Hybrid Floating-Point Modules with Low Area Overhead on a Fine-Grained Processing Core

Jon J. Pimentel and Bevan M. Baas
Department of Electrical and Computer Engineering
University of California, Davis
{jjpimentel, bbaas}@ucdavis.edu

***Abstract*—This paper proposes Hybrid Floating-Point Modules (HFPMs) as a method to improve software floating-point (FP) throughput without incurring the area overhead of hardware floating-point units (FPUs). The proposed HFPMs were synthesized in 65 nm CMOS. They increase throughput over a fixed-point software FP implementation by 3.6× for addition/subtraction, 2.3× for multiplication, and require less area than hardware modules. Nine functionally equivalent FPU implementations using combinations of software, hardware, and hybrid modules are synthesized and provide 1.07-3.34× higher throughput than a software FPU implementation, while requiring 1.08-12.5× less area than a hardware FPU for multiply-add operations.**

I. Introduction

Floating-point (FP) arithmetic is the most commonly used method for real number representation in modern computers [1]. However, some architectures are limited to fixed-point arithmetic due to the large area and power requirements of FP [2], [3]. Therefore, it remains desirable to increase the FP throughput provided by a software implementation without incurring the area overhead of a full hardware floating-point unit (FPU).

Many techniques for achieving lower overhead and increasing FP throughput have been introduced. Fused and cascade multiply-add FPUs improve accuracy and provide computational speedup, however, they introduce large area and power overheads that are undesirable for simple fixed-point processors [4]. Block FP (BFP) is useful for increasing computational accuracy, however it is most effective for handling blocks of data with similar magnitudes over an entire block [5]. Virtual FPUs were presented which utilize microoperations and custom instructions, however, the architecture presented is not similar to our platform and area data was not published for comparison [6]. Some architectures reduce the exponent and mantissa widths [7], and others develop a reduced FP format [8]. Although certain applications such as speech recognition and image processing don't require the full mantissa width, these techniques require accuracy analysis for their particular applications [9]. However, certain elements of the IEEE-754 standard can be removed for multimedia applications, as in the CELL processor, including rounding, exceptions, and denormal number handling to increase performance [10]. Hockert et al. utilized custom FP instructions with fractured floating-point units to increase FP throughput with lower area overhead than a full hardware FPU [11]. However, this work did not consider modular FPUs nor the throughput when performing the multiply-add operation. Additionally, this work focused on adding FP support on a Nios II embedded processor on a FPGA rather than on a simple processor with a 16-bit fixed-point datapath.

To simultaneously take advantage of the low area overhead of fixed-point software implementations and the high throughput of FP hardware acceleration, Hybrid Floating-Point Modules (HFPMs) are presented. These modules perform FP arithmetic on a simple fixed-point processor using a combination of fixed-point instructions and custom FP instructions. Area overhead is reduced by reusing the existing fixed point hardware, such as the multiplier and adder. Throughput is increased by replacing long sections of serial code with custom FP instructions to perform the same operation in fewer cycles. As portrayed in Fig. 1, hybrid modules provide higher throughput than full software FP modules and require less area than conventional, or full hardware FP modules.

Full Software
Low Area
Low Throughput
Hybrid
Area-Throughput Tradeoff
Full Hardware
High Area
High Throughput

Fig. 1. Hybrid modules offer alternatives to pure software and pure hardware designs to balance the area-throughput tradeoff.

The remainder of this paper is organized as follows: Section II presents the FP format and arithmetic. Section III covers the targeted many-core architecture. Section IV discusses the full software FP implementations which perform FP arithmetic using only fixed-point instructions. Section V presents the full hardware FP modules which contain full hardware support to perform FP arithmetic in a single instruction. Section VI discusses the proposed hybrid modules and presents three example modules, two for addition/subtraction, and one for multiplication. Sections VII and VIII compare the FP modules and FPU implementations. Section IX concludes the paper.

II. Floating-Point Format and Arithmetic

All FP modules presented utilize the IEEE-754 binary32 format (single precision) and inputs are restricted to the normalized value interval $\pm[2^{-126}, (2-2^{-23})\times 2^{127}]$ [12]. Exception handling, NaNs, ±Inf, and denormal values are not supported, and rounding is performed using the IEEE standard's default rounding mode, round to nearest even, in order to reduce overhead and because many multimedia applications do not rely on them [10].

Addition/subtraction is performed by comparing the exponents and mantissas to determine the smaller magnitude input operand. The second operand's sign is inverted when performing FP subtraction. The result sign is determined by the sign of the larger operand. The smaller operand's mantissa is right shifted by the exponent difference to align mantissas. Effective subtraction requires complementing and adding one to the smaller operand's mantissa. The mantissas are then added, the sum is normalized, and the result is rounded.

For multiplication, the sign of the result is the XOR of the operand signs. The result exponent is calculated by adding the operand exponents and subtracting the extra bias of 127. The result is formed by multiplying the operand mantissas, right shifting for normalization, and then rounding.

All proposed architectures are synthesized with a 65 nm CMOS standard cell library using Synopsys DC Compiler configured to 1.3 V and 25°C with a 1.2 GHz clock frequency. For validation and performance analysis, FPgen, a FP test-suite is used to include special cases unlikely to be covered by pure random test generation such as including all possible values for a shift between input operands for addition/subtraction operations [13]. Additionally, testing is supplemented by using pseudorandomly generated FP values on the normalized value interval $\pm[2^{-126}, (2-2^{-23}) \times 2^{127}]$.

III. Targeted Many-Core Architecture

The AsAP2 architecture is an example of a fine-grained many-core system [14], capable of computing complex DSP application workloads such as audio and video processing [15]. It features 164 simple programmable processors each with a six-stage pipeline, 16-bit fixed-point arithmetic logic unit (ALU) and multiplier, and 40-bit accumulator. Calculations are performed using two's complement arithmetic. Each processor occupies 0.17 mm^2 in 65 nm CMOS technology, can operate at a maximum clock frequency of 1.2 GHz at 1.3 V, and includes no specialized instructions [3]. Additionally, each programmable processor contains a 128×35-bit instruction memory, 128×16-bit data memory, and two 64×16-bit dual-clock FIFOs for inter-processor communication. Although this platform includes a BFP unit, it operates on 16-bit words and, therefore, provides less precision than a 32-bit FP design. Several methods for performing floating point operations on this platform are considered next.

IV. Full Software FP Modules

To enable the target platform to perform FP arithmetic, a FP software library is created in AsAP assembly. This library consists of an addition/subtraction and multiplication software module. These modules emulate FP hardware by executing fixed-point operations on the platform's integer ALU. They are referred to as "full software" because they only use fixed-point instructions and do not include any custom FP instructions. Due to the width of the input bus, FP values are sent on chip as two 16-bit words. When using the software modules each FP value is first split into four 16-bit words, consisting of the sign, exponent, the 12 most significant mantissa bits (with an explicit hidden bit), and the 12 least significant mantissa bits. Splitting each FP value across four words allows fixed-point operations without affecting adjacent bits.

The full software modules have large instruction counts for their programs due to the number of comparisons, the splitting of the FP value across four words, and the lack of unsigned ALU instructions. As a result, the FP addition/subtraction software module requires two cores for sufficient instruction memory. The bottlenecks for software FP are due to operand comparison, mantissa alignment and addition, normalization, and rounding. It is desirable to increase the throughput of these modules while keeping the area overhead small.

A. Multiplication Module (Full SW Mult)

The software FP multiplication module requires 80 instructions and achieves a throughput of 15.6 MFLOPS and throughput per area of 92.69 MFLOPS/mm^2.

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

The software FP add/sub module requires 216 instructions, therefore necessitating two cores for sufficient instruction memory. Although it achieves a throughput of 19.7 MFLOPS, the large area overhead reduces the throughput per area to 58.47 MFLOPS/mm^2.

V. Full Hardware FP Modules

To determine the throughput and area achievable for this platform, two full hardware FP modules are implemented, an addition/subtraction and multiplication module. Both operate without any FP emulation. They are referred to as "full hardware" because all FP arithmetic is done with FP hardware using a single FP instruction and no fixed-point instructions are used for computation. However, the target platform has a 16-bit datapath and the FP operands must first be loaded in FP registers using fixed-point instructions. Each register is loaded using the standard *move* assembly instruction. A single FP instruction performs an entire FP operation, then the results are read from the registers, 16-bits at a time. To compare the throughput and area of FP modules integrated with a 32-bit datapath, separate versions of these modules are created. These *32-bit I/O* modules permit both the source operands and the destination to be specified in a single instruction.

A. Multiplication Module (Full HW Mult)

This full hardware module performs FP multiplies, using the *FPMult* instruction with a single cycle execution latency. This module requires 7 instructions, achieves a throughput of 200 MFLOPS, and requires 18322.7 μm^2, which increases the core area by 10.9%.

Full HW Mult (32-bit I/O) is a separate version of this module that can be integrated with a 32-bit datapath. This module uses the *FPMult32* instruction, which has a single cycle execution latency. If both FP operands must be received from a neighboring core, than three instructions are required to perform a multiplication. If instead all operands are readable from a core's data memory, than this module requires only a single instruction to perform multiplication. A throughput of 1200 MFLOPS is achievable and requires 17565.6 μm^2 silicon area. Although the area is less than the 16-bit datapath module, this value does not consider the additional area for a 32-bit datapath.

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

The full hardware add/sub module uses the *FPAdd* and *FPSub* instructions, each with a two-cycle execution latency. This module requires 7 instructions, achieves a throughput of 171 MFLOPS, and requires 14111.8 μm^2, which increases the core area by 8.4%.

Full HW Add/Sub (32-bit I/O) is a separate version of this module that can be integrated into a 32-bit datapath. This module uses the *FPAdd32* and *FPSub32* instructions for addition and subtraction, respectively. Each instruction has a single cycle execution latency. Three instructions are required to perform an addition or subtraction if both operands are read from a neighboring core, otherwise only a single instruction is necessary. This module has a throughput of 1200 MFLOPS and requires 10468.6 μm^2 silicon area. This area value value does not consider the additional area for a 32-bit datapath.

While adding full hardware FP support dramatically increases the throughput versus a full software implementation, the area overhead increases as well. An approach at reducing the hardware overhead while still increasing throughput versus a FP software library is presented next.

VI. Proposed Hybrid Floating-Point Modules (HFPMs)

The HFPMs are fixed-point software and custom FP instructions operating together on FP workloads to reduce the bottlenecks of software emulation while requiring less area than full hardware FP modules. HFPMs are alternatives along the software/hardware continuum. Different modules can be combined into a FPU to achieve a wide range of area and throughput targets. Similar to the full hardware FP modules, the HFPMs utilize a set of FP registers.

Custom FP instructions then perform operations on the data stored in these registers and output the results 16-bits at a time.

Table I indicates which instructions are utilized in each module. The HFPMs use these instructions to perform FP operations on data stored in the FP registers. Each instruction has a single cycle execution latency.

A. HFPM Mult Ver. 1

This module performs the mantissa multiplication in fixed-point software instructions. The normalization, rounding and exponent calculation are performed with the *FPMult_NormRnd* custom instruction. Fig. 2 shows the hardware for implementing this instruction into the execution stage of the AsAP2 pipeline [3]. The *FPMult_NormRnd* instruction is described in detail below.

1) FPMult_NormRnd: The mantissa multiplication is performed in software and then the product is loaded into FP Reg 1. The sign bits and exponents of both operands are loaded into FP Reg 2. This instruction calculates the new sign bit, exponent, and normalized and rounded product. The result is selectable via a 16-bit mux.

B. HFPM Add/Sub Ver. 1

This module is used for performing FP addition/subtraction operations. It uses fixed-point software instructions for determining the larger and smaller exponents. The rest of the calculation is carried out by the four custom FP instructions described below.

1) FPAdd_SatAlign: Following operand sorting in software, both operands are loaded into the FP registers. This instruction uses the software-calculated mantissa shift amount as an operand. This instruction inserts the hidden bit, saturates the alignment amount, then aligns and adds the mantissas. For effective subtraction, this instruction inverts and adds one to the smaller magnitude operand's mantissa. The unnormalized result is written into a FP register and the 16 MSBs are output.

2) Lzd: After the mantissas are added, this instruction counts leading zeroes to determine the shift amount for normalization. *Lzd* operates on 16-bit chunks of the sum to permit hardware reuse and reduce the area overhead. This instruction can also be reused for non-FP general purpose workloads.

TABLE I
FLOATING-POINT INSTRUCTION USE BY VARIOUS MODULES

Instruction	FP Module								
	Full SW Add/Sub	HFPM Add/Sub Ver. 1	HFPM Add/Sub Ver. 2	Full HW Add/Sub	Full HW Add/Sub (32-bit I/O)	Full SW Mult	HFPM Mult Ver. 1	Full HW Mult	Full HW Mult (32-bit I/O)
FPAdd_Compare			X						
FPAdd_Align			X						
FPAdd_SatAlign		X							
Lzd		X	X						
BShiftL		X	X						
FPAdd_Round		X	X						
FPMult_NormRnd							X		
FPAdd				X					
FPSub				X					
FPMult								X	
FPAdd32					X				
FPSub32					X				
FPMult32									X

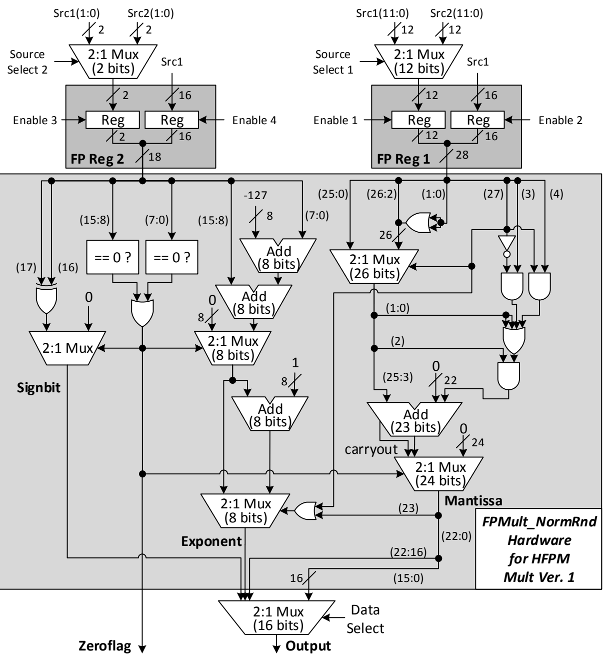


Fig. 2. Hardware to implement the *FPMult_NormRnd* instruction for *HFPM Mult Ver. 1 module*. FP Reg 1 is loaded with the product of the mantissa multiplication. FP Reg 2 is loaded with the sign bits and exponents of both operands. The final result of the FP multiplication is read out 16-bits at a time via the mux shown at the bottom.

3) BShiftL: Using the shift amount determined by *Lzd* and the sum already stored in the FP registers by *FPAdd_SatAlign*, this instruction shifts left for normalization, adjusts the exponent, and stores the 27 result LSBs in a FP register. An additional benefit of this instruction is that it can be reused for non-FP general purpose workloads.

4) FPAdd_Round: After normalization, this instruction rounds and adjusts the exponent and mantissa. The result is stored in a FP register and the 16 MSBs are returned.

C. HFPM Add/Sub Ver. 2

This module is an alternative to the *HFPM Add/Sub Ver. 1* module, and supplements it with two additional instructions, described below. The *FPAdd_SatAlign* instruction, however, is replaced by *FPAdd_Align*. Instead of handling operand sorting and the exponent difference calculation in fixed-point software, *FPAdd_Compare* and *FPAdd_Align* perform these operations.

1) FPAdd_Compare: Both operands are first loaded into the FP registers. *FPAdd_Compare* then compares the exponents and mantissas for FP addition and subtraction and overwrites the FP registers with the sorted operands. The saturated shift amount is output since exponent differences greater than 25 require identical mantissa alignments.

2) FPAdd_Align: This instruction is identical to *FPAdd_SatAlign*, except that it does not saturate the mantissa alignment shift amount. Using the sorted operands from the FP registers, the mantissas are aligned and added. The rest of the addition/subtraction operation is performed in the same fashion as the *HFPM Add/Sub Ver. 1* module using *Lzd*, *BShiftL*, and *FPAdd_Round*.

VII. COMPARISON OF FP MODULES

Table II displays the throughput and instruction count for each FP module. Fig. 3 plots the area and throughput of the FP modules, with cycles per FLOP on the vertical axis and additional core area on the horizontal axis. *HFPM Mult Ver. 1* requires 1.5% additional core area and provides 2.3× higher throughput than the full software multiplication module, *Full SW Mult*. Compared to the full software addition/subtraction module, *HFPM Add/Sub Ver. 1* and *HFPM Add/Sub Ver. 2* provide 1.8× and 3.6× higher throughput, and require 5.1% and 6.5% additional core area, respectively.

Although the full hardware FP modules provide dramatic throughput improvements; 8.7× higher throughput for addition/subtraction and 12.8× for multiplication, these modules are too large, especially if increasing core size sacrifices the total number of cores that fit on a die. The next section considers this overhead and the tradeoffs when the FP modules are combined into FPU implementations, consisting of one multiplication and one addition/subtraction module.

VIII. Comparison when Performing Unfused Multiply-Add Operation

Thirteen FPU implementations consisting of one addition/subtraction and one multiplication module are compared in Table III. Each implementation offers an alternative to a full software or full hardware FPU by trading off area and throughput. The unfused multiply-add operation is used to evaluate the performance of each FP module combination. Whereas a fused multiply-add performs the operation $a + b \times c$ in one step with a single rounding; the unfused multiply-add first calculates the product $b \times c$, rounds the result, adds the rounded product to a, then performs a second rounding. Although the fused multiply-add can reduce latency and offer higher accuracy, the additional area overhead is large for this platform.

Implementation 1, a full software FPU does not require any increase in core area, however the throughput is relatively low at 15.6 MFLOPS. Implementation 12, a full hardware FPU, requires 19.6% additional core area and provides 92.3 MFLOPS throughput. Table II demonstrates that the hybrid modules provide higher addition/subtraction throughput than the software module. However, in implementations 4 and 7; the instruction overhead to interface with the *Full SW Mult* module outweighs the benefits from using the *HFPM Add/Sub Ver. 1* or *Ver. 2* module. Although implementation 7 offers the lowest throughput of all the FPUs considered, the program size is reduced by 57.4% and throughput/area increases by 1.7× versus the full software implementation.

Fig. 4 plots the cycles per FLOP for the unfused multiply-add operation versus core area. Fig. 4(a) shows the results for all FP implementations, and (b) presents those with the highest throughput per area. Nine of the FPU designs provide higher throughput per area than the full software FPU, the smallest of which requires 92% less area than a full hardware FPU implementation. Implementation 8, consists of only hybrid modules and increases throughput per area by 4.1× versus the software implementation and is 9.95% smaller than the full hardware FPU.

TABLE II
Throughput and Instruction Count for Each Module.

	Addition/Subtraction Module					Multiplication Module			
	Full SW Add/Sub	HFPM Add/Sub Ver. 1	HFPM Add/Sub Ver. 2	Full HW Add/Sub	Full HW Add/Sub (32-bit I/O)	Full SW Mult	HFPM Mult Ver. 1	Full HW Mult	Full HW Mult (32-bit I/O)
Throughput (MFLOPS)	19.7	35.8	70.6	171	1200	15.6	35.3	200	1200
Average Speedup	1x	1.8x	3.6x	8.7x	60.9x	1x	2.3x	12.8x	76.9x
# of Instr.	216	41	18	7	3	80	35	7	3

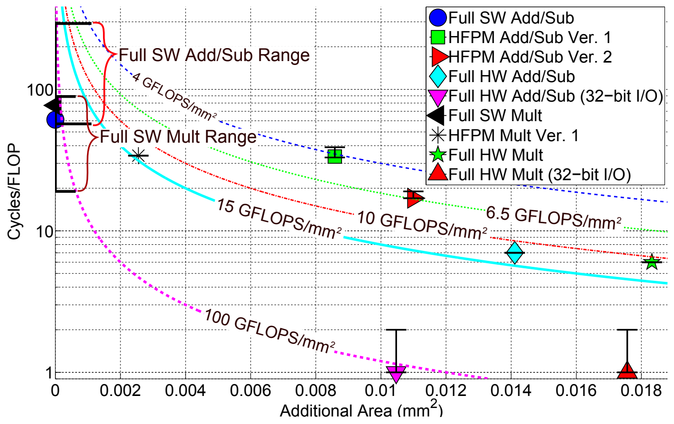


Fig. 3. Additional area versus cycles per FLOP for all FP modules. The nine symbols in the legend denote the average clock cycles per FLOP, and the endpoints of the interval bars for each symbol denote the corresponding minimum and maximum. Contour lines show throughput per additional area tradeoffs. Results are obtained from synthesis in 65 nm CMOS.

IX. Conclusion

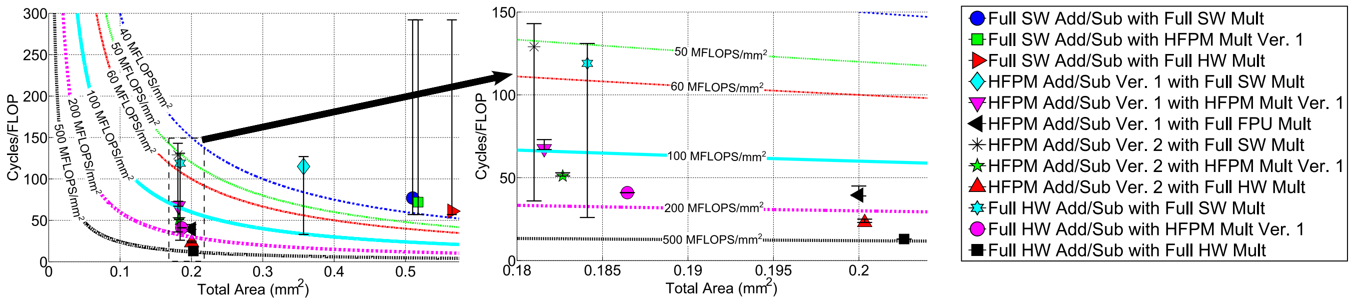
In this paper, three HFPMs are presented for a fine-grained processor. These modules increase throughput versus a software implementation by adding custom FP instructions. Area overhead is kept low by reusing the existing fixed-point functional units such as the multiplier and adder. Thirteen functionally equivalent FPU implementations using combinations of fixed-point software, FP hardware, and hybrid modules are synthesized in 65 nm CMOS. Nine of these implementations increase throughput per area by 1.05-8.5× when compared to a software implementation, and use 1.08-12.5× less area than a full hardware alternative.

Acknowledgment

The authors gratefully acknowledge support from C2S2 Grant 2047.002.014, NSF Grant 1018972 and 0903549 and CAREER Award 0546907, SRC GRC Grant 1598, 1971, and 2321 and CSR Grant 1659, Intel, UC Micro, SEM, Fudan University, Frances Muenzer, Nima Mostafavi, Aaron Stillmaker, Bin Liu, and Brent Bohnenstiehl. 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. Birkhäuser Basel, 2009.
[2] S. Gilani, N. S. Kim, and M. Schulte, "Energy-efficient floating-point arithmetic for software-defined radio architectures," in *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE Intl. Conf. on*, 2011, pp. 122–129.
[3] D. Truong *et al.*, "A 167-processor computational platform in 65 nm CMOS," *Solid-State Circuits, IEEE Journal of*, vol. 44, no. 4, pp. 1130–1144, 2009.
[4] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *Computers, IEEE Trans. on*, vol. 60, no. 7, pp. 913–922, 2011.
[5] W. T. Padgett and D. V. Anderson, "Fixed-point signal processing," *Synthesis Lectures on Signal Processing*, vol. 4, no. 1, pp. 50–51, 2009.
[6] S. Gilani, N. S. Kim, and M. Schulte, "Virtual floating-point units for low-power embedded processors," in *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd Intl. Conf. on*, 2012, pp. 61–68.



(a) (b)

Fig. 4. Total area versus cycles per FLOP for performing unfused multiply-add $(A + B{\times}C)$ using different FP implementations. The twelve symbols in the legend denote the average clock cycles per FLOP, and the endpoints of the interval bars denote the minimum and maximum . Contour lines indicate throughput per area values. (a) All implementations shown, (b) Implementations with largest average throughput per area. Implementations with a HFPM or full hardware module provide the largest throughput per area.

TABLE III
COMPARISON OF FPU IMPLEMENTATIONS USING DIFFERENT COMBINATIONS OF FP MODULES

FPU Implementation	1	2	3	4	5	6	7	8	9	10	11	12	13
Addition/ Subtraction Module	Full SW Add/ Sub	Full SW Add/ Sub	Full SW Add/ Sub	HFPM Add/ Sub Ver. 1	HFPM Add/ Sub Ver. 1	HFPM Add/ Sub Ver. 1	HFPM Add/ Sub Ver. 2	HFPM Add/ Sub Ver. 2	HFPM Add/ Sub Ver. 2	Full HW Add/ Sub	Full HW Add/ Sub	Full HW Add/ Sub	Full HW Add/ Sub (32-bit I/O)[a]
Multiplication Module	Full SW Mult	HFPM Mult Ver. 1	Full HW Mult	Full SW Mult	HFPM Mult Ver. 1	Full HW Mult	Full SW Mult	HFPM Mult Ver. 1	Full HW Mult	Full SW Mult	HFPM Mult Ver. 1	Full HW Mult	Full HW Mult (32-bit I/O)[a]
FPU Area (mm^2)	-	0.0026	0.0183	0.0086	0.0116	0.0300	0.0110	0.0127	0.0303	0.0141	0.0165	0.0326	0.0294
Core Area (mm^2)	0.168	0.171	0.186	0.177	0.180	0.198	0.179	0.181	0.198	0.182	0.185	0.201	-
Area Increase (%)	-	1.79	10.7	5.36	7.14	17.9	6.55	7.74	17.9	8.33	10.1	19.6	-
Max Number of Cores	164	161	148	155	153	139	153	152	139	151	148	137	-
Fewer Cores (%)	-	1.83	9.76	5.49	6.71	15.2	6.71	7.32	15.2	7.93	9.76	16.5	-
Unfused Multiply-Add $(A + B{\times}C)$ Operation[c]													
Number of Cores Required	3	3	2	2	1	1	1	1	1	1	1	1	-
Number of Instructions	296	279	251	149	76	48	126	53	25	115	42	14	6
Throughput[b] (MFLOPS)	15.6	16.7	19.7	10.4	17.8	30.4	9.30	23.5	52.2	10.1	29.3	92.3	600.
Throughput/Area[b] (MFLOPS/mm^2)	31.0	32.6	53.0	29.4	98.9	154.	52.0	130.	264.	55.5	158.	459.	-

Results are based on synthesis in 65 nm CMOS with a supply voltage of 1.3 V at 1.2 GHz.
[a] 32-bit I/O implementation requires a 32-bit datapath (platform uses 16-bit datapath), therefore integration and area data is not available.
[b] Throughput calculations are performed using average cycles per FLOP.
[c] Unfused multiply-add involves the multiplication $(b \times c)$, a rounding step, followed by the addition operation $a + (b \times c)$, and then a second rounding step. A fused multiply-add operation requires that $a + b \times c$ is performed in a single operation with a single rounding.

[7] F. Fang, T. Chen, and R. A. Rutenbar, "Lightweight floating-point arithmetic: case study of inverse discrete cosine transform," *EURASIP Journal on Applied Signal Processing*, vol. 2002, no. 1, pp. 879–892, Jan. 2002.

[8] S.-W. Lee and I.-C. Park, "Low cost floating-point unit design for audio applications," in *Circuits and Systems, 2002. ISCAS 2002. IEEE Intl. Symp. on*, vol. 1, 2002, pp. I–869–I–872 vol.1.

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

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

[11] N. Hockert and K. Compton, "Improving floating-point performance in less area: Fractured Floating Point Units (FFPUs)," *Journal of Signal Processing Systems*, vol. 67, no. 1, pp. 31–46, Apr. 2012.

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

[13] M. Aharoni *et al.*, "FPgen - a test generation framework for datapath floating-point verification," in *High-Level Design Validation and Test Workshop, 2003. Eighth IEEE Intl.*, 2003, pp. 17–22.

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

[15] Z. Xiao and B. Baas, "A 1080p H.264/AVC baseline residual encoder for a fine-grained many-core system," *Circuits and Systems for Video Technology, IEEE Trans. on*, vol. 21, no. 7, pp. 890–902, July 2011.