Configurable Multimode Embedded Floating-Point Units for FPGAs

Yee Jern Chong, *Student Member, IEEE*, and Sri Parameswaran, *Member, IEEE*

***Abstract*—Performance of field-programmable gate arrays (FPGAs) used for floating-point applications is poor due to the complexity of floating-point arithmetic. Implementing floating-point units (FPUs) on FPGAs consume a large amount of resources. This makes FPGAs less attractive for use in floating-point intensive applications. Therefore, there is a need for embedded FPUs in FPGAs. However, if unutilized, embedded FPUs waste space on the FPGA die. To overcome this issue, we propose a flexible multimode embedded FPU for FPGAs that can be configured to perform a wide range of operations. The floating-point adder and multiplier in our embedded FPU can each be configured to perform one double-precision operation or two single-precision operations in parallel. To increase flexibility further, access to the large integer multiplier, adder and shifters in the FPU is provided. Benchmark circuits were implemented on both a standard Xilinx Virtex-II FPGA and on our FPGA with embedded FPU blocks. The results using our embedded FPUs showed a mean area improvement of 5.5 times and a mean delay improvement of 5.8 times for the double-precision benchmarks, and a mean area improvement of 3.8 times and a mean delay improvement of 4.2 times for the single-precision benchmarks. The embedded FPUs were also shown to provide significant area and delay benefits for fixed-point and integer circuits.**

***Index Terms*—Dual-precision, embedded block, field-programmable gate array (FPGA), floating-point, floating-point unit (FPU), FPGA architecture.**

I. Introduction

FIELD-PROGRAMMABLE gate arrays (FPGAs) allow designers to build virtually any logic device in hardware quickly and easily. The programmability and flexibility of FPGAs make them ideal for prototyping, quick time-to-market applications, one-off implementations, and customized hardware. They are especially valuable in applications where a custom circuit is required, but the production volume does not justify the costs and time of fabricating them on application-specific integrated circuits (ASICs).

While the flexibility of FPGAs is highly valued, the performance of FPGAs significantly lags that of ASICs. Advances in process technology and FPGA architectures have allowed FPGAs to close the gap to a certain extent. For example, an increasing amount of hard logic has been embedded into FPGAs in an effort to improve performance. These include embedded block RAMs, fast carry-chains, embedded multipliers, digital signal processing (DSP) cores and even whole microprocessors (e.g., PowerPC cores in Xilinx Virtex-II Pro). One area where FPGA performance is still lacking is in floating-point applications.

FPGAs are widely used for scientific computation because of the ease of customizing the hardware for the application. Floating-point is also important for scientific computations for their numerical stability. Despite the poor floating-point performance of FPGAs, there has been significant interest in using FPGAs for scientific applications [1]–[5]. While FPGAs excel at integer and fixed-point applications, floating-point applications occupy a large and often impractical amount of resources when implemented on FPGAs. The limited size and architecture of FPGAs are not well-suited for floating-point applications. On the other hand, ASICs can be very efficient at floating-point operations, but lack the programmability and flexibility that is desired in many situations, and the cost of an ASIC can be prohibitively high. General purpose processors with floating-point units (FPUs) are programmable and can be capable of high clock rates, but performance is limited by the lack of customizable hardware [6]. Thus, FPGAs are a very attractive platform for floating-point applications if their limitations can be overcome.

Currently available commercial FPGAs still have not addressed the problems associated with implementing floating-point applications on FPGAs. There has been research work investigating the use of embedded floating-point units (FPUs) in FPGAs, such as [7]–[9], but despite demonstrating the significant benefits of embedded FPUs, commercial vendors have yet to include them. One argument for this non-inclusion is that the demand for embedded FPUs is still not high enough to warrant the addition of embedded FPUs. Another is that if the embedded FPUs are not utilized, the area is wasted. Therefore, to make it a more attractive proposition for vendors to include embedded FPUs, this paper presents a more flexible FPU that may be utilized in multiple different ways. The multimode embedded FPU presented in this paper may be configured to perform one double-precision operation, two single-precision operations in parallel, or a variety of integer operations. Such an FPU increases the number of ways in which the designer may utilize these embedded blocks, and therefore reduces the likelihood that the area is wasted. A single FPGA can contain a number of these multimode embedded FPUs, where each can be configured to perform a different task or they can be used to build massively parallel circuits.

Existing commercial FPGAs already contain many dedicated hardware blocks to accelerate arithmetic operations. Embedded multipliers are common in many modern FPGAs, like the Xilinx

Manuscript received February 22, 2010; revised May 27, 2010 and August 15, 2010; accepted August 16, 2010. Date of publication October 28, 2010; date of current version September 14, 2011.

The authors are with the School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia (e-mail: yeejernc@cse.unsw.edu.au; sridevan@cse.unsw.edu.au).

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.2010.2072996

Virtex-II. Multiply-accumulate (MAC) blocks are available in Altera's Stratix and Stratix II FPGAs. To make it more attractive for FPGA vendors to include embedded FPUs, the FPU should be multifunctional, rather than a static single-function block. Rose [10] explores the question of what hard, dedicated circuits should be included on FPGAs. Rose suggests that when determining what structures to include on an FPGA, the more useful the hard structure is across a wider range of applications, the better, even if they are less than perfectly efficient. This is the philosophy we are employing in our design. Our goal is to create a flexible, generic embedded FPU, which over a variety of applications will improve performance and save a significant amount of FPGA real estate when compared to implementations on current FPGAs.

With this goal of flexibility in mind, our embedded FPU was designed so that it can be configured to perform several useful functions. Since multiplication and addition are two of the most commonly used arithmetic operations, both in integer and floating-point domains, each embedded FPU contains a floating-point multiplier and adder. Both the floating-point multiplier and adder can perform a double-precision operation, or two single-precision operations in parallel, thus potentially doubling the throughput in single-precision mode. To make the embedded FPU useful for integer based applications, it can be configured so that several of the FPU's internal integer components are made available. From the floating-point multiplier, the fast 53×53-bit multiplier (can be configured as two 24×24-bit multipliers) is accessible. From the floating-point adder, the 53-bit carry-lookahead adder (can be configured as two smaller 27- and 26-bit adders) and two large shifters are accessible. Building large shifters in the fine-grained fabric of the FPGA is expensive in terms of resources, therefore it would be beneficial to allow access to the large shifters present in the floating-point adder. The overheads associated with adding this functionality to the embedded FPU is not excessive for the amount of flexibility added. Another common arithmetic operation is a fused multiply-add operation $(a \times b + c)$. To avoid routing delays, we have also designed the embedded FPU to allow the output of the multiplier to be linked to an input of the adder, allowing it to perform any one of the following operations: one double-precision multiply-add, two single-precision multiply-adds in parallel, one 53-bit integer multiply-add, or two 24-bit integer multiply-adds in parallel.

There are many different possible configurations that could be made, but we believe that the configurability of our embedded FPU is sufficient for many common applications. It is beyond the scope of this paper to explore the multitude of combinations of different configurations possible and is left for possible future work.

The rest of this paper is organized as follows. Section II provides a review of related work. Section III provides some background on floating-point numbers and conventional floating-point adder and multipliers. Section IV describes the architecture of our dual-precision FPU and the modifications made to the floating-point adder and multiplier. Section V describes the architecture of the FPGA with embedded FPU blocks. Section VI describes the modelling methodology chosen to model our FPGA with embedded FPU blocks. Section VII details the experiments performed and Section VIII discusses the results obtained. The conclusions are presented in Section IX followed by a brief discussion on future work in Section X.

II. Related Work

A number of recent works have investigated the use of embedded FPUs in FPGAs [7]–[9]. Beauchamp *et al.* [7] proposed an island-style FPGA architecture with embedded FPUs. Ho *et al.* [8] also modelled the use of embedded floating-point units in FPGAs. Both works demonstrate that significant area and delay improvements are gained by using an FPGA with embedded FPUs over using a standard FPGA for implementing floating-point applications.

Another work by Ho *et al.* presents a Hybrid FPGA architecture [9], where coarse-grained units are embedded into the FPGA. Each coarse-grained unit contains word blocks in addition to floating-point multipliers and adders. The word blocks can be configured to perform some simple operations, such as integer additions and comparisons. The coarse-grained units employ bus-based routing internally to improve speed and density. The advantage of Ho's architecture is that it moves more of the circuit that would have been implemented in the fine-grained fabric into the coarse-grained unit, which would reduce routing delays.

Dual-precision floating-point adders and multipliers have been presented in [11]–[13]. Akkaş presents a floating-point adder [11] that can each be configured to perform either one quad-precision addition, or two double-precision additions in parallel. Even *et al.* [13] presents a floating-point multiplier, that can be configured to perform either one double-precision multiplication or one single-precision multiplication. Akkaş improves on Even's design in [12] with a floating-point multiplier that can perform either one quad-precision multiplication, or two double-precision additions in parallel. Akkaş' designs can be scaled down to one double-precision/two single-precision units.

Diniz and Govindu [14] presented the design of a field programmable dual-precision FPU that can be configured at run-time to switch between single-precision (two in parallel) and double-precision multiplication and addition. The FPU is a soft-core that is implemented in the fine-grained fabric, so does not provide the improved speed and area of an embedded hardware FPU. Their design targets run-time reconfigurable systems, so provides no benefits to applications where changing precision at run-time is not needed since there is a speed and area overhead.

In contrast to the works by Beauchamp [7] and Ho [8], who only estimate area and delay of the embedded FPUs (from FPUs in existing commercial processors) when modelling their FPGAs with embedded FPUs, we have built actual FPUs specifically for this work. This provides a more accurate model for our embedded FPUs.

The Hybrid FPGA work by Ho [9] synthesizes an FPU library for the FPUs embedded in their coarse-grained unit. The FPUs in the coarse-grained units were double-precision only. The configuration of the coarse-grained units in the Hybrid FPGA was customized towards the benchmark circuits. Our

embedded FPU is more generic and flexible, but lacks the customizable word-blocks of the Hybrid FPGA. The Hybrid FPGA architecture is an interesting design and some of the ideas could be adapted for our multimode embedded FPU in future work.

While Beauchamp [7] briefly suggests modifying the double-precision FPUs into dual single-precision FPUs and allowing access to internal components of the FPU to reduce the disadvantage of unutilized embedded FPUs wasting space, our work is the first to objectively implement and evaluate this idea.

Akkaş [11], [12] and Even [13] have presented designs for dual-precision FPUs, but our work is the first to investigate employing dual-precision FPUs as embedded blocks in FPGAs. Our dual-precision FP adder is similar in design to Akkaş' [11], but we use a different approach for our dual-precision FP multiplier. Akkaş uses two multipliers for the two lower precision multiplications, and uses the two multipliers in a multicycle feedback arrangement for the higher precision multiplication [12]. This reduces area, but because it is a multicycle operation, it increases latency by an extra clock cycle and blocks the use of the multiplier during that cycle when performing the higher precision multiplication, thus resulting in lower throughput. Our novel design uses a multiplier tree large enough for performing a double-precision multiplication in one cycle. When in single-precision mode, the partial products for each pair of operands are injected into the multiplier tree at different locations. This is explained in more detail in Section IV-B. Our design has the advantage of completing both single- and double-precision multiplications in one cycle. The disadvantage of our design is that the delay to complete the single-precision multiplications is slightly longer than necessary and area is greater than Akkaş' design. These disadvantages are not much of a concern for the embedded FPU because it will be many times faster and smaller than implementing a floating-point multiplier in the fine-grained logic.

Unlike Diniz and Govindu [14]'s work which targets dynamically reconfigurable applications only, our embedded FPU provides benefits in both reconfigurable and non-reconfigurable applications. Reconfiguration time of our embedded block is low because it only involves changing the control signals for the multiplexers.

Therefore, the main contributions of this paper are as follows:

- an FPGA architecture with a novel flexible multimode embedded FPUs is presented and modelled;
- an architecture for an IEEE754 compliant flexible multimode FPU that can perform double-precision, or dual single-precision operations, as well as a variety of integer operations is presented;
- a novel dual-mode integer multiplier that can compute one large multiplication or two smaller multiplications within in a single cycle is presented, which is the key component in the dual-precision floating-point multiplier.

III. Background

A. Floating-Point Representation

The IEEE754 standard [15] floating-point format consists of three fields—a sign bit (s), a biased exponent (e), and a mantissa (f). Single-precision numbers have a 1-bit sign, 8-bit exponent, and 23-bit mantissa as shown in Fig. 1(a). Double-precision numbers have a 1-bit sign, 11-bit exponent, and 52-bit mantissa as shown in Fig. 1(b).

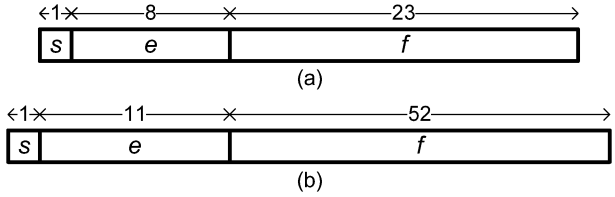

Fig. 1. (a) Single-precision and (b) double-precision IEEE754 floating-point numbers.

The three fields make up a floating-point number according to (1) (single-precision) and (2) (double-precision). There is an implied "1" to the left of the binary point (except in the special case of denormal numbers)

$$X = (-1)^s \times 1.f \times 2^{(e-127)} \tag{1}$$

$$X = (-1)^s \times 1.f \times 2^{(e-1023)}. \tag{2}$$

Floating-point numbers have an advantage of being able to cover a much larger dynamic range compared to fixed-point numbers. The disadvantage is that floating-point computations are much more complex to implement in hardware.

B. Floating-Point Addition

The conventional floating-point addition algorithm consists of five stages—exponent difference, pre-alignment, addition, normalization and rounding [16]. Given floating-point numbers $X_1 = (s_1, e_1, f_1)$ and $X_2 = (s_2, e_2, f_2)$, the stages for computing $X_1 + X_2$ are described as follows.

1) Find exponent difference $d = e_1 - e_2$. If $e_1 < e_2$, swap position of mantissas. Set larger exponent as tentative exponent of result.
2) Prealign mantissas by shifting smaller mantissa right by d bits.
3) Add or subtract mantissas to get tentative result for mantissa.
4) Normalization. If there are leading-zeros in the tentative result, shift result left and decrement exponent by the number of leading zeros. If tentative result overflows, shift right and increment exponent by 1 bit.
5) Round mantissa result. If it overflows due to rounding, shift right and increment exponent by 1 bit.

Fig. 2 shows the datapath for a floating-point addition. Only the main parts of the datapath are shown for clarity. The prealignment and normalization stages require large shifters. The prealignment stage requires a right shifter that is twice the number of mantissa bits (i.e., 48 bits for single-precision, 106 bits for double-precision) because the bits shifted out have to be maintained to generate the guard, round and sticky bits needed for rounding. The shifter only needs to shift right by up to 24 places for single-precision or 53 places for double-precision.

The normalization stage requires a left shifter equal to the number of mantissa bits plus 1 (to shift in the guard bit), i.e., 25-bits for single-precision and 54-bits for double-precision.

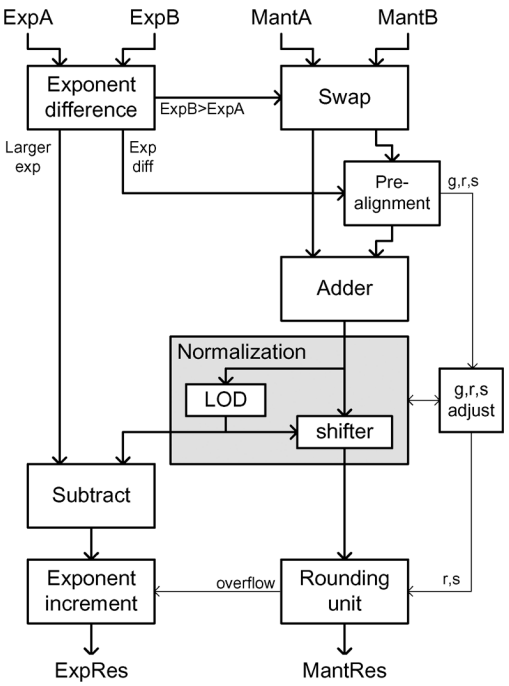

Fig. 2. Floating-point adder datapath.

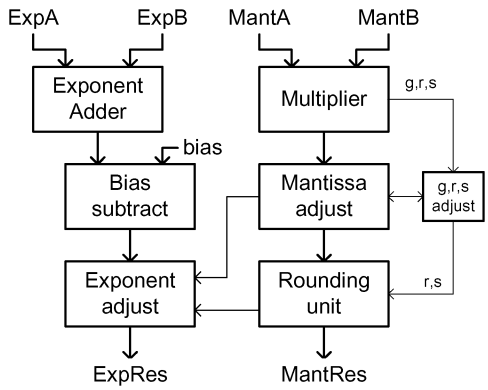

Fig. 3. Floating-point multiplier datapath.

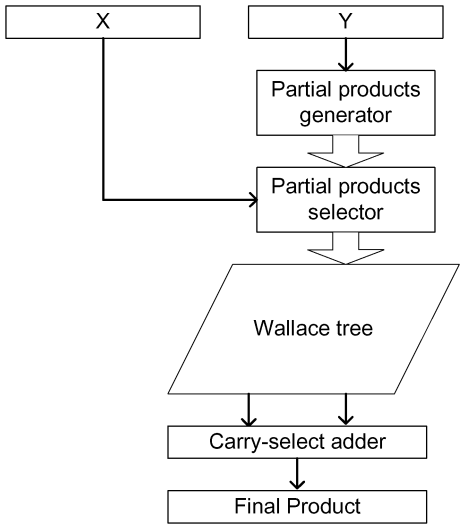

Fig. 4. Booth Wallace multiplier structure. X is the multiplier and Y is the multiplicand.

The shift amount is determined by the leading one detector (LOD) circuit [17], which outputs the number of leading zeros before the first one in the bit string.

The final stage of the floating-point adder is the rounding unit. It makes a rounding decision based on the rounding mode, the LSB of the mantissa, the round bit and the sticky bit. If rounding is necessary, "1" is added at the LSB of the mantissa.

There are other variations of the conventional floating-point adder architecture that improve performance, such as the leading one predictor (LOP) architecture [18] and the dual-path architecture [19]. The tradeoff involved with these two architectures is that they require additional hardware and area for the added performance. The conventional architecture was chosen over the faster architectures for area savings and reduced complexity, which simplifies the conversion to a dual-precision structure.

C. Floating-Point Multiplication

Algorithmically, floating-point multiplication is much simpler than floating-point addition. However, a very wide integer multiplier is required. Given floating-point numbers $X_1 = (s_1, e_1, f_1)$ and $X_2 = (s_2, e_2, f_2)$, $X_p = X_1 \times X_2$ can be computed using

$$s_p = s_1 \oplus s_2 \tag{3}$$

$$e_p = e_1 + e_2 - bias \tag{4}$$

$$1.f_p = 1.f_1 \times 1.f_2. \tag{5}$$

Fig. 3 shows the datapath for a floating-point multiplier. Only the main parts of the datapath are shown for clarity. If the result from the multiplier has two bits left of the binary point, the mantissa has to be shifted right to compensate and the exponent is incremented. If the rounding of the mantissa results in an overflow, the mantissa is shifted right by one and the exponent is incremented.

Equation (5) calls for a very wide multiplier—53 $\times$ 53-bit unsigned multiplier for double-precision and 24 $\times$ 24-bit for single-precision. Therefore, an efficient multiplier must be employed. In our work, we use a Radix-4 modified booth encoded (MBE) Wallace multiplier as shown in Fig. 4, which was based on the designs in [20]–[22]. Radix-4 recoding halves the number of partial products, thus reducing the number of levels required in the Wallace tree, which improves performance and reduces area. For more on Booth recoding, refer to [23] and [24]. The Wallace tree [25] reduces the number of partial products to two, which are added together by a fast final adder to get the final product. For the 53 $\times$ 53-bit multiplication, 7 reduction levels in the Wallace tree are needed and 5 reduction levels for the 24$\times$ 24-bit multiplication.

IV. Multimode FPU

The multimode embedded FPU was designed to include a dual-precision floating-point multiplier and a dual-precision

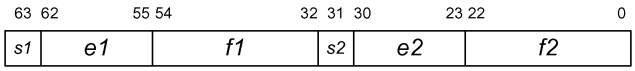

Fig. 5. Two single-precision numbers in one 64-bit word.

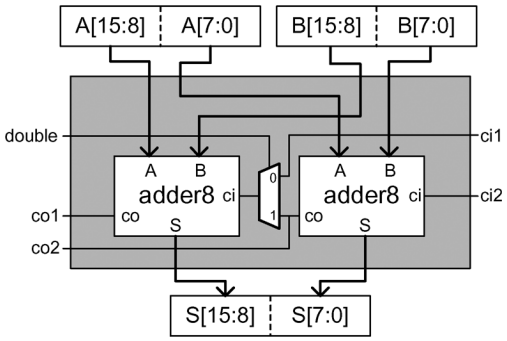

Fig. 6. 2 × 8-bit/1 × 16-bit adder—capable of performing two independent 8-bit additions in parallel or one 16-bit addition.

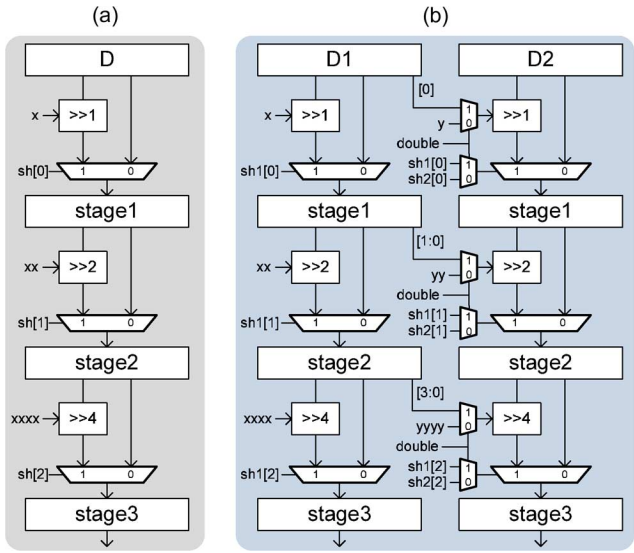

Fig. 7. (a) Conventional multistage shifter and (b) modified multistage shifter capable of operating as two smaller shifters or as a larger combined shifter.

floating-point adder. They were designed to be IEEE754 compliant [15], except that hardware support for denormals was not included and only the IEEE754 default rounding mode (round-to-nearest even) was implemented. The design can be easily modified to support the other rounding modes specified in IEEE754. The dual-precision FPU accepts 64-bit inputs, where double-precision operands occupy the full 64-bits and single-precision operands each occupy half of the 64-bits as shown in Fig. 5.

The following two sub-sections explain the modifications made to a standard floating-point adder and multiplier to convert them into dual-precision versions capable of performing one double-precision operation or two single-precision operations in parallel.

A. Dual-Precision Floating-Point Adder

As in [11], the method for converting a standard floating-point adder into a dual-precision adder involves duplicating the datapath for a single-precision adder and then linking duplicated functional blocks together (and widen them where necessary) to accommodate double-precision. Multiplexers controlled by a mode signal (double) selects between single-precision mode and double-precision mode.

A double-precision exponent is 11-bits, while a single-precision exponent is 8-bits. For all the operations on the exponents that involve adding or subtracting, we use two 8-bit adders that can combine into one 16-bit adder, as shown in Fig. 6. When double $= 0$, it computes $S[15:8] = A[15:8] + B[15:8]$ and $S[7:0] = A[7:0] + B[7:0]$ in parallel. When double $= 1$, it computes $S[15:0] = A[15:0] + B[15:0]$. This 2 × 8-bit/1 × 16-bit adder/subtractor is used in the exponent difference stage, the exponent subtractor (after normalization) and the exponent incrementer. The 11-bit exponent can be aligned to the MSB position (i.e., A[15:5]) so that the carry-out is valid. If the carry-in is used, the remaining bits can be padded appropriately to allow the carry-in to propagate or it can be multiplexed directly into the carry-in of A[5]. Alternatively, the 11-bit exponent may also be aligned to the LSB position (i.e., A[10:0]) and the carry-out read out of the 12th bit position (i.e., A[11]).

A single-precision mantissa is 24-bits and linking two 24-bit functional units in the same way as Fig. 6 results in 48-bits, which is not wide enough for a 53-bit double-precision mantissa. Therefore, most of the functional units on the mantissa side of the datapath must be widened. The main adder has the same structure as Fig. 6, but the left adder is widened to 27-bits and the right adder is widened to 26-bits, to give a total of 53-bits when linked. The other adders on the mantissa datapath, like in the rounding unit, are also modified in a similar fashion.

The variable right shifter in the prealignment stage is modified so that it behaves as two independent 53-bit right shifters when in single-mode and a combined 106-bit shifter in double-mode. Both shifters are able to shift the inputs by 64 places (6-bit shift amount). Fig. 7 compares the structure of a standard multistage shifter with the modified shifter. In Fig. 7(a), D is the input to be shifted and sh is the shift amount. In Fig. 7(b), double $= 0$ sets it to single-mode, where $D1$ is shifted by $sh1$ places and $D2$ is shifted by $sh2$. The inputs on the sides of the shift boxes indicate what values are to be shifted in. For prealignment, x and y are the sign extensions of $D1$ and $D2$, respectively. When double $= 1$, the bits shifted into the $D2$ shifter comes from the LSBs of the $D1$ shifter to form a larger combined shifter for double-mode. $D1 : D2$ is shifted by $sh1$ places in double-mode.

The variable left shifter in the normalization stage would be the left shift version of Fig. 7(b), with two 27-bit shifters in single-mode and a combined 54-bit shifter in double-mode.

B. Dual-Precision Floating-Point Multiplier

The main challenge in modifying a standard floating-point multiplier into a dual-precision multiplier is modifying its 53 × 53-bit integer multiplier. The design by Akkaş [12] uses two smaller multipliers, which are reconfigured into a multicycle (two cycles) arrangement in high precision mode. The problem with Akkaş' design is that a new high precision multiplication can only start every other cycle as the hardware is unavailable until the previous multiplication is completed. To avoid this problem, our novel design uses the full 53 × 53-bit multiplier for

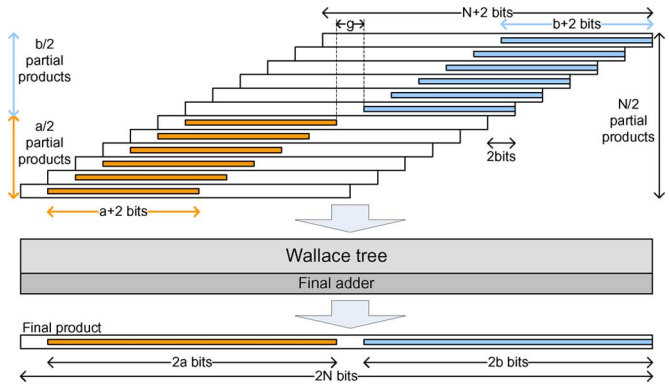

Fig. 8. Modified Booth Wallace multiplier that can perform either one N-bit unsigned multiplication or two smaller a and b bit multiplications in parallel.

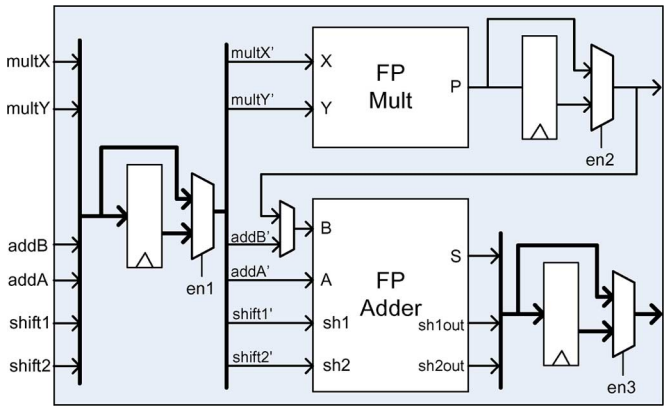

Fig. 9. Structure of embedded FPU block. Contains a dual-precision floating-point multiplier and adder.

both double-mode and single-mode. Our design requires more hardware compared to Akkaş', but is partly offset by the use of Radix-4 recoding. A higher radix could be employed to reduce hardware further if necessary.

The dual-mode integer multiplier has to perform one 53-bit multiplication $P = X \times Y$ in double-mode or two 24-bit multiplications $p_1 = x_1 \times y_1$ and $p_2 = y_1 \times y_2$ in single-mode. The Radix-4 MBE Wallace multiplier from Fig. 4 was retained for its performance and area efficiency. The partial products generator was modified to generate partial products for the full 53-bit Y when in double-mode and for both of the 24-bit portions y_1 and y_2 in single-mode. The partial products selector operates as normal in double-mode, but in single-mode, the partial products for each of the 24-bit portions need to be placed in a special arrangement before being fed into the Wallace tree as shown in Fig. 8.

Illustrated in Fig. 8 is how the partial products for two smaller a and b bit multiplications are arranged within the partial products array for the larger N-bit significand (for this application, $N = 53$ and $a = b = 24$). Each partial product is padded with two extra bits on the left to avoid needing to sign-extend each partial product as described in [20]. The white rectangles represent the partial products for the N-bit significand. Each partial product is offset 2-bits to the left of the partial product above it as a result of the Radix-4 recoding. The thinner rectangles within the white rectangles represent the positions of the partial products for the two a and b bit multiplications. They are placed at opposite "corners" of the array and all unused bits are padded to zero. There must be a separation of at least g unused bits between the top-most a partial product and the bottom-most b partial product to ensure that the intermediate values of the b computation will not contaminate the values of the a computation as they pass through the Wallace tree. The maximum bit-length of the partial products increases by 1-bit every odd numbered level of the Wallace tree (excluding the first level). Our Wallace tree has seven levels for double-precision when using Radix-4 recoding. Therefore, a separation of at least $g = 3$ is sufficient to guarantee correctness. A double-precision partial product is 55-bits long and a single-precision partial product is 26-bits. Therefore, the condition $g \geq 3$ is satisfied. The Wallace tree reduces the partial products down to two, which are added together by the final carry-select adder. The output is a $2N$-bit result, from which the smaller $2a$-bit and $2b$-bit results can be extracted.

The modifications for the rest of the dual-precision floating-point multiplier datapath are relatively straightforward. The adders and subtractors on the exponent part of the datapath were modified into 2 × 8-bit/1 × 16-bit adders and subtractors like in Fig. 6. The rounding unit is modified in the same way as the rounding unit in the floating-point adder in Section IV-A.

C. Embedded Multimode FPU Block

Fig. 9 shows the structure of the embedded FPU block. The FPU block contains one floating-point multiplier and one floating-point adder. These can be used independently, or configured in a multiply-add configuration by enabling a bus connecting the output of the multiplier to an input of the adder. Optional registers are available at the inputs and outputs of the FPU block to allow for easy implementation of pipelined or multicycle circuits.

To increase the usefulness of the floating-point units, several key integer components within the floating-point units were made accessible. These components are as follows.

- Dual-mode 53 × 53-bit integer multiplier in the floating-point multiplier. Can be configured as two independent 24 × 24-bit multipliers.
- Dual-mode 53-bit integer adder in the floating-point adder. Can be configured as independent 27-bit and 26-bit adders.
- The 106-bit right shifter in the pre-alignment stage of the floating-point adder. Since maximum shift is 64 places, access is given to 64-bits of the shifter so that it appears to be a 64-bit shifter with maximum shift of 64 places.
- The 54-bit left shifter in the normalization stage of the floating-point adder. Maximum shift is 54 places.

To make these integer components available, multiplexers are added at the inputs of each component. The multiplexers select their normal inputs when in floating-point mode and select external inputs in integer mode. Multiplexers at the outputs of the FPUs select between the floating-point result and the integer result. The integer and floating-point modes share the same inputs and outputs to minimize the number of pins needed (but the shifters are given dedicated input/outputs). Sharing the input

TABLE I
AREA OF EMBEDDED FPU BLOCK IN TERMS OF NUMBER OF VIRTEX-II SLICES

	Area(A)(μm^2)	Feature size (L)(μm)	Normalized area(A/L^2)	Area in slices	Input pins	Output pins
Virtex-II slice	10,912	0.15	485,013	1	8(8)	2(2)
Embedded FPU	302,524	0.13	17,900,856	37	396(295)	288(74)

TABLE II
AREA AND DELAY OVERHEAD OF MULTIMODE FPU

	Conventional FPU	**Multi-mode FPU**	Overhead
Area (μm^2)	200828	263065	31%
Delay (ns)	5.1	6.17	21%

and output pins allows the embedded block to be configured to perform integer multiply-add operations using the same bus as the floating-point multiply-add.

The decision to include the large integer multiplier and adders was made because they are very commonly used operations. Large multipliers are slow and very costly in terms of resources when implemented in the fine-grained FPGA fabric, which has led FPGA vendors to include embedded multipliers in many of their FPGAs. By embedding our multimode FPU block, the FPGA vendors could remove many of the embedded multipliers because we provide access to the fast integer multiplier that is used within the FPU block. The multiply-add functionality of our embedded FPU block could also allow FPGA vendors to replace MAC units in their FPGAs with our embedded FPU block.

Large shifters are slow and occupy a lot of resources when implemented in the fine-grained FPGA fabric. Since we have large shifters present in the FPU blocks, we make these available as well. The shifters have dedicated input and output ports so that the shifters can be used at the same time as the integer adder.

To estimate the overhead of the dual-precision FPU and allowing access to the integer components, a circuit with the same structure as Fig. 9 was built, but instead of a dual-precision adder and multiplier, a conventional double-precision adder and multiplier was used and access to integer components was not provided. They were both synthesized using Synopsys Design Compiler with a 0.13-μm TSMC standard cell library to obtain area and delay. The area and critical path delay results are compared in Table II. A 31% area overhead and 20% delay overhead was observed. The overhead is relatively small considering the amount of flexibility that the modifications have given the multimode embedded FPU.

Table III shows a summary of the main differences between the conventional double-precision FPU and the multimode FPU that contribute to the overheads. The first column shows the main modules of the FPU. The second column lists the components where there are differences between the FPUs. The third and fourth columns show the differences in components between the FPUs. In the cases of MUXes and registers, the fourth column shows the number of additional muxes and registers required in the multimode FPU compared to the conventional FPU. In the case of adders, the third and fourth columns show the different adder configurations used in the conventional and multimode FPUs, and the fifth column shows the number of instances where they are different. For example, the seventh

TABLE III
DIFFERENCES IN COMPONENTS BETWEEN CONVENTIONAL DOUBLE-PRECISION FPU AND MULTIMODE FPU

	Component	Conventional FPU	Multi-mode FPU	#
Top Level	Mux 2:1	-	460	-
	Mux 3:1	-	170	-
	Register	-	118	-
FP Multiplier	Mux 2:1	-	1585	-
	Mux 3:1	-	26	-
	Mux 4:1	-	730	-
	Adder	11-b	2 $\times$ 8-b	3
	Adder	53-b	26-b, 27-b	1
FP Adder	Mux 2:1	-	393	-
	Mux 3:1	-	249	-
	Mux 4:1	-	64	-
	Adder	11-b	2 $\times$ 8-b	3
	Adder	53-b	26-b, 27-b	3

row indicates that in the floating-point multiplier, the multimode FPU has three pairs of 8-bit adders in place of three 11-bit adders when compared to the conventional FPU. The eighth row indicates that in the floating-point adder, in place of a 53-bit adder, the multimode FPU has 26-bit and 27-bit adders. Note that the numbers in Table III are obtained pre-synthesis, so the synthesis tools may have performed optimizations that may have altered the numbers. Table III shows that a significant number of additional MUXes are required to enable the additional functionality.

To verify the design, simulations were performed using ModelSim. It was tested against SoftFloat [26], an IEEE754 compliant, publicly available open-source floating-point software emulation package, which includes a program that generates test patterns. This included random inputs, as well as specific patterns to cover border conditions and rounding cases. Single- and double-precision modes were tested and verified.

V. FPGA ARCHITECTURE

The proposed architecture for the FPGA with embedded multimode FPUs is an island-style FPGA structure based on the Xilinx Virtex-II. The embedded FPUs would be distributed in a regular arrangement around the FPGA, surrounded by fine-grained configurable logic blocks (CLBs) as illustrated in Fig. 10. The number and arrangement of the the embedded FPUs would be decided by the FPGA vendor and would depend on the size of the FPGA. In this work, embedded FPUs are placed evenly spaced in two columns near the center of the FPGA.

VI. MODELLING

Beauchamp [7] used a VPR [27] methodology to model embedded FPUs in an FPGA. There are several downsides of using the VPR methodology. VPR can only roughly approximate commercial FPGA architectures. Commercial synthesis

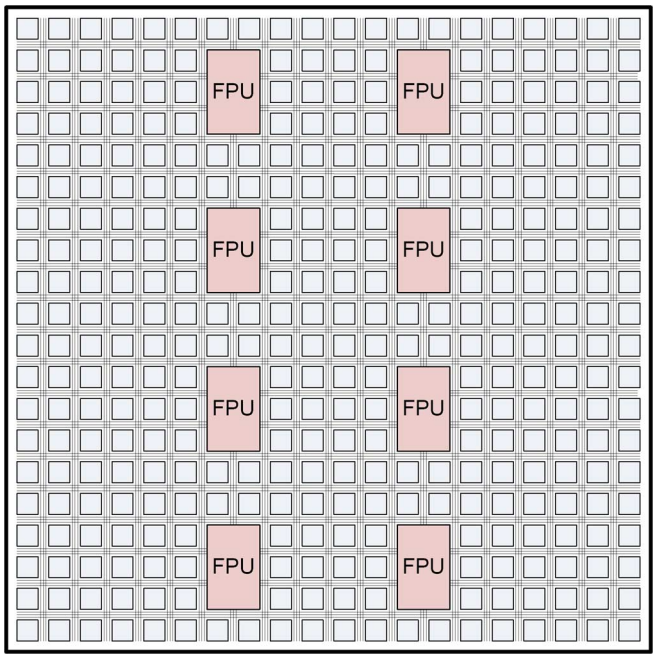

Fig. 10. Architecture of FPGA with embedded FPU blocks. Small squares represent CLBs. Note that this diagram is for illustration purposes only and is not to scale.

tools cannot be used with the VPR flow, so optimizations such as retiming are unavailable. Ho *et al.* [8] presented a modelling methodology, which they call the virtual embedded block (VEB) methodology, that overcomes these issues by modelling the embedded blocks on commercial FPGAs. By modelling on commercial FPGAs, a more accurate model is possible and comparisons with commercial FPGAs are more accurate. The VEB methodology uses a commercial tool flow, so optimizations like retiming are available. The VEB methodology was chosen to model our FPGA architecture because of the above mentioned advantages and for the convenience of using the vendor's toolset.

The VEB methodology involves first synthesizing the embedded block using a standard cell ASIC flow. The multimode embedded FPU was written in structural VHDL and synthesized using Synopsys Design Compiler. A 0.13-μm TSMC standard cell library was used to approximate the technology used by the Xilinx Virtex-II (0.12/0.15 μm process). Area and timing information for the embedded block were obtained from the synthesis tool. The case analysis option in Synopsys Design Compiler was used so that it would take the mode bits into account when generating timing results.

Using the area and delay results, we then construct a dummy logic block (VEB block) that approximates the size and delay of our embedded block. The VEB blocks can then be instantiated when building the benchmark circuits and placed in the Virtex-II device using area and location constraints.

Using the method in [8] of instantiating and adjusting the length of carry-chains, VEB blocks with a close approximation of the delay could be constructed. To emulate the area taken up by our embedded FPU when placed into the FPGA, we need to determine the area in terms of the number of slices. Since our target FPGA is the same as the one used by Ho in [9] (a Xilinx XC2V3000-6-FF1152), we use the same parameters in our calculations. The area of a slice is approximated to be 10 912 μm^2 from [9]. The area of our embedded FPU block was reported to be 264 065 μm^2 by the synthesis tools. Following the procedure described in [9], we assume a 15% area overhead after place and route and we determine the size of our embedded FPU block to be approximately 37 slices as shown in Table I.

A problem we encountered with using the VEB methodology to model our embedded FPU block was that because the VEB block is constructed with Virtex-II slices, the VEB block would not have enough input and output pins to model our embedded FPU block. The last two columns in Table I show the number of input and output pins of the Virtex-II slice and our embedded FPU (with the maximum number of pins supported in brackets). Each Virtex-II slice has 8 inputs and 2 outputs, so if we were to build a VEB block of 37 slices, it would support a maximum of 295 inputs and 74 outputs. However, our embedded FPU requires 396 inputs and 288 outputs. To support the required number of input and output pins, we enlarge the VEB block to 144 slices (to give a maximum of 1152 inputs and 288 outputs), which is considerably larger than the 37 slices reported by the synthesis tools. Therefore, area results presented in Section VIII are overly pessimistic.

VII. Experiments

In order to evaluate the proposed FPGA architecture, a set of benchmark circuits were built and implemented on our FPGA with embedded FPUs, and then compared to their implementation on a standard XC2V3000-6-FF1152 device. The benchmark circuits were a butterfly circuit (bfly), a digital sine cosine generator (dscg), a four-tap finite impulse response filter (fir4), a 3 $\times$ 3 matrix multiply circuit (mm3), and an ordinary differential equation solver (ode). The benchmark circuits were chosen to be the same as those used in [8] and [9]. Note that the benchmark circuits were built independently from [8] and [9], so results may not be directly comparable due to differences in implementation. Both single-precision and double-precision versions of each circuit were built in order to evaluate the multimode embedded FPUs in both precision modes. For the circuits implemented on the XC2V3000-6-FF1152 device, the circuits made use of the embedded multipliers. All the circuits were pipelined, except for the ode circuit, which was a multicycle implementation due to dependencies. Retiming in the synthesis tools was enabled for all benchmarks for improved performance. The tools used were Synplicity Synplify Pro 9.24 for synthesis and Xilinx ISE 9.2i for place and route.

For the implementations on our FPGA with embedded FPU blocks, we map the floating-point operations onto the VEB blocks. Placement constraints are used to force the place and route tool to place the embedded blocks in the desired locations and to prevent any other logic from being placed within the area occupied by the embedded blocks. The number and placement of the embedded FPU blocks are fixed on the FPGA, but only the blocks that are actually utilized by the benchmark are reported in the results.

To evaluate the benefits of the integer mode of the embedded FPUs, fixed-point versions of the benchmark circuits were built, which make use of the integer components in the embedded

TABLE IV
TABLE OF RESULTS—AREA AND DELAY RESULTS FOR DOUBLE-PRECISION BENCHMARK CIRCUITS

	FPGA without Embedded FPUs					FPGA with Embedded FPUs					Improvement	
	EMs	EM area (slices)	CLB area (slices)	Total area (slices)	Delay (ns)	FPUs	FPU area (slices)	CLB area (slices)	Total area (slices)	Delay (ns)	Area (×)	Delay (×)
bfly	36 (37%)	288	5625 (39%)	5913	34.82	4	576 (4%)	117 (0.8%)	693 (4.8%)	6.21	8.5	5.6
dscg	36 (37%)	288	3734 (25%)	4022	36.61	4	576 (4%)	61 (0.4%)	637 (4.4%)	6.85	6.3	5.3
fir4	36 (37%)	288	5597 (37%)	5885	40.64	4	576 (4%)	478 (3.3%)	1054 (7.4%)	6.38	5.6	6.4
mm3	27 (28%)	216	4410 (30%)	4626	34.33	3	432 (3%)	1055 (7.4%)	1487 (10.4%)	6.20	3.1	5.5
ode	18 (18%)	144	3402 (23%)	3546	43.10	3	432 (3%)	242 (1.7%)	674 (4.7%)	6.74	5.3	6.4
										Geometric mean	5.5	5.8

TABLE V
TABLE OF RESULTS—AREA AND DELAY RESULTS FOR SINGLE-PRECISION BENCHMARK CIRCUITS

	FPGA without Embedded FPUs					FPGA with Embedded FPUs					Improvement	
	EMs	EM area (slices)	CLB area (slices)	Total area (slices)	Delay (ns)	FPUs	FPU area (slices)	CLB area (slices)	Total area (slices)	Delay (ns)	Area (×)	Delay (×)
bfly	16 (16%)	128	2057 (14%)	2185	23.63	2	288 (2%)	28 (0.2%)	316 (2.2%)	5.93	6.9	4.0
dscg	16 (16%)	128	1267 (8%)	1395	26.78	2	288 (2%)	31 (0.2%)	319 (2.2%)	6.22	4.4	4.3
fir4	16 (16%)	128	1984 (13%)	1984	26.19	2	288 (2%)	254 (1.8%)	542 (3.8%)	5.84	3.9	4.5
mm3	12 (12%)	96	1487 (11%)	1487	20.85	2	288 (2%)	524 (3.7%)	812 (5.7%)	5.92	1.9	3.5
ode	8 (8%)	64	1237 (8%)	1301	30.22	2	288 (2%)	68 (0.5%)	356 (2.5%)	6.34	3.7	4.8
										Geometric mean	3.8	4.2

FPUs. For each benchmark, circuits were built for a range of different bit-widths (16, 24, 32, and 53 bits). The 24- and 53-bit cases were chosen because they utilize the full bit-widths of the integer components. The 16- and 32-bit widths were chosen because of their common usage and as cases when the bit-widths of the integer components are not fully utilized. For these experiments, the worst-case delay was used for each VEB block. For example, the same delay is used in the VEB block whether a 32-bit add or a 53-bit add is performed, even though the expected delay is lower if only a 32-bit add is performed. Therefore, the delay results would be conservative.

To provide further insight into the evaluation of the integer modes, a set of integer circuits were built on the XC2V3000-6-FF1152 device. These integer circuits perform the same operations as the integer operations of the embedded FPU. By comparing the number of slices occupied and the delay when the integer operations are implemented on a standard FPGA, we can gauge the usefulness of providing access to these integer components in the embedded FPU. These integer circuits were: two parallel 24 × 24-bit multipliers $(2 \times \ mult24)$, a 53 × 53-bit multiplier (mult53), 64-bit right shifter (rshift64), 54-bit left shifter (lshift54), 53-bit adder (adder53) and two parallel 26-bit and 27-bit adders (adder26,27).

VIII. RESULTS AND DISCUSSION

The results comparing the implementation of the benchmark circuits on a standard Xilinx XC2V3000-6-FF1152 without embedded FPUs against the implementation on our FPGA with embedded FPU blocks are presented in Tables IV and V. Table IV shows the results for double-precision versions of the benchmark circuits and Table V shows the results for single-precision versions of the benchmark circuits.

For both Tables IV and V, columns 2–6 show the resource utilization and delay when implementing the benchmark circuits on the Xilinx XC2V3000-6-FF1152 device. The second column shows the number of 18 × 18-bit embedded multipliers used. The third column shows the estimated area of the embedded multipliers used in terms of slices. From the Xilinx data-sheet [28], an embedded multiplier is four CLBs tall. The width is not published, but let us assume it is half a CLB wide. Each CLB contains four slices, so the area of an embedded multiplier is estimated to be equivalent to approximately eight slices. The fourth column shows the number of slices used for logic and the fifth column shows the total area. The percentages in brackets show the percentage of resources used out of the total resources of that type available on the FPGA. The sixth column shows the maximum delay.

Columns 7–11 show the resource utilization and delay when implementing the benchmark circuits on our FPGA with embedded FPU blocks. Column 7 shows the number of embedded FPU blocks used in the circuit and column 8 shows the total area (in terms of slices) that those FPU blocks occupy. Column 9 shows the number of slices used in the fine-grained CLB fabric and column 10 shows the total area used (inclusive of the area occupied by the embedded FPUs). Column 11 shows the maximum delay of each circuit. The final two columns show the improvement gained by using the embedded FPUs.

Table IV shows that a mean of 5.5 times improvement in area and 5.8 times improvement in delay was obtained for the double-precision benchmarks. Table V shows that a mean of 3.8 times improvement in area and 4.2 times improvement in delay was obtained for the single-precision benchmarks. These are significant improvements in area and performance. Given the same amount of real estate occupied by an XC2V3000-6-FF1152 implementation, it would be possible to build up to five parallel circuits for each of the benchmarks using our embedded FPU blocks, all operating at up to five times the clock frequency.

Our results in Table IV are better than the embedded FPUs in Ho's VEB paper [8], which had means of 3.7 times area improvement and 4.4 times delay improvement. The differences could partly be because Ho estimated the area and delay of the embedded FPUs from FPUs in commercial processors, while

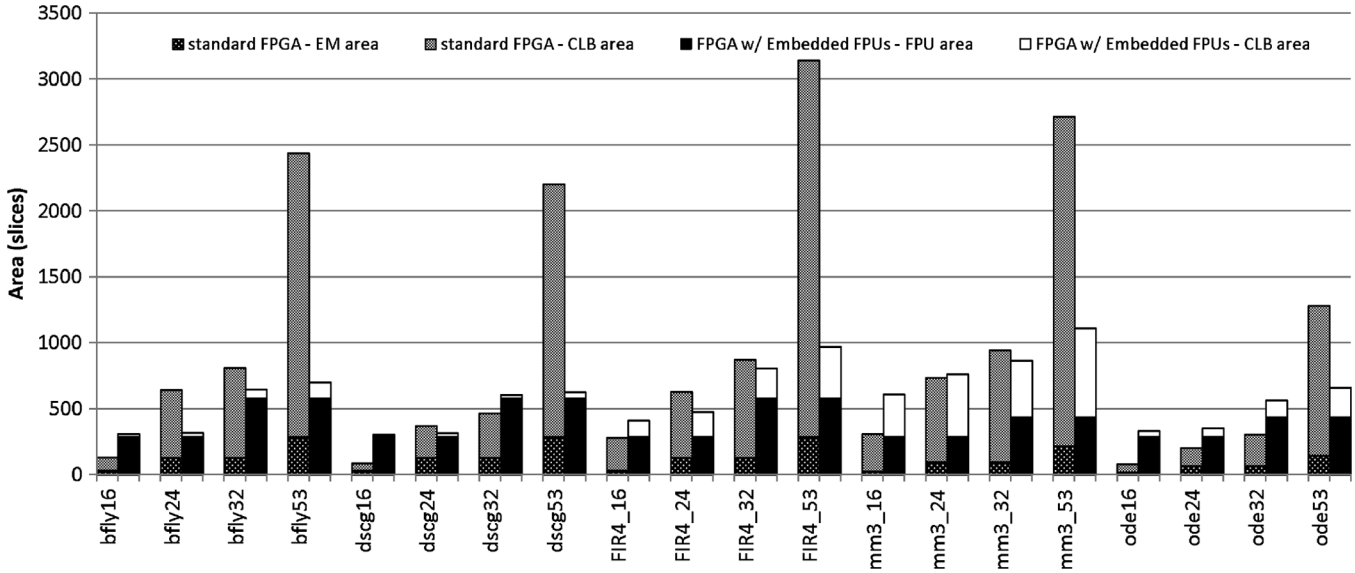

Fig. 11. Area results for fixed-point benchmarks, showing the estimated embedded multiplier (EM) and CLB usage area for a standard FPGA, compared to the embedded FPU and CLB usage area for the proposed FPGA.

we built and synthesized our FPUs, making our FPU models more accurate. Differences in implementation could also affect the results.

Our results in Table IV also compare well to the results obtained by Ho with their Hybrid FPGA [9], who obtained means of 18.3 times area improvement and 2.48 times delay improvement. There are several reasons for the differences in our area improvement compared to Ho's Hybrid FPGA. First, our area results are severely pessimistic because our embedded FPUs were enlarged significantly in order to overcome the limitations of the modelling technique as described in Section VI. Second, each of the coarse-grained units in Ho's Hybrid FPGA have two floating-point multipliers and two floating-point adders, in addition to configurable word-blocks, so they are able to map more logic to each coarse-grained block. The parameters of the Hybrid FPGA's coarse-grained units were also tuned to best-fit their set of benchmark circuits. Our embedded blocks are more generic, so only the floating-point operations are mapped to the embedded blocks. Third, our delay results were better, so it appears that the implementation of the circuits in Ho's work were tuned more for area savings than performance. Last, while we chose similar benchmark circuits as Ho's, our benchmark circuits were built independently, so differences between our implementations and Ho's could account for differences in results.

Our results in Table IV show greater improvement compared to Beauchamp's work [7], who obtained averages of 2.2 times area improvement and 1.35 times delay improvement by using embedded FPUs. However, direct comparison is difficult because Beauchamp used a different modelling technique (VPR), different tools, different benchmarks and estimated FPU area and delay from other FPUs.

As shown in Table V, the capability of our embedded blocks to perform two parallel single-precision additions and two parallel single-precision multiplications reduces the number of embedded blocks required to implement each benchmark circuit by up to 2 times. Alternatively, by using the same number of embedded FPU blocks as used by the double-precision benchmarks, the throughput could be doubled. This gives the designer an option of using double-precision if required, or choosing single-precision for increased throughput.

Figs. 11 and 12 show the area and delay results respectively for the fixed-point benchmark circuits. Fig. 11 shows the number of slices required to implement each benchmark using the standard XC2V3000-6-F1152 FPGA compared to the implementations on our FPGA with embedded FPUs. The proportion of area attributed to the embedded multipliers and embedded FPUs are also illustrated in Fig. 11. Each embedded multiplier is assumed to occupy a space equivalent to eight slices.

The results show that using the fine-grained CLBs and embedded multipliers on the standard XC2V3000-6-F1152 FPGA is more efficient for the small 16-bit circuits. This is to be expected because the embedded FPUs are of fixed size and would be under-utilized at 16-bits. Also, smaller circuits can be efficiently implemented on FPGAs because the carry-chains are not long and the multiplications can fit within the 18×18-bit multipliers. The standard FPGA was also relatively efficient for implementing the 16-, 24-, and 32-bit circuits and the results are comparable to the implementations using the embedded FPUs. When the bit-widths are increased to 53 bits, the amount of resources used increases dramatically. The main reason is that the embedded multipliers are only 18×18 bits, so to achieve a multiplication larger than 18×18 bits, the operands have to be split across multiple multipliers. In general, given $n \times n$-bit multipliers as building blocks, a $x.n \times x.n$-bit multiplier would require x^2 multipliers. Therefore, nine 18×18-bit multipliers are required to make a 53×53-bit multiplier. The partial products would then have to be summed, which also requires a very large amount of logic and routing resources.

The delay results in Fig. 12 show that the implementations using our embedded FPUs were faster in almost all cases, despite using the worst case (and hence pessimistic) delay in our

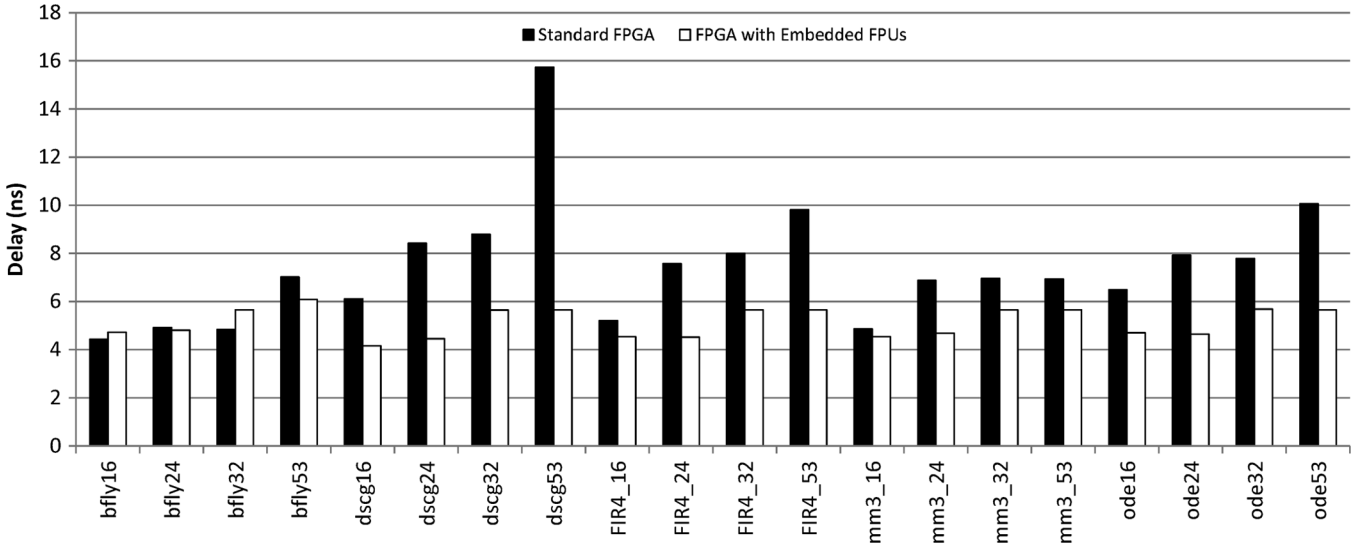

Fig. 12. Delay results for fixed-point benchmarks.

TABLE VI
TABLE OF RESULTS—AREA AND DELAY RESULTS FOR INTEGER CIRCUITS

	FPGA without Embedded FPUs					FPGA with Embedded FPUs		Improvement	
	EMs	EM Area (slices)	CLB area (slices)	Total area (slices)	Delay (ns)	FPU area (slices)	Delay (ns)	Area (×)	Delay (×)
2×mult24	8 (8%)	64	106 (1%)	170	4.681	144 (1%)	4.36	1.18	1.07
mult53	9 (9%)	72	389 (3%)	461	9.185	144 (1%)	5.63	3.20	1.63
rshift64	0 (0%)	0	343 (2%)	343	8.046	144 (1%)	2.25	2.38	3.58
lshift54	0 (0%)	0	225 (1%)	225	5.74	144 (1%)	2.11	1.56	2.72
adder53	0 (0%)	0	107 (1%)	107	4.695	144 (1%)	3.71	0.74	1.27
adder26,27	0 (0%)	0	84 (1%)	84	3.834	144 (1%)	3.22	0.58	1.19
							Geometric mean	1.35	1.71

VEB models. Dedicated large multipliers will always be faster than building a large multiplier out of smaller ones, and it shows in the results.

Table VI shows the area and delay of implementing integer circuits on the XC2V3000-6-FF1152 compared to the area and delay of the embedded FPU in its integer modes. In all cases, the embedded FPU outperforms the XC2V3000-6-FF1152 implementations in terms of delay. The performance gains are modest because the XC2V3000-6-FF1152 implementation made use of the fast embedded multipliers for the multiplier circuits and the carry-chains for the adder circuits, which allowed it to keep up for the small circuits. However, with the larger adders and multipliers, the gap widens in favor of our embedded FPUs. This is because the XC2V3000-6-FF1152 implementation needs to combine more embedded multipliers together to form larger multipliers and the delay of the carry-chain increases linearly with carry-chain length. The shifters show the largest gains, justifying their inclusion. Large variable shifters are still not efficiently implemented on typical FPGA architectures. It is worth noting that the area comparisons in Table VI are pessimistic. First, with each embedded FPU, the multiplier, adder, and shifters are all available to use at the same time, i.e., the area of a single embedded FPU concurrently supports four different integer operations. Second, the area occupied by our embedded FPUs is overly pessimistic for the reason described in Section VI.

The performance of the integer modes of the embedded FPUs suggests that they may be useful resources to be embedded into FPGAs. The coarse-grained nature of the blocks may prevent them from completely replacing the finer-grained structures available in current FPGAs, but could complement them well. The multifunctional nature of the embedded FPUs make efficient use of silicon area.

Each embedded FPU block only occupies about 1% of the area of an XC2V3000-6-FF1152 FPGA, despite increasing the area of each FPU block by a factor of 3.9 (from 37 to 144 slices) to overcome the limitations of the VEB modelling methodology described in Section VI. In a real implementation, the area of each embedded FPU block should be less than 1%. Thus, the area overhead of implementing the flexible multimode FPUs on a commercial FPGA is not prohibitive.

IX. CONCLUSION

This paper presented a flexible multimode embedded floating-point unit for FPGAs. Each embedded FPU contains a dual-precision floating-point adder and multiplier, which can each perform one double-precision operation or two single-precision operations in parallel. The output of the floating-point multiplier can be internally linked to an input of the floating-point adder to perform a fused multiply-add operation. To further increase flexibility of the embedded FPU, access to integer components of the FPU are provided,

including a large and fast integer multiplier, adder and two shifters. This paper also presented a novel design for a dual-precision floating-point multiplier. Results show that the FPGA with embedded multimode FPUs provide considerable performance and area benefits in single-precision, double-precision, fixed-point, and integer applications. We expect the benefits of the embedded multimode FPU to scale accordingly when implemented on the latest FPGAs, such as the Xilinx Virtex-5 FPGA, since the technology shrink will allow for shorter delays and greater density.

X. Future Work

Future work includes exploring other ways to improve flexibility, such as reconfiguring components within the FPU to perform other functions or adding more multifunction hardware to the embedded blocks. Making the components used by the FPU more general-purpose is possible at the expense of performance and/or area, so an exploration of these tradeoffs may be worthwhile.

In our current design, when in the floating-point multiply-add mode, the result of the multiplication is rounded prior to the addition. This could cause rounding errors to build up. A dedicated floating-point MAC unit does not round the product result prior to addition, but requires a different and more complex design. Future work could see the design modified to allow an unrounded product to be fed to the floating-point adder to minimize rounding error, like in a dedicated floating-point MAC unit.

References

[1] Y. Dou, S. Vassiliadis, G. Kuzmanov, and G. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *Proc. ACM/SIGDA 13th Int. Symp. Field-Program. Gate Arrays*, 2005, pp. 86–95.

[2] K. S. Hemmert and K. D. Underwood, "An analysis of the double-precision floating-point FFT on FPGAs," presented at the ACM Int. Symp. Field Program. Gate Arrays, Monterey, CA, Feb. 2004.

[3] G. Govindu, S. Choi, V. K. Prasanna, V. Daga, S. Gangadharpalli, and V. Sridhar, "A high-performance and energy efficient architecture for floating-point based LU decomposition on FPGAs," in *Proc. 11th Reconfigurable Arch. Workshop (RAW)*, Santa Fe, NM, Apr. 2004, p. 149a.

[4] M. de Lorimer and A. DeHon, " Floating point sparse matrix-vector multiply for FPGAs," in *Proc. ACM Int. Symp. Field Program. Gate Arrays*, Monterey, CA, Feb. 2005, pp. 75–85.

[5] Convey Computer Corporation, "Convey computer," Richardson, TX, 2008–2010 [Online]. Available: http://www.conveycomputer.com/

[6] K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," in *Proc. ACM/SIGDA 12th Int. Symp. Field Program. Gate Arrays*, Monterey, CA, Feb. 2004, pp. 171–180.

[7] M. J. Beauchamp, S. Hauck, and K. S. Hemmert, "Embedded floating-point units in FPGAs," in *Proc. IEEE Symp. Field Program. Gate Arrays (FPGA)*, 2006, pp. 12–20.

[8] C. H. Ho, P. H. W. Leong, W. Luk, S. J. E. Wilton, and S. Lopez-Buedo, "Virtual embedded blocks: A methodology for evaluating embedded elements in FPGAs," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2006, pp. 35–44.

[9] C. H. Ho, C. W. Yu, P. H. W. Leong, W. Luk, and S. J. E. Wilton, "Domain-specific hybrid FPGA: Architecture and floating point applications," in *Proc. Int. Conf. Field Program. Logic Appl. (FPL)*, 2007, pp. 196–201.

[10] J. Rose, "Hard vs. Soft: The central question of pre-fabricated silicon," in *Proc. 34th Int. Symp. Multiple-Valued Logic (ISMVL)*, 2004, pp. 1–4.

[11] A. Akkaş, "Dual-mode quadruple precision floating-point adder," in *Proc. 9th Euromicro Conf. Digit. Syst. Des. (DSD)*, 2006, pp. 211–220.

[12] A. Akkaş and M. J. Schulte, "A quadruple precision and dual double precision floating-point multiplier," in *Proc. Euromicro Symp. Digit. Syst. Des. (DSD)*, 2003, p. 76.

[13] G. Even, S. M. Mueller, and P.-M. Seidel, "A dual precision IEEE floating-point multiplier," *Integr. VLSI J.*, vol. 29, no. 2, pp. 167–180, 2000.

[14] P. C. Diniz and G. Govindu, "Design of a field-programmable dual-precision floating-point arithmetic unit," in *Proc. Int. Conf. Field Program. Logic Appl. (FPL)*, 2006, pp. 1–4.

[15] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754, 1985.

[16] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, 3rd ed. San Francisco, CA: Morgan Kaufmann, 2005, ch. H.5.

[17] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 2, no. 1, pp. 124–128, Mar. 1994.

[18] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, and T. Sumi, "Leading-zero anticipatory logic for high speed floating-point addition," *IEEE J. Solid-State Circuits*, vol. 31, no. 8, pp. 157–1164, Aug. 1996.

[19] M. Farmwald, "On the design of high performance digital arithmetic units," Ph.D. dissertation, Dept. Elect. Eng., Stanford Univ., Stanford, CA, Aug. 1981.

[20] M. Nicolaidis and R. O. Duarte, "Fault-secure parity prediction booth multipliers," *IEEE Des. Test*, vol. 16, no. 3, pp. 90–101, Jul. 1999.

[21] W.-C. Yeh and C.-W. Jen, "High-speed booth encoded parallel multiplier design," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 692–701, Jul. 2000.

[22] J. A. Hidalgo, V. Morena-Vegara, O. Oballe, A. Gago, A. Daza, and M. J. Martin-Vázquez, " A Radix-8 multiplier unit design for specific purpose," in *Proc. XIII Conf. Des. Circuits Integr. Syst.*, Madrid, Spain, Nov. 1998.

[23] A. D. Booth, "A signed binary multiplication technique," *Quarterly J. Mechan. Appl. Math.*, vol. 4, pp. 236–240, 1951.

[24] H. Sam and A. Gupta, "A generalized multibit recoding of two's complement binary numbers and its proof with application in multiplier implementations," *IEEE Trans. Comput.*, vol. 39, no. 8, pp. 1006–1015, Aug. 1990.

[25] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electron. Comput.*, vol. EC-13, no. 1, pp. 14–17, Feb. 1964.

[26] J. R. Hauser, "SoftFloat," Berkeley, CA, 2010 [Online]. Available: http://www.jhauser.us/arithmetic/SoftFloat.html

[27] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proc. 7th Int. Workshop Field-Program. Logic Appl.*, 1997, pp. 213–222.

[28] Xilinx, San Jose, CA, "Virtex-II platform FPGAs: Complete data sheet," 2007. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf

Yee Jern Chong (S'08) received the B.E. degree with first-class honors in electrical & electronic engineering from the University of Canterbury, New Zealand in 2002; the M.Eng.Sc. degree in electrical engineering (microelectronics) and the Ph.D. degree in computer engineering from the University of New South Wales, Sydney, Australia in 2004 and 2009 respectively.

His research interests include design automation, embedded systems, computer architecture and floating-point arithmetic circuits.

Sri Parameswaran (M'92) received the B.E. degree from Monash University, Australia, in 1986 and the Ph.D. degree from the University of Queensland, Australia, in 1991.

He is a Professor with the School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, where he also serves as the Program Director for Computer Engineering. His research interests include system level synthesis, low-power systems, high-level systems, and network on chips. He is also an Associate Editor of the Association for Computing Machinery *Transactions on Embedded Computing Systems* and the European Association for Signal Processing *Journal on Embedded Systems*.

Prof. Parameswaran has served on the program committees of numerous international conferences, such as the *Design Automation Conference*; *Design and Test in Europe*; the *International Conference on Computer Aided Design*; the *International Conference on Hardware/Software Codesign and System Synthesis* (as the Technical Program Committee Chair); and the *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*.