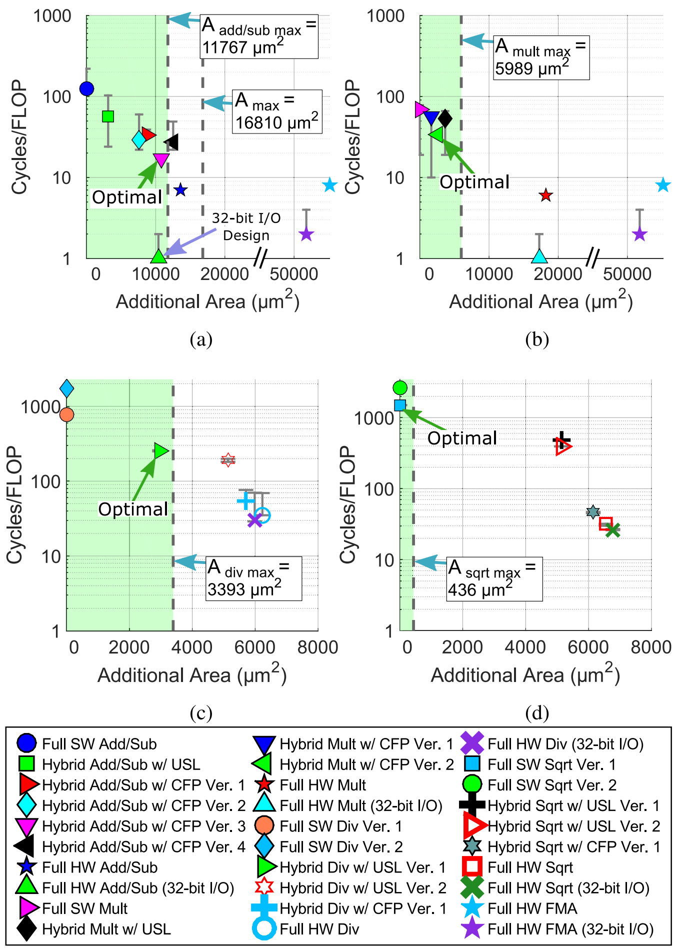


Fig. 4. Additional area versus cycles per FLOP for each FP design and determining the optimal designs from area constraints. The 30 markers in the legend denote the average cycles per FLOP, and the endpoints of the interval bars for each symbol denote the corresponding minimum and maximum. Cycles per FLOP are scaled by the number of cores required. Area constraints are indicated by the vertical dashed lines. The optimal design has the least average cycles per FLOP and an area below the area constraint. For this example, the area available for additional hardware $A_{\max}$ is equal to 10% of the processor area. (a) Optimal adder/subtractor is first determined using an area constraint of 70% of the maximum area constraint. (b)–(d) Optimal multiplication, division, and square root designs are determined using the remaining available area. Designs satisfying the area constraint appear in the green regions. Results are obtained from synthesis in 65-nm CMOS at 1.3 V and 1.2 GHz.

also be used for general-purpose workloads because the USL instructions are non-FP specific. For the full software kernels, division and square root require less cycles per FLOP using the long-division and digit-by-digit algorithms, respectively. However, the division and square root hybrid implementations with USL support require slightly less cycles per FLOP when using the Newton–Raphson algorithm.

B. Comparison When Combining FP Designs

To compare the throughput and area when combining multiple designs, the FP designs discussed in Sections III–VI are combined into 38 functionally equivalent FPU implementations consisting of an addition/subtraction and multiplication unit. These designs are evaluated for performing unfused multiply–add, and Newton–Raphson division and square root. These Newton–Raphson and FMA implementations of divide and square root are mapped in a pipelined fashion and loops are unrolled to potentially provide high throughput [45]. These implementations are compared with full software, full hardware, and hybrid designs using the long-division, digit-by-digit, nonrestoring, or Newton–Raphson algorithm.

Fig. 5 plots the cycles per FLOP for the unfused multiply–add operation versus additional area. Just as in Section VII-A, the optimal design subject to an area constraint can be easily determined. The optimal multiply–add design is first determined, followed by the optimal division and square root designs. For this example, the area constraint $A_{\max}$ is equal to 10% of the target platform processor area; however, 80% of the area constraint $A_{80\%\ \max}$ is allocated for the addition/subtraction and multiplication hardware. Based on the cycles per FLOP for performing the multiply–add operation, denoted by the green arrow, *Hybrid Add/Sub w/ CFP Ver. 3* is the optimal addition/subtraction design and *Hybrid Mult w/ CFP Ver. 2* is the optimal multiplication design. Despite offering reduced latency and higher throughput, the additional area overhead for the FMA does not meet the area constraint.

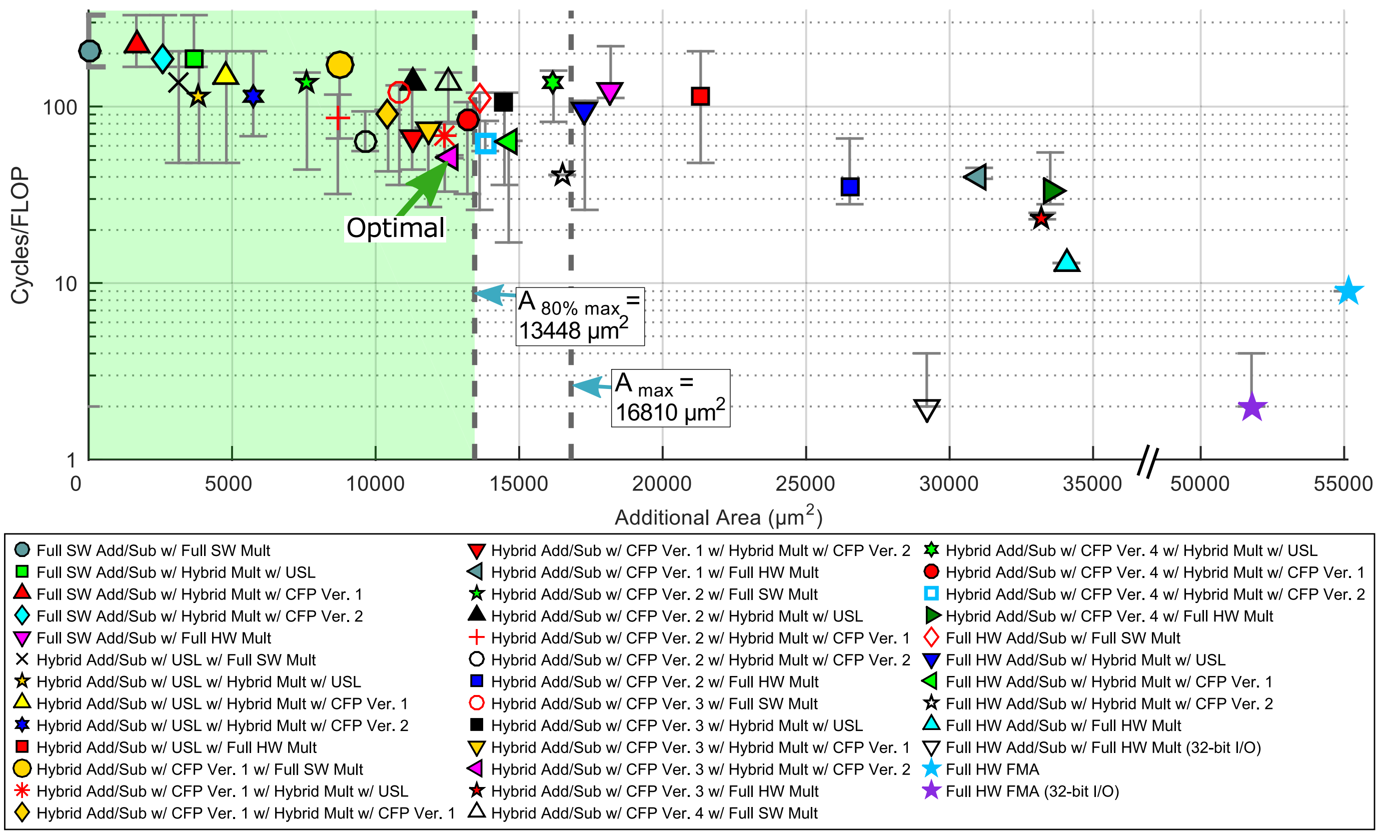


Fig. 5. Multiply–add (A + B × C) area versus cycles per FLOP for all FPU implementations and determining the optimal implementation from an area constraint. Design point symbols are placed at the average cycles/FLOP point with interval bars showing the range over all possible values. Cycles per FLOP are scaled by the number of cores required. Area constraints are indicated by the vertical dashed lines. The optimal design has the least average cycles per FLOP and an area below the area constraint. For this example, the area available for additional hardware, $A_{\max}$, equals 10% of the processor area. The optimal adder/subtractor and multiplier are first determined using 80% of the maximum area constraint $A_{80\%\ \max}$. Using the remaining available area, the optimal designs for division and square root are determined in Figs. 6 and 7, respectively. Designs satisfying the area constraint appear in the green highlighted region. Results are obtained from synthesis in 65-nm CMOS at 1.3 V and 1.2 GHz.

Fig. 6 plots the cycles per FLOP for the division operation versus additional area. The combinations of addition/subtraction and multiplication designs from Fig. 5 are used to perform Newton–Raphson division. Using one of the FPU implementations from Fig. 5 to implement division does not require any additional area other than that already incurred for the addition/subtraction and multiplication designs. Using the area left over from choosing a design in Fig. 5, the optimal design for improving division throughput is determined. The division implementation using the optimal FPU from Fig. 5 is denoted by the blue arrow; however, it does not improve throughput versus full software. Subject to the constraint $A_{\text{div max}}$, the optimal division design is *Hybrid Div w/ USL Ver. 1*, which uses the long-division algorithm. Scaled by core count, none of the addition/subtraction and multiplication combinations using Newton–Raphson division increases throughput.

Fig. 7 plots the cycles per FLOP for the square root operation versus additional area. The combinations of addition/subtraction and multiplication designs from Fig. 5 are used to perform Newton–Raphson square root. Using one of the FPU implementations from Fig. 5 to implement square root does not require any additional area other than that already incurred for the addition/subtraction and multiplication designs. $A_{\text{sqrt max}}$ is the area left over from choosing an addition/subtraction and multiplication design in Fig. 5 and a division design in Fig. 6 and is used to determine an optimal square root design. The square root implementation using the optimal FPU from Fig. 5 is denoted by the blue arrow; however, it achieves lower throughput than the software implementation. Subject to the area constraint, the optimal square root design is *Full SW Sqrt Ver. 1*, which implements the digit-by-digit algorithm. Contrary to division, some Newton–Raphson square root implementations using combinations of addition/subtraction and multiplication designs improve throughput over the full software implementation.

The FPU implementations are also evaluated for performing two scientific kernel benchmarks. Fig. 8(a) and (b) plots the cycles per FLOP for a radix-2 complex butterfly computation and a 2 × 2 matrix multiplication. These benchmarks are two examples of kernels in many scientific workloads [46]. They are implemented using the minimum number of cores and the addition/subtraction and multiplication designs from Fig. 5 and subject to the same area constraint. Similar tradeoffs between the designs are seen in Figs. 5 and 8. As denoted by the green arrow, *Hybrid Add/Sub w/ CFP Ver. 3* and *Hybrid Mult w/ CFP Ver. 2* remain the optimal addition/subtraction and multiplication designs, respectively. The full hardware designs

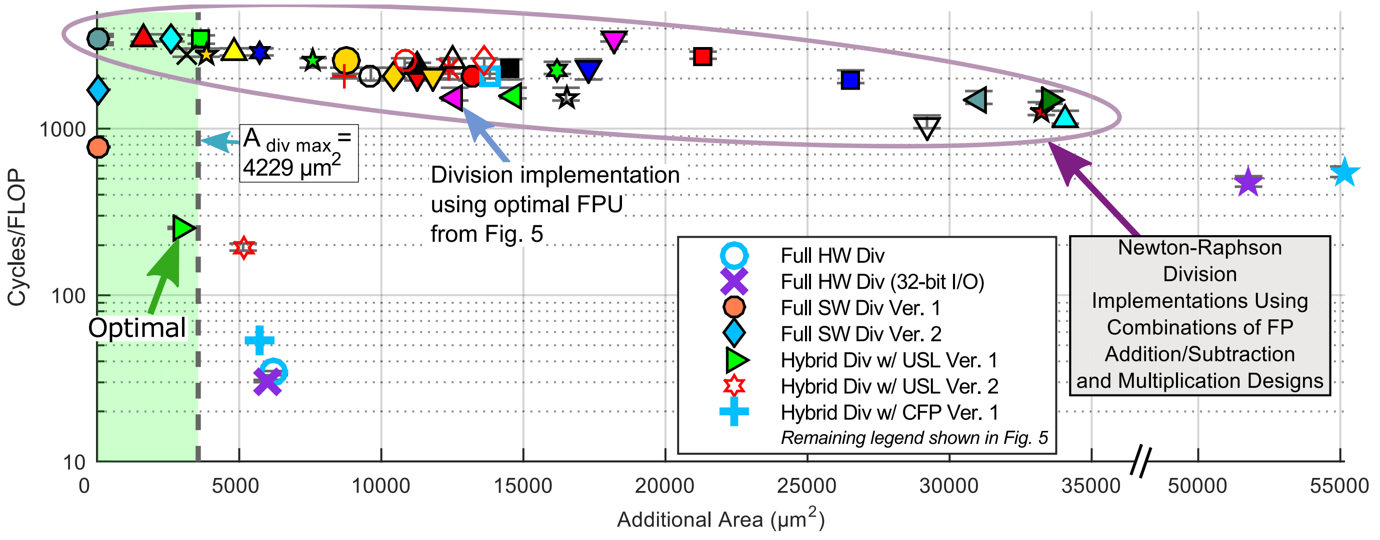


Fig. 6. Division area versus cycles per FLOP with all implementations and determining the optimal implementation from an area constraint. Four methods are evaluated for performing division: the Newton–Raphson method, division in software, division in hardware, and hybrid division. Design point symbols are placed at the average cycles/FLOP point with interval bars showing the range over all possible values. Cycles per FLOP are scaled by the number of cores required. The optimal division implementation is determined with the remaining available area, $A_{\text{div max}}$. Designs satisfying the area constraint appear in the green highlighted region. Using the remaining available area, the optimal design for square root is determined in Fig. 7. The remaining legend is shown in Fig. 5. Results from synthesis in 65-nm CMOS at 1.3 V and 1.2 GHz.

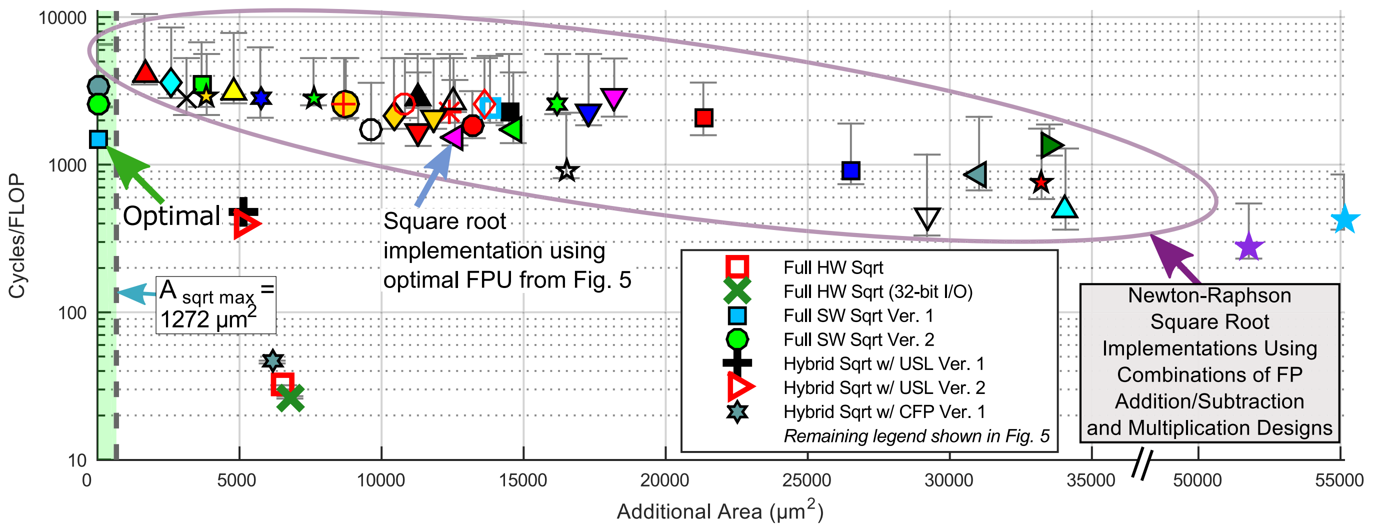


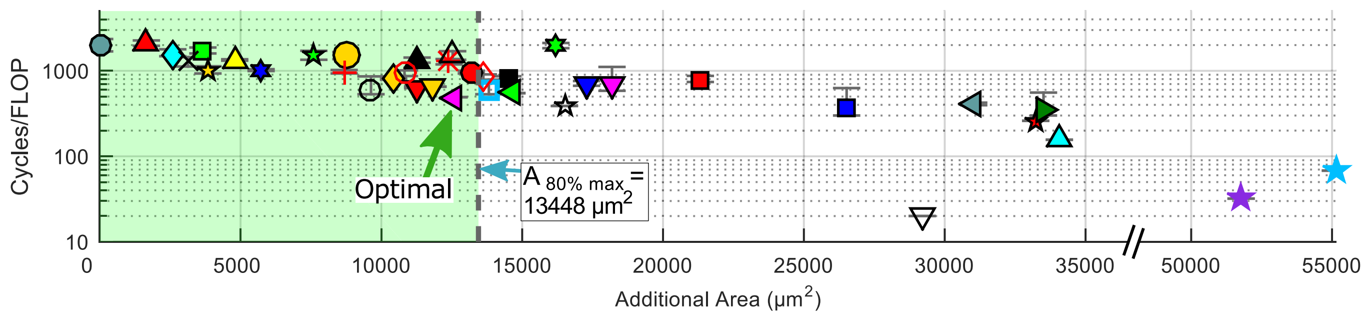
Fig. 7. Square root area versus cycles per FLOP with all implementations and determining the optimal implementation from an area constraint. Four methods are evaluated for performing square root: the Newton–Raphson method, square root in software, square root in hardware, and hybrid square root. Design point symbols are placed at the average cycles/FLOP point with interval bars showing the range over all possible values. Cycles per FLOP are scaled by the number of cores required. The optimal square root implementation is determined with the remaining available area, $A_{\text{sqrt max}}$. Designs satisfying the area constraint appear in the green highlighted region. Remaining legend shown in Fig. 5. Results from synthesis in 65-nm CMOS at 1.3 V and 1.2 GHz.

provide the highest throughput but do not meet the area constraint. The policy of using cycles per FLOP and additional area for each FP design from Figs. 4–7 can be employed to roughly estimate the performance and area requirements for computing other benchmarks.

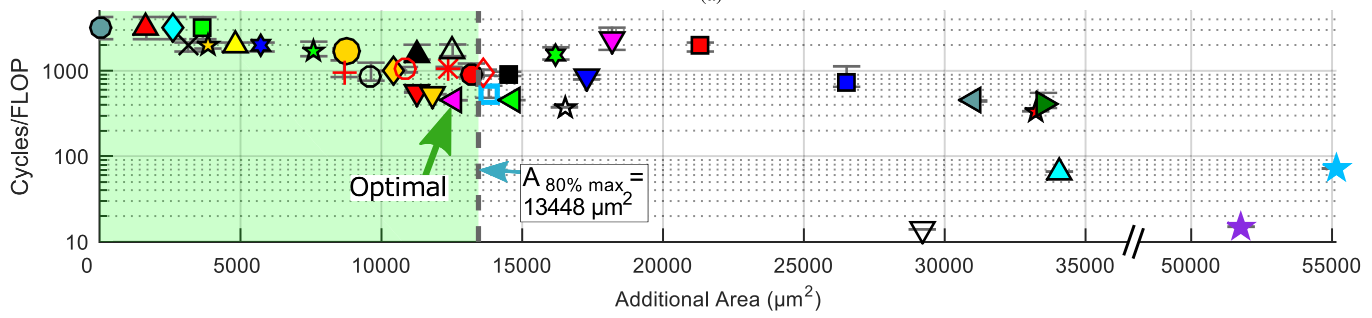
VIII. Advantages of Hybrid Approaches

When area cannot be increased, software implementations are the only option for performing FP arithmetic. Dedicated hardware designs are ideal when the goal is maximum throughput. When area is constrained, hybrid designs are optimal because they increase throughput and require less area than dedicated FP hardware. They provide a method for satisfying an area constraint that dedicated hardware would violate. Hybrid implementations with USL support increase throughput and reduce area overhead by adding functionality to existing hardware to simplify multiword operations. The hybrid implementations with CFP exceed the performance of the USL support designs by adding custom hardware which performs specific steps of an FP operation. These steps would otherwise require many fixed-point instructions.

The full software implementations require many operations on large (multiword) data values. Multiword operations require carrying between words or summing and carrying between



(a)



(b)

Fig. 8. Benchmark results for two scientific application kernels. (a) Benchmark results for calculating a 2×2 matrix multiplication. (b) Benchmark results for computing a radix-2 complex butterfly operation. Design point symbols are placed at the average cycles/FLOP point with interval bars showing the range over all possible values. Cycles per FLOP are scaled by the number of cores required. Area constraints are indicated by the vertical dashed lines. The optimal design has the least average cycles per FLOP and an area below the area constraint. The optimal adder/subtractor and multiplier are determined using the same area constraint as Fig. 5, $A_{80\%\ \max}$. Designs satisfying the area constraint appear in the green highlighted region. The legend is shown in Fig. 5. Results from synthesis in 65-nm CMOS at 1.3 V and 1.2 GHz.

partial products. The programmer must avoid using the bit that is treated as signed (bit 16 for the target platform) and must handle carry flags and partial product summation in software. Therefore, signed hardware cannot operate on completely utilized 16-bit words. Words must be partitioned into 15 bits each at most. The hybrid implementations with USL support provide unsigned hardware, which allows efficient handling of multiword values, improving Newton–Raphson throughput. The long-division and digit-by-digit methods see much less benefit, as they depend more on shifts.

Multiple hybrid designs with CFP are implemented to explore the benefits of different design approaches. Each version differs in terms of which steps or the proportion of the FP operation that is performed in software. Which steps justify hardware support is based on the throughput increase and area overhead. *Hybrid Add/Sub w/ CFP Ver. 3* increases addition/subtraction throughput the most by supporting operand comparison in hardware. Otherwise, sorting the operands requires many instructions to compare the exponents and the multiword mantissa. *Hybrid Add/Sub w/ CFP Ver. 4* includes an LZA, which increases throughput, but requires more area and improves throughput less than supporting operand sorting in hardware. For multiplication, *Hybrid Mult w/ CFP Ver. 2* increases throughput the most by adding more hardware support than *Ver. 1*. This implementation reduces the executed instruction count by calculating the sign bit and exponent and determining if the result is zero in hardware. This additional circuitry increases area and is shown in Fig. 2. The division and square root implementations use less area than dedicated FP hardware by performing sign bit and exponent calculation in software; the throughput of these operations is increased by performing the rest of the operations in hardware.

IX. Related Work and Comparison

Since this work presents single-precision FP implementations, we compare our results with other work that increase single-precision FP throughput with less area overhead than a dedicated hardware design and do not compare with implementations using BFP or a reduced FP word width. Table III summarizes a comparison with other methods for improving FP throughput. Our results include designs with a 16-bit and 32-bit word size and datapath, all implementing single-precision FP. Not every work reports area data; therefore, to make a consistent comparison, the area overhead of each design is evaluated against the area of the dedicated full hardware design reported in that respective work. This paper and two of the papers in Table III explore alternatives to an FMA [14], [15], while one compares against an Altera FPU without divide [21]. This paper reports the area overhead for supporting each FP operation individually. Other work do not publish area for specific operations; therefore, area is recorded under the FP operation categories for which cycle counts are reported. Except for the FMA design, our implementations for multiply–add perform an unfused operation.

The work by Gilani *et al.* [14] and Viitanen *et al.* [15] did not explore modular designs. Hockert and Compton [21] explored modular designs with varying amounts of hardware support, but did not evaluate the overhead for supporting individual FP operations.

Our work presents a wider range of area overheads for improving FP throughput, allowing more versatility across a

TABLE III

COMPARISON OF METHODS FOR IMPROVING FP THROUGHPUT WITH LESS OVERHEAD

	Clock Freq (MHz)	Process Node (nm)	Area Overhead Compared to Full HW Design*(%)					Cycles/FLOP				
			Multiply-Add	Add /Sub	Multiply	Divide	Square Root	Multiply-Add	Add /Sub	Multiply	Divide	Square Root
Gilani et al. [14]	1000	65	19.2†			N/A	N/A	6	5	5	N/A	N/A
Hockert and Compton [21]	500	65	N/A	22.8–80.0†				N/A	77–118	109	218–238	289–313
Viitanen et al. [15]	300	110	N/A	N/A	N/A	13.3†		N/A	N/A	N/A	49	43
This Work‡ (16-bit I/O)	**1200**	65	**4.71**–61.8	**5.69**–24.7	**3.01**–33.0	**5.36**–11.3	**9.32**–61.8	13–186	7–57	6–57	**35**–254	**32**–1373
This Work‡ (32-bit I/O)	**1200**	65	56.4	20.1	33.3	**11.6**–56.4	**13.1**–56.4	**2**	**1**	**1**	**30**–1053	**26**–447

Results reported for alternatives to baseline implementations. Cycles per FLOP are scaled by the number of cores required.
This paper provides more options for improving FP throughput and a wider range of area overheads than previous work.
* Area overhead relative to FMA area reported in [14], [15], and this paper. Area overhead relative to Altera FPU without divide in [21].
† Total area to perform these FP operations reported.
‡ Results from synthesis in 65 nm CMOS at 1.3 V and 1.2 GHz.

large range of area constraints. Our designs also offer the lowest cycles per FLOP for both the divide and square root operations while requiring less area than an FMA. Comparing our 32-bit I/O designs with other work that reduce FP area overhead compared with dedicated FP hardware, our implementations achieve the lowest cycles per FLOP for all operation types.

X. CONCLUSION

In this paper, eight hybrid implementations with CFP hardware and six hybrid implementations with USL support are presented for a fixed-point processor. These implementations increase the throughput of FP operations by adding USL support instructions to the ISA, and custom FP instructions. The area overhead is kept low by utilizing the existing fixed-point functional units.

The circuit area and throughput are found for 38 multiply–add, 8 addition/subtraction, 6 multiplication, 45 division, and 45 square root designs. This paper presents designs that improve FP throughput versus a baseline software implementation and require less area overhead compared with an FMA than other works. Several examples demonstrate how to determine the optimal FP designs for a given area constraint. Hybrid implementations are an effective design method for increasing FP throughput and require up to 97.0% less area than a traditional FMA.

ACKNOWLEDGMENT

The authors would like to thank STMicroelectronics for donating the chip fabrication.

REFERENCES

[1] J.-M. Muller *et al.*, *Handbook of Floating-Point Arithmetic*, 1st ed. Basel, Switzerland: Birkhäuser, 2009.

[2] S. Z. Gilani, N. S. Kim, and M. Schulte, "Energy-efficient floating-point arithmetic for software-defined radio architectures," in *Proc. IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Sep. 2011, pp. 122–129.

[3] S. M. Shajedul Hasan and S. W. Ellingson, "An investigation of the Blackfin/uClinux combination as a candidate software radio processor," Dept. Elect. Comput. Eng., Virginia Polytechn. Inst. State Univ., Blacksburg, VA, USA, Tech. Rep. 2, 2006.

[4] *Floating Point Arithmetic on the PicoArray*, accessed on Jun. 5, 2015. [Online]. Available: https://support.picochip.com/picochip-resource-folder/Nexu5utu/4598jf4897f/floatingpoint.pdf/download

[5] C. Iordache and P. T. P. Tang, "An overview of floating-point support and math library on the Intel XScale architecture," in *Proc. 16th IEEE Symp. Comput. Arithmetic*, Jun. 2003, pp. 122–128.

[6] Z. Yu *et al.*, "AsAP: An asynchronous array of simple processors," *IEEE J. Solid-State Circuits*, vol. 43, no. 3, pp. 695–705, Mar. 2008.

[7] D. N. Truong *et al.*, "A 167-processor computational platform in 65 nm CMOS," *IEEE J. Solid-State Circuits*, vol. 44, no. 4, pp. 1130–1144, Apr. 2009.

[8] Y. J. Chong and S. Parameswaran, "Configurable multimode embedded floating-point units for FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 11, pp. 2033–2044, Nov. 2011.

[9] *The Industry's First Floating-Point FPGA. Altera*, accessed on Jun. 5, 2015. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/po/bg-floating-point-fpga.pdf

[10] S. Boldo and J.-M. Muller, "Exact and approximated error of the FMA," *IEEE Trans. Comput.*, vol. 60, no. 2, pp. 157–164, Feb. 2011.

[11] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 913–922, Jul. 2011.

[12] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert, "Architectural modifications to enhance the floating-point performance of FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 2, pp. 177–187, Feb. 2008.

[13] K. Kalliojarvi and J. Astola, "Roundoff errors in block-floating-point systems," *IEEE Trans. Signal Process.*, vol. 44, no. 4, pp. 783–790, Apr. 1996.

[14] S. Z. Gilani, N. S. Kim, and M. Schulte, "Virtual floating-point units for low-power embedded processors," in *Proc. IEEE 23rd Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2012, pp. 61–68.

[15] T. Viitanen, P. Jääskeläinen, and J. Takala, "Inexpensive correctly rounded floating-point division and square root with input scaling," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, Oct. 2013, pp. 159–164.

[16] F. Fang, T. Chen, and R. A. Rutenbar, "Lightweight floating-point arithmetic: Case study of inverse discrete cosine transform," *EURASIP J. Appl. Signal Process.*, vol. 2002, no. 1, pp. 879–892, Jan. 2002.

[17] S.-W. Lee and I.-C. Park, "Low cost floating-point unit design for audio applications," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, vol. 1. 2002, pp. I-869–I-872.

[18] J. J. Pimentel, A. Stillmaker, B. Bohnenstiehl, and B. M. Baas, "Area efficient backprojection computation with reduced floating-point word width for SAR image formation," in *Proc. 49th Asilomar Conf. Signals, Syst. Comput.*, Nov. 2015, pp. 726–732.

[19] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 3, pp. 273–286, Jun. 2000.

[20] H.-J. Oh *et al.*, "A fully pipelined single-precision floating-point unit in the synergistic processor element of a CELL processor," *IEEE J. Solid-State Circuits*, vol. 41, no. 4, pp. 759–771, Apr. 2006.

[21] N. Hockert and K. Compton, “Improving floating-point performance in less area: Fractured floating point units (FFPUs),” *J. Signal Process. Syst.*, vol. 67, no. 1, pp. 31–46, Apr. 2012.

[22] *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2008, 2008.

[23] D. R. Lutz and C. N. Hinds, “Novel rounding techniques on the NEON floating-point pipeline,” in *Proc. 39th Asilomar Conf. Signals, Syst. Comput.*, Oct./Nov. 2005, pp. 1342–1346.

[24] A. H. Karp and P. Markstein, “High-precision division and square root,” *ACM Trans. Math. Softw.*, vol. 23, no. 4, pp. 561–589, Dec. 1997.

[25] M.-B. Lin, *Digital System Designs and Practices: Using Verilog HDL and FPGAs*. New York, NY, USA: Wiley, 2008.

[26] S. F. Oberman and M. J. Flynn, “Design issues in division and other floating-point operations,” *IEEE Trans. Comput.*, vol. 46, no. 2, pp. 154–161, Feb. 1997.

[27] Z. Jin, R. N. Pittman, and A. Forin, “Reconfigurable custom floating-point instructions,” Microsoft Res., Tech. Rep. MSR-TR-2009-157, Aug. 2009.

[28] S. F. Oberman and M. Flynn, “Division algorithms and implementations,” *IEEE Trans. Comput.*, vol. 46, no. 8, pp. 833–854, Aug. 1997.

[29] H. Nikmehr, “Architectures for floating-point division,” Ph.D. dissertation, School Elect. Electron. Eng., Univ. Adelaide, Adelaide, SA, Australia, 2005.

[30] N. Takagi, S. Kadowaki, and K. Takagi, “A hardware algorithm for integer division,” in *Proc. 17th IEEE Symp. Comput. Arithmetic (ARITH)*, Jun. 2005, pp. 140–146.

[31] M. J. Schulte, J. Omar, and E. E. Swartzlander, Jr., “Optimal initial approximations for the Newton–Raphson division algorithm,” *Computing*, vol. 53, nos. 3–4, pp. 233–242, Sep. 1994.

[32] Y. Li and W. Chu, “Implementation of single precision floating point square root on FPGAs,” in *Proc. 5th Annu. IEEE Symp. Field-Program. Custom Comput. Mach.*, Apr. 1997, pp. 226–232.

[33] P. Soderquist and M. Leeser, “Division and square root: Choosing the right implementation,” *IEEE Micro*, vol. 17, no. 4, pp. 56–66, Jul. 1997.

[34] P. Montuschi and M. Mezzalama, “Optimal absolute error starting values for Newton–Raphson calculation of square root,” *Computing*, vol. 46, no. 1, pp. 67–86, Mar. 1991.

[35] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*. Englewood Cliffs, NJ, USA: Prentice-Hall, 2000.

[36] D. Truong *et al.*, “A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling,” in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2008, pp. 22–23.

[37] Z. Xiao and B. M. Baas, “A 1080p H.264/AVC baseline residual encoder for a fine-grained many-core system,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 21, no. 7, pp. 890–902, Jul. 2011.

[38] Z. Xiao and B. Baas, “A high-performance parallel CAVLC encoder on a fine-grained many-core system,” in *Proc. 26th IEEE Int. Conf. Comput. Design (ICCD)*, Oct. 2008, pp. 248–254.

[39] D. N. Truong and B. M. Baas, “Massively parallel processor array for mid-/back-end ultrasound signal processing,” in *Proc. IEEE Biomed. Circuits Syst. Conf. (BioCAS)*, Nov. 2010, pp. 274–277.

[40] R. K. Montoye, E. Hokenek, and S. L. Runyon, “Design of the IBM RISC system/6000 floating-point execution unit,” *IBM J. Res. Develop.*, vol. 34, no. 1, pp. 59–70, Jan. 1990.

[41] E. Hokenek and R. K. Montoye, “Leading-zero anticipator (LZA) in the IBM RISC system/6000 floating-point execution unit,” *IBM J. Res. Develop.*, vol. 34, no. 1, pp. 71–77, Jan. 1990.

[42] V. G. Oklobdzija and R. K. Krishnamurthy, Eds., *High-Performance Energy-Efficient Microprocessor Design* (Integrated Circuits and Systems), 2006th ed. Berlin, Germany: Springer, Aug. 2006.

[43] J. J. Pimentel and B. M. Baas, “Hybrid floating-point modules with low area overhead on a fine-grained processing core,” in *Proc. 48th Asilomar Conf. Signals, Syst. Comput.*, Nov. 2014, pp. 1829–1833.

[44] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, “FPgen—A test generation framework for datapath floating-point verification,” in *Proc. 8th IEEE Int. High-Level Design Validation Test Workshop*, Nov. 2003, pp. 17–22.

[45] S. F. Oberman and M. J. Flynn, “An analysis of division algorithms and implementations,” Dept. Elect. Eng. Comput. Sci., Stanford Univ., Stanford, CA, USA, Tech. Rep. CSL-TR-95-675, 1995.

[46] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, “Scientific computing kernels on the cell processor,” *Int. J. Parallel Program.*, vol. 35, no. 3, pp. 263–298, Jun. 2007.

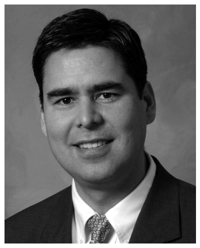
Jon J. Pimentel (S’07) received the B.S. degree in electrical engineering in 2009, and the M.S. degree in electrical and computer engineering from the University of California at Davis (UCD), Davis, CA, USA, in 2015, where he is currently pursuing the Ph.D. degree in electrical and computer engineering.

He has been a Graduate Student Researcher with the VLSI Computation Laboratory, Davis, since 2009. In 2013, he was an Intern with the Many Integrated Core Group, Intel, Hillsboro, OR, USA. His current research interests include floating-point architectures, VLSI design, synthetic aperture radar imaging, and many-core processor architecture.

Mr. Pimentel has been a GAANN Fellow since 2009. He received the Graduate Research Mentorship Fellowship in 2011, the UCD and Humanities Graduate Research Award in 2012 and 2014, the Frank and Carolan Walker Fellowship in 2012 and 2014, the George S. and Marjorie Butler Fellowship in 2014, the ECEGP Fellowship in 2014 and 2015, the ECE TA Program Support Fellowship in 2015, and the Herbert Tryon Fellowship and Laura Perrot Mahan Fellowship in 2016. He also received the Third Place Best Student Paper at Asilomar 2014.

Brent Bohnenstiehl (S’15) received the B.S. degree in electrical engineering from the University of California at Davis, Davis, CA, USA, in 2006, where he is currently pursuing the Ph.D. degree in electrical and computer engineering.

He has been a Graduate Student Researcher with the VLSI Computation Laboratory, Davis, since 2011. His current research interests include processor architecture, VLSI design, hardware–software codesign, DVFS algorithms, and many-core simulation tools.



Bevan M. Baas (M’95–SM’11) received the B.S. degree in electronics engineering from California Polytechnic State University, San Luis Obispo, CA, USA, in 1987, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 1990 and 1999, respectively.

He was with Hewlett-Packard, Cupertino, CA, USA, from 1987 to 1989, where he participated in the development of the processor for a high-end minicomputer. In 1999, he joined Atheros Communications, Santa Clara, CA, USA, where he served as an early employee and served as a core member of the team which developed the first IEEE 802.11a (54 Mbps, 5 GHz) Wi-Fi wireless LAN solution. In 2003, he joined the Department of Electrical and Computer Engineering, University of California at Davis, Davis, CA, USA, where he is currently an Associate Professor. He leads projects in architecture, hardware, software tools, and applications for VLSI computation with an emphasis on DSP workloads. Notable projects include the 36-processor Asynchronous Array of simple Processors (AsAP) chip, applications, and tools; a second generation 167-processor chip; low density parity check decoders; FFT processors; viterbi decoders; and H.264 video codecs. In 2006, he was a Visiting Professor with the Circuit Research Laboratory, Intel, Hillsboro, OR, USA.

Dr. Baas was a National Science Foundation Fellow from 1990 to 1993 and an NASA Graduate Student Researcher Fellow from 1993 to 1996. He was a recipient of the National Science Foundation CAREER Award in 2006 and the Most Promising Engineer/Scientist Award by AISES in 2006. He received the best paper award at ICCD 2011, the Third Place Best Student Paper at Asilomar 2014, and Best Student Paper Nominations at Asilomar 2011 and BioCAS 2010. He also supervised the research that received the Best Doctoral Dissertation Honorable Mention in 2013. From 2007 to 2012, he was an Associate Editor of the IEEE JOURNAL OF SOLID-STATE CIRCUITS and an IEEE Micro Guest Editor in 2012. He was the Program Committee Co-Chair of HotChips in 2011, and a Program Committee Member of Hotchips from 2009 to 2010, of ICCD from 2004 to 2005 and from 2007 to 2009, of ASYNC in 2010, and of the ISSCC SRP Forum in 2012.