# An Asynchronous Floating-Point Multiplier

Basit Riaz Sheikh and Rajit Manohar
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853, U.S.A.
{basit,rajit}@csl.cornell.edu

***Abstract*—We present the details of our energy-efficient asynchronous floating-point multiplier (FPM). We discuss design trade-offs of various multiplier implementations. A higher radix array multiplier design with operand-dependent carry-propagation adder and low handshake overhead pipeline design is presented, which yields significant energy savings while preserving the average throughput. Our FPM also includes a hardware implementation of denormal and underflow cases. When compared against a custom synchronous FPM design, our asynchronous FPM consumes 3X less energy per operation while operating at 2.3X higher throughput. To our knowledge, this is the first detailed design of a high-performance asynchronous IEEE-754 compliant double-precision floating-point multiplier.**

***Keywords*-Floating point arithmetic; asynchronous logic circuits; very-large-scale integration; pipeline processing**

## I. Introduction

Energy-efficient floating-point computation is important for a wide range of applications. Traditionally, VLSI designers primarily relied on CMOS technology and voltage scaling to reduce power consumption [4]. With the transistor threshold voltage fixed [10], $V_{DD}$ has been scaling very slowly if at all, which means all performance improvements come at an increased energy consumption. Furthermore, process variations in deep sub-micron range have made devices far less robust, which is increasingly making it difficult for synchronous designers to overcome the problems associated with clock skew rates and clock distribution [6]. The findings of a recent in-depth study, to explore and devise ways to further scale supercomputer *petaFLOP* performance by 1000X, indicate the inadequacy of current design practices and technologies to achieve the desired throughput within a sustainable power budget [1]. This underscores a pressing need for alternate design practices, to reduce energy consumption for floating-point computations while preserving robust behavior in advanced technology nodes.

At the other end of the spectrum, embedded systems that have traditionally been considered low performance are demanding higher and higher throughput for the same power budget to support compute-intensive floating-point applications that improve the user experience. Since these applications have to be deployed on portable devices with limited battery-life, it is critical that we develop *energy-efficient* floating-point hardware for these embedded systems, not simply high performance floating-point hardware.

The IEEE 754 standard [19] for binary floating-point arithmetic provides a precise specification of floating-point number formats, computation operations, and exceptions and their handling. The combination of a vast range of inputs, special cases, and rounding modes makes the hardware implementation of fully IEEE 754 standard compliant floating-point arithmetic a very challenging task. Ignoring certain aspects of the standard can lead to unexpected consequences in the context of numerical algorithms. Hence, most floating-point hardware is IEEE-compliant or has an IEEE-compliant mode. The IEEE format specifies two main groups of floating-point format: *single-precision* and *double-precision*. In this work, we primarily focus on double-precision format since it is commonly used in most scientific and emerging applications.

We introduce a number of micro-architectural and circuit level optimizations to reduce power consumption in the floating-point multiplier (FPM) datapath. A floating-point multiplier consumes significantly more energy compared to a floating-point adder (FPA) [21,24]. This combined with the knowledge that the frequency of floating-point multiplication operations in emerging applications is similar to that of floating-point addition computations makes energy and power optimizations in the FPM datapath highly essential for an efficient full floating-point unit (FPU) design.

## II. Background and Related Work

In terms of micro-architectural complexity, the floating-point multiplier (FPM) datapath is simpler than the FPA datapath. The FPM datapath for double precision multiplication operation is shown in Figure 1. The double-precision inputs into the datapath, $A$ and $B$, comprise 1-bit of sign, 11-bits of exponent, and 52-bits of mantissa (also known as the significand) each.

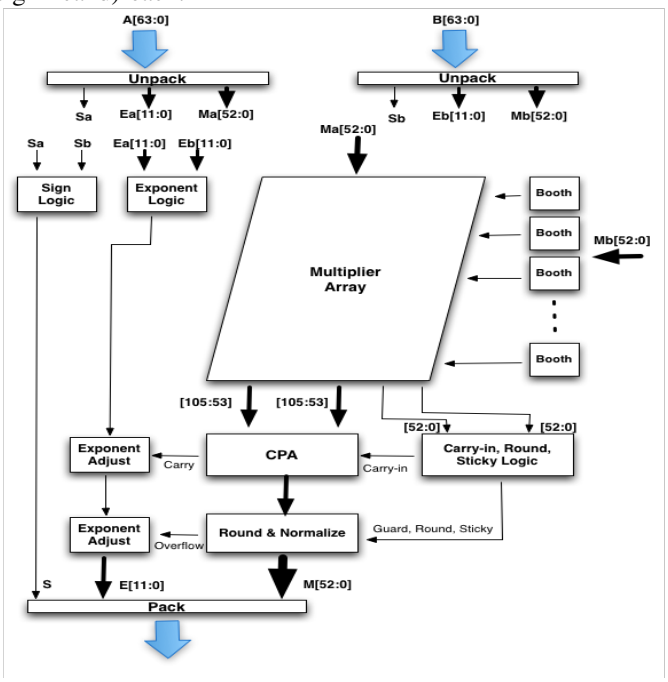


Fig. 1. Floating-point Multiplier Datapath

The following summarizes the key steps in an IEEE compliant FPM datapath:

- The first step in the FPM datapath is to unpack the IEEE representation and analyze the sign, exponent, and mantissa bits of each input to determine if the inputs are standard normalized or are of one of the special types (NaN, infinity, denormal).

- The mantissa bits are extended with the implicit bit. It is set to one for normal inputs and zero for a denormal input.
- The 53-bit long mantissas of both inputs are used to generate partial products corresponding to 106-bit product. Since high throughput and low latency are of essence in floating-point applications, most FPMs use some form of an array multiplier, such as a booth-encoded multiplier as shown Figure 1, to meet the performance demands. Most array multipliers employ an array of carry-save-adders (CSAs) [27] to reduce the large number of partial products to two final full product-length bit streams.
- The most significant 53-bits of the two output bit streams from the CSA array are summed up using a carry propagation adder (CPA) to generate a 53-bit mantissa. The least significant 53-bits are used to generate the carry input to the CPA as well as compute the guard, round, and sticky bits to be used in post normalization rounding.
- In parallel, the exponent logic computes the resulting exponent, which is a sum of the exponent values of both inputs minus the bias. The bias has a value of *1023* in case of double-precision operations. The sign of final product is also computed.
- The post multiplication step includes normalization of the 53-bit mantissa. For normal inputs and non-underflow cases, either the mantissa is already normalized or it may require a right shift by a single bit position, in which scenario the exponent is adjusted, in parallel, by adding one to it. The guard, round, and sticky bits are updated and are used, along with the round mode, to determine if the product needs to be rounded or not.
- In case of rounding, the mantissa is incremented by one. If rounding yields a carry out, the exponent is adjusted by adding one to it and right shifting the mantissa by one bit position.
- The final stage checks for a NaN, infinity, or a denormal outcome before outputting the correct result in the IEEE format.

With normalization step limited to a simple shift of no more than one-bit position and the exponent logic comprising only 11-bit long arithmetic, the FPM's complexity is largely a function of its *53x53 multiplier*, *sticky bit computation block*, and the final *carry propagation adder*. We present various structural and circuit-level optimization techniques to reduce the complexity and power consumption footprint of the aforesaid logic blocks.

### A. Asynchronous Multipliers and Floating-Point Arithmetic

In terms of the multiplier design, the delay variability nature of iterative multipliers makes them a popular choice amongst asynchronous designers [7,12]. An iterative multiplier utilizes a few functional units repeatedly to produce the result. In a simple iterative $n$ by $n$ multiplier implementation, where $n$ is the number of bits, the product is computed after $n$ iterations. Each iteration comprises a minimum n-bit addition and a serial shift by one-bit position. Furber et al. [13] proposed a low power integer multiplier which exploits the commonly occurring pattern of low number of significant bits in integer inputs as means to reduce the total number of iterations. These iterative multiplier designs, though compact in terms of area, are not feasible to be used in floating-point multiplier hardware due to their very high latency and low throughput and the fact that unlike the inputs in integer arithmetic, the most significant bits of floating-point mantissa inputs are non zero.

Joel Noche et al. [17] used asynchronous circuits in their design of a single-precision FPU. However, their FPU is completely non-pipelined, doesn't include any energy optimization techniques, and does not implement rounding logic. Their FPU has many orders of magnitude higher latency compared to all recent floating-point designs from synchronous domain. Sheikh et al. [24] employed fine-grain asynchronous circuit techniques for various operand-dependent optimization techniques to reduce average-case power consumption in the FPA datapath. However, their work is restricted to FPA design only.

### B. Synchronous Floating-Point Multipliers

There is a large body of work on synchronous FPM design. Ercegovac and Lang [8] contains an overview of the different techniques used to optimize floating-point multiplication. The focus of prior work has been the array multiplier block, which is the single largest logic structure within the FPM datapath. Earlier designs have employed various architecture and circuit-level optimizations to reduce array multiplier latency and increase its throughput [18,20,22,28]. However, there is relatively much less work on improving the energy efficiency of multiplier datapath [5], which is one of our primary contributions.

## III. Floating-Point Multiplier Power Breakdown

We use quasi-delay-insensitive (QDI) asynchronous circuits for our baseline FPM design. The fine-grain asynchronous pre-charge-enable-half-buffer (PCeHB) pipelines in our design contain only a small amount of logic (e.g. a two-bit full-adder). The actual computation is combined with data latching, which removes the overhead of explicit output registers. This pipeline style has been used in previous high-performance asynchronous designs, including a fully-implemented and fabricated asynchronous microprocessor [15].

Unlike in the FPA datapath where total power is distributed roughly evenly amongst a number of different logic blocks [24], the FPM's complexity is largely a function of its *53x53 multiplier*. This is highlighted in Figure 2 which shows the power breakdown estimates of our baseline fully QDI FPM datapath. The booth-encoded array multiplier accounts for roughly 76% of the total power consumption. Hence, in this work, we primarily focus on reducing energy/power of the array multiplier block.

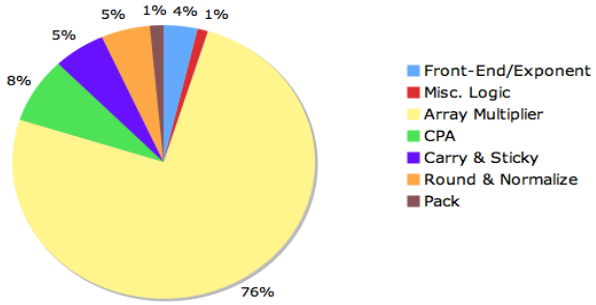


Fig. 2. FPM Pipeline Power Breakdown

The *Front-End/Exponent* block corresponds to the logic that unpacks IEEE format inputs and analyzes the sign, exponent, and mantissa bits of each input to determine if the inputs are standard normalized or are of one of the special types (NaN, infinity, denormal). It also includes the logic to compute the resultant exponent of the FPM product. The array multiplier outputs two 106-bit streams. The most significant 53-bits of the two output bit streams from the array multiplier are summed up using a carry propagation adder (CPA) to generate a 53-bit mantissa. The least significant 53-bits are used to generate the carry input to the CPA as well as compute the guard, round,

and sticky bits to be used in post normalization rounding. The *sticky bit computation block* and the final *carry propagation adder* are the other power consuming structures within the FPM datapath. In this work, we present various structural and circuit-level optimization techniques to reduce the complexity and power consumption footprint of the aforesaid logic blocks.

## IV. Multiplier Design Trade-offs

The choice of a particular multiplier design depends on a number of factors. These include: desired throughput, overall latency, circuit complexity, and the allowed power budget. Traditionally, high performance has been the key driving factor in multiplier design. However, as power consumption has become a major design constraint lately, a number of low-power multiplier designs have been proposed both in synchronous [5,11] and asynchronous domains [9,12,13].

### A. Iterative Multipliers

Iterative multipliers represent a low complexity design choice. An iterative multiplier utilizes a few functional units repeatedly to produce the result. Iterative multipliers can be used to reduce energy consumption by exploiting input data patterns; stages which add zero to the partial product could be detected in advance and skipped, hence reducing delay and energy consumption. Though compact in terms of area, iterative multipliers are not feasible to be used in floating-point multiplier hardware due to their very high latency and low throughput.

Reduction in the total number of partial products is the key goal of all multiplier optimization techniques, as it helps to reduce both latency as well as energy consumption. Along these lines, Efthymious et al. [7] proposed an asynchronous multiplier implementation based on the *original* Booth algorithm [3]. Their design scans the multiplier operand and skips chains of consecutive ones or zeros. This can greatly reduce the number of partial product additions required to produce the product. The downside is that it requires a variable length shifter to correctly align multiplicands for generating each partial product row. The effectiveness of this algorithm for high performance FPM hardware is dependent on the number of variable length shifts, which in turn depends on the number of partial product rows that are to be generated.

Our application profiling results for a number of scientific and emerging floating-point applications, using Intel's PIN [14] toolkit, indicate that although the *original* Booth algorithm is able to reduce the number of partial products from the maximum of 27, a sufficiently large number of partial products rows, more than 18 on average, still need to be generated, each of which requires the use of variable shifter. The latency overhead of such a large number of variable shift operations is too costly for any high performance FPM design. Hence, we did not pursue this algorithm any further.

### B. Array Multipliers

Array multipliers are the common choice for high throughput and low latency multiplication operations in most commercial FPM designs [20,26]. They produce a pre-determined fixed number of partial products, which greatly minimizes if not fully eliminates the opportunities for exploiting data dependent optimizations. For example, introducing logic to bypass a zero partial product instance may add the same amount of delay as summing the extra term in a carry save adder (CSA) used to reduce the partial product terms. As array multipliers present very limited opportunities for data dependent optimizations, there has not been much work on asynchronous array multiplier solutions.

The simplest implementation of an $n$ by $n$ array multiplier produces $n$ partial products in parallel, which are then summed up using CSAs. The large number of partial products makes this simple design unfeasible for both latency and power consumption perspective. As a result, many advanced multiplier implementations from academia [21] and industry [20,26,28] use some form of radix-4 modified booth algorithm, which cuts the number of partial products to $n/2$. The reduction in the number of partial products yields significant savings in energy consumption, latency, as well as the total transistor count.

For a 53x53-bit multiplier in an FPM datapath, with inputs $Y$ and $X$, a radix-4 booth-encoded algorithm produces 27 partial products. Each of the $Y$ and $X$ inputs is in a radix-4 format. The multiplier bits, $X$, are used to generate booth control signals for each partial product row. One of the big advantages of radix-4 booth multiplication is the relative simplicity of the logic which generates partial product rows. The only multiples of the multiplicand that are needed are: $0$, $\pm\ Y$, and $\pm\ 2Y$. Partial product term $Y$ is generated by simply assigning it the multiplicand. The $2Y$ multiple can be generated with relative ease by assigning it one bit right shifted value of the multiplicand. Bitwise inversion is used to generate complemented multiples. To reduce these 27 partial product rows to two partial product rows, a reduction tree comprising 7 stages of 3:2 counters/carry-save-adders (CSAs), is usually employed [27].

The energy consumption of the multiplier array is directly correlated to the number of partial product terms. With more partial product terms, more logic is needed first to produce those terms and then to sum and reduce those terms using a reduction tree. To further improve energy efficiency, one of the alternatives is to use a radix-8 Booth-encoded multiplier which reduces the number of partial product rows from 27 down to 18. The biggest disadvantage of a radix-8 multiplier is that it requires a $3Y$ multiple which needs a full length carry propagation adder to compute. Since the $3Y$ multiple must be available before any partial product term is computed, a tree adder topology such as a hybrid Kogge-Stone carry-select adder [27] must be used to minimize any latency degradation in a synchronous design.

Table I compares three different radix length implementations of a 53x53-bit multiplication unit in terms of the total partial products bits and the number of logic stages required to reduce the total number of partial product rows to two rows. A radix-8 Booth-encoded implementation produces 62.4% and 31.3% less partial products bits compared to bitwise radix-2 and Booth-encoded radix-4 multipliers respectively. But in terms of latency, when compared to a radix-8 version, a radix-4 implementation needs only one extra logic stage because partial product terms are summed and reduced using CSAs in a tree structure, which has logarithmic logic depth. This gives a radix-8 multiplier a single logic stage cushion to compute the tough $3Y$ multiple. Hence, for any radix-8 Booth multiplier to be considered a viable alternative, it must provide a very low latency $3Y$ computation unit with energy consumption significantly lower than the savings attained with the use of 31.3% less partial product bits. The use of power intensive tree adders greatly diminishes the savings that result from the reduction in the number of partial product terms. As a result, radix-8 multipliers are not commonly used in synchronous FPM implementations.

TABLE I
ARRAY MULTIPLIER

| Multiplier Type | Partial Product Bits | Reduction Stages |
|---|---|---|
| Radix-2 Bitwise | 2809 | 9 |
| Radix-4 Booth | 1539 | 7 |
| Radix-8 Booth | 1056 | 6 |

## V. 53X53-BIT RADIX-8 ARRAY MULTIPLIER

### A. 3Y Adder

The highly operand dependent nature of the *3Y* multiple computation makes it a strong potential target for asynchronous circuit optimizations. The application profiling results in Figure 3 show that the longest carry chain in a radix-4 *3Y* ripple-carry addition is limited to 3 ripple positions for over 90% of the operations across most floating-point application benchmarks. The delay of an adder depends on how fast the carry reaches each bit position. For input patterns that yield such small carry chain lengths on average, we need not resort to an expensive tree adder topology designed for the worst-case input pattern of carry propagating through all bits.

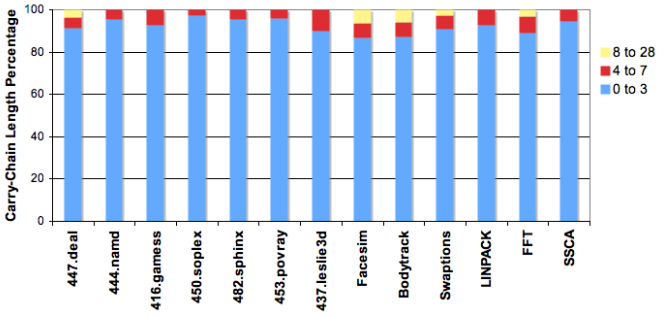


Fig. 3. Radix-4 3Y Adder Longest Carry Length

The *interleaved* adder topology provides an energy efficient solution for computing the bottleneck *3Y* multiple term required in radix-8 Booth multiplication. It comprises two 53-bit radix-4 ripple-carry adders, where each *3Y* block shown in Figure 4 computes the *3Y* multiple for the corresponding *Y* input. The first arriving data tokens *YRs* are forwarded to the *right 3Y* adder. In standard PCeHB reshuffling, the *interleave split* stage has to wait for the acknowledge signal from ripple-carry adder before it can enter neutral stage and accept new tokens. However, this would cause the pipeline to stall in case of a long carry chain. The *interleaved* adder topology circumvents this problem by instead issuing the next arriving data tokens to the *left 3Y* adder. Hence, the two ripple-carry adders could be in operation at the same time on different input operands. The *interleave merge* stage receives outputs from both *right* and *left* adders and forwards them to the next stage in the same interleaved order. With our pipeline cycle time of approximately 18 logic transitions (gate delays), the next data tokens for the *right* adder are scheduled to arrive after 36 transitions of the first one. This gives ample time to quite rare inputs with very long carry-chains to ripple through as well without causing any throughput stalls.

For inputs patterns observed in our various floating-point application benchmarks, the forward latency of computing the *3Y* term using the *interleaved* adder is less than that attained with power-intensive tree adders, which are frequently used in synchronous designs to guarantee low latency computation. Compared to a 53-bit hybrid Kogge-Stone carry-select tree adder implementation, the *interleaved* adder consumes approximately 68.1% less energy at 8.3% lower latency for the average case input patterns shown in Figure 3. We exploit this data dependent adder design topology, not possible within synchronous domain, to design an energy-efficient radix-8 Booth-encoded multiplier for our asynchronous FPM datapath.

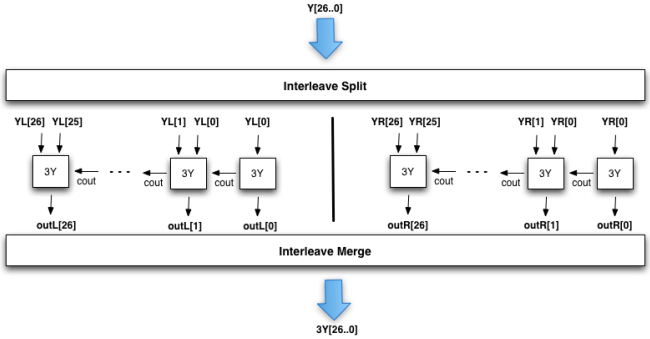


Fig. 4. Interleaved 3Y Adder

### B. Pipeline Design

Although, the radix-8 multiplier reduces the number of partial products bits by 31.3% compared to a radix-4 implementation, it still needs to produce and sum over 1050 partial product bits. As discussed by Sheikh et al. [25], the standard PCeHB pipelines, though very robust, consume considerable power in handshake circuitry, which gets worse as the complexity of PCeHB templates increases with more input and output bits. The handshake overhead, in a two-bit full adder PCeHB pipeline implementation, is as high as 69% of the total power consumption [25]. Therefore, for circuits with large number of inputs, intermediate and final outputs, such as a multiplier array, the PCeHB pipelines represent a non-optimum choice from energy efficiency perspective.

We use *N-Inverter* pipeline templates, first proposed in [25], to implement the multiplier array. An N-Inverter pipeline reduces the total handshake overhead by packing multiple stages of logic computation within a single pipeline block, in contrast to PCeHB template which contains only one effective logic computation per pipeline. The handshake complexity is amortized over a large number of computation stacks within the pipeline stage. Sheikh et al. [25] showed that compared to a PCeHB pipelined implementation the N-Inverter pipelines can reduce the overall energy consumption by 52.6% while maintaining the same throughput. These improvements come at the cost of some timing assumptions and require the use of single-track handshake protocol. The design trade-offs associated with N-Inverter templates are discussed extensively in [25].

The block-level pipeline breakdown of our radix-8 multiplier array is depicted in Figure 5 . The granularity at which the array is split is critical from both performance and energy efficiency perspective. The N-Inverter templates allow us to pack considerable logic within each stage, which helps to reduce the handshake associated power consumption significantly. However, as the number of logic computations within a pipeline block increase, so do the number of outputs. With more outputs, although the number of transitions per pipeline cycle remain the same with the use of wide NOR completion detection logic, each of these transitions incur a higher latency [25]. The choice of *8x4* pipeline blocks, with 15 outputs per each stage, was made to provide a good balance of low power and high throughput. The pipeline block labeled *8x4 Sign* is identical to an *8x4* block except that it includes a sign bit for each partial product row. The sign bit acts as an input of one in the least significant position for any of the cases involving a complemented partial product multiple of *-Y*, *-2Y*, *-3Y*, or *-4Y*. The pipeline blocks labeled *10x4 Sign Ext*

are similar in design to the frequent *8x4* block, except that it provides support for sign extension bits required for supporting complemented multiples. The *8x2* block is a reduced version of an *8x4* block with only two booth rows. The similarity between these different pipeline blocks and the frequent use of the *8x4* pipeline block provides us with great design modularity, which helped to reduce the overall design effort required to optimize the multiplier array for throughput and energy efficiency.

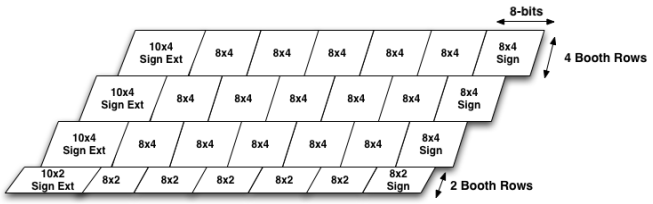


Fig. 5. Radix-8 Multiplier Array

Due to the similarity between different pipeline blocks, we only present the details of the *8x4* block. Each *8x4* pipeline block receives Booth-control, *Y* and *3Y* input tokens. The eight bits of *Y* and *3Y* inputs are encoded as four 1-of-4 tokens each. Figure 6 shows the intermediate and final logic outputs within an *8x4* pipeline. It also shows the corresponding mapping of these outputs to a simplified circuit level depiction of an N-Inverter pipeline template. The NMOS stacks in the first stage compute four rows of eight bit partial product terms in inverted sense. These inverted outputs drive the inverters in the second stage of the pipeline block to produce corresponding partial product, *PP*, outputs. The next stage of NMOS stacks implements carry-save addition logic [27] to sum and reduce these four rows of partial products to two rows of inverted sum and carry outputs. These inverted outputs drive the PMOS transistors in the last stage to produce sum and carry outputs, *SS* and *CC*, in correct sense for the following pipeline blocks.

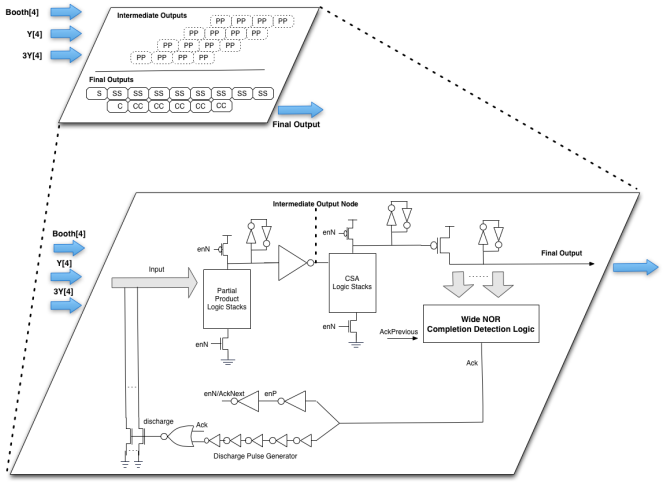


Fig. 6. 8x4 Multiply Logic Block

For array multiplication, all pipeline blocks have to be in operation in parallel. The parallel operation requires multiples copies of input tokens to be consumed simultaneously by multiple pipeline blocks. For example, each booth control token is required in seven different pipeline blocks. To facilitate this, we include multiple *copy* stages prior to initiating the array computation. These copy blocks generate the desired number of copies for each input token. These tokens are then forwarded to the pipeline blocks which consume them to produce sum and carry outputs.

The next computation step is the summation of the large number of *SS* and *CC* outputs that are produced in parallel. This summation step is commonly referred to as *reduction tree* in arithmetic literature. A *reduction tree* basically employs 3:2 counters, often referred to as carry-save-adders (CSAs), to sum and reduce three inputs to two outputs at each stage of the tree. Within a few stages, the large number of tokens spanning over many partial product rows are reduced to mere two 106-bit long rows, which are finally summed using a carry-propagation adder. We implemented a full 3:2 counter reduction tree [27] using multiple N-Inverter pipeline blocks. The NMOS stacks within each block implement carry-save addition logic. In terms of logic density, each pipeline block was restricted to produce no more than 15 outputs to maintain cycle time similar to *8x4* pipeline blocks.

The N-Inverter templates use single-track handshake protocol. As a result, the input tokens are first converted from four-phase handshake protocol into single-track protocol using conversion templates. This adds an additional logic stage to the FPM datapath latency. Since the final carry-propagation adder uses four-phase handshake protocol, the output tokens from the *reduction tree* are converted back to four-phase protocol. We hide the latency of this conversion stage by implementing the final stage of the *reduction tree* within these conversion templates.

The energy, latency, and throughput estimates of FPM implementations with radix-4 and radix-8 array multipliers are presented in Figure 7. The results are normalized to FPM datapath with a radix-4 multiplier. The 31.3% reduction in the number of partial product bits translates into 19.8% reduction in energy per operation. But this improvement in energy efficiency comes at a cost of 5.9% increase in the FPM latency because of the *3Y* partial product computation that needs to determined prior to initiating the multiplier array logic. A part of the *3Y* computation latency is masked within booth control token-generation and copy pipelines. Since the radix-4 multiplier requires one extra computation stage in the *reduction tree* compared to a radix-8 multiplier implementation, the latency overhead of the *3Y* computation can be further hidden. The 5.9% latency increase is attributed to the *3Y* multiple computation part which is not masked. Despite the increase in latency, the throughput for both implementations remains the same due to sufficient slack availability within the interleaved *3Y* computation block. The choice of a particular multiplier implementation represents a design trade-off. Since our goal was to optimize for energy consumption and throughput, we chose the radix-8 multiplier implementation in our final FPM design.

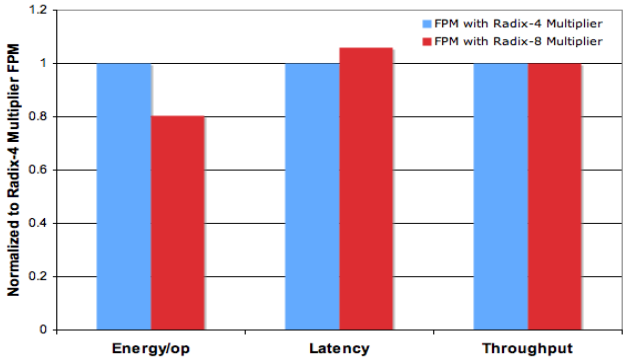


Fig. 7. Radix-4 Multiplier vs. Radix-8 Multiplier

## VI. Sticky-bit Logic and Carry-Propagation Adder

The multiplier array outputs two rows of 106-bit long partial sum and carry terms. The next step is to compute the 53-bit mantissa of the FPM output. This requires the summation of the most significant 53-bits of the two incoming partial sum and carry terms using a carry-propagation adder (CPA). The least significant 53-bits of the partial sum and carry terms are needed to compute the *carry* input into the CPA as well as the *guard*, *round*, and *sticky* [19] terms required during the rounding step.

### A. Carry Computation and Sticky-bit Logic

The multiplier array requires relatively less number of summation steps to produce its least significant output bits. This is because there are less partial product terms to be summed since each successive partial product row is skewed by three bit positions from the previous one in radix-8 multiplication. As a result, the least significant bits are available relatively earlier than rest of the multiplier array outputs. We take advantage of our fine-grain pipelining by initiating the carry computation as soon as the least significant bits arrive. Furthermore, the application profiling results in Figure 8 show that for over 90% operations across all applications the longest ripple-carry length to compute the carry input term is less than four radix-4 bit positions. These average-case patterns indicate that the carry term could be computed well in time for the CPA operation, hence alleviating the need of any speculative CPA implementations as is usually done in the case of most high performance synchronous FPMs.

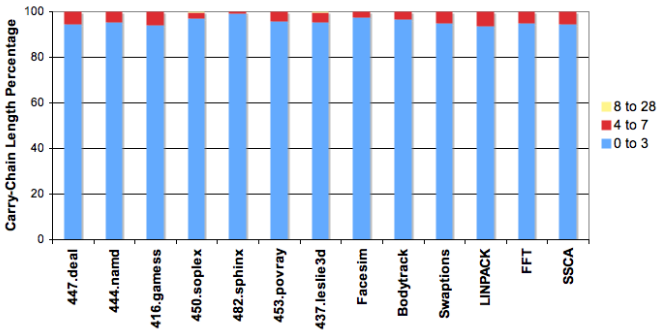


Fig. 8. Longest ripple-carry length for computing CPA carry input

The micro-architecture of carry and sticky-bit computation is depicted in Figure 9. It uses *interleaved* split and merge pipelines, first introduced with the design of *interleaved* adder. The inputs *A* and *B* in Figure 9 are in one-of-four encoded format and correspond to 52 least significant bits of partial sum and carry output terms from the multiplier array. The odd data tokens are sent on the output channels labeled with *R* prefix, while the next arriving even data tokens are sent on channels with *L* prefix. Each *Carry Sticky* block computes the carry and sticky bit terms at that bit position. With carry chain lengths of less than four, as seen in Figure 8, the final carry term is computed within four logic levels on average. This represents logarithmic average latency. The odd tokens are used to compute the carry term *cinR* used as carry input in the odd ripple-carry adder of our *interleaved* CPA, whereas the next arriving even data tokens compute the carry term *cinL* used as carry input in the even ripple-carry adder of our *interleaved* CPA topology.

For sticky-bit computation, we use parallel tree topology which combines bitwise sticky-bit values to compute the final sticky-bit. A ripple flow architecture similar to the one used to compute carry input term was deemed not feasible as it yielded consistently long ripple chains, which caused throughput degradation. Our *interleaved* topology prevents throughput degradation up to ripple lengths of *14* bit positions only. The application profiling results yield ripple lengths of *15* or more quite frequently. The sticky-bit is set to one if any of the bits is one, but for it to be set to zero it has to ensure that all prior bits in the sequence are zero. This is what causes the long ripple chains and renders ripple-flow design infeasible.

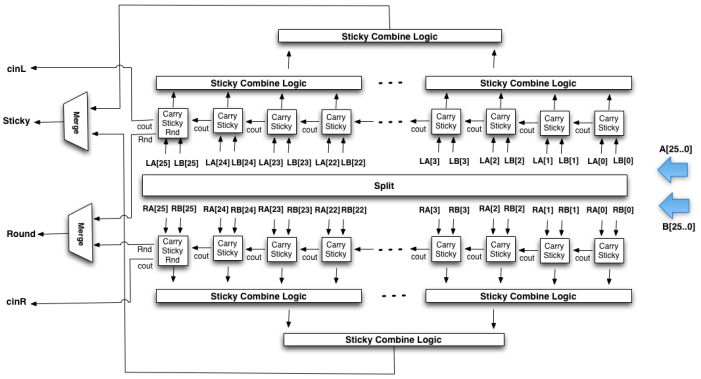


Fig. 9. Interleaved topology to compute sticky-bit and carry input

### B. 53-bit Carry-Propagation Adder

We harness the timing flexibility of our underlying asynchronous circuits by using *interleaved* adder topology for the 53-bit carry-propagation adder design. The *interleaved* adder comprises two ripple-carry adders. The adder topology is identical to the one used earlier for *3Y* multiple computation. Our choice of the *interleaved* adder was made on the basis of application profiling results, which indicate very small carry chain lengths on average across all application benchmarks. It yields average throughput similar to that attained with expensive tree adder designs while consuming up to 4X less energy per operation.

## VII. Denormal, Underflow, and Zero-input Case

While discussing the various trade-offs involved in the FPM datapath design, we have so far ignored certain special cases specified in the IEEE format [19]. Two of these special cases: the denormal numbers and underflow case represent the most difficult operations to implement in an FPM datapath. The scenarios under which these two special cases arise and the tasks that need to be performed are summarized as follows:

- One of the FPM inputs is a denormal number, which yields a mantissa with zeroes in its most significant bit positions. If the non-bias exponent for the product is greater than the minimum value of one, the product needs to be left shifted while decrementing the exponent until it is normalized or the exponent reaches the value of one. We refer to this scenario as the *Denormal* case.
- One of the FPM inputs is a denormal number or both FPM inputs are very small numbers and the resulting exponent is less than the minimum value of one. In this case, the mantissa needs to be right shifted. The value of right shift is equal to the difference between the minimum value and resulting exponent or an amount which zeroes out the mantissa, whichever of the two is smaller. We refer to this scenario as the *Underflow* case.

The need of variable left shift and right shift logic blocks makes the hardware support for denormal and underflow cases expensive. However, the infrequent occurrence of these special case inputs and the extensive hardware complexity required to support these operations has meant that many

FPM designs [16,28] do not fully support these operations in hardware. Instead, these operations are implemented in software via traps. This yields very long execution time [23]. It also means that the FPM hardware is no longer fully IEEE compliant.

We use serial shifters to provide full hardware support for these special case inputs. Using conditional split pipelines, the output bits from the CPA are directed to either *Normal* or *Denormal/Underflow* logic path. The *Normal* datapath includes single-bit normalization shift block and rounding logic. The *Denormal/Underflow* unit comprises serial left and right shift blocks and a combined rounding block. For input tokens diverted to the *Normal* datapath, no dynamic power is consumed within the *Denormal/Underflow* block and likewise for input tokens headed for *Denormal/Underflow* block, there is no dynamic power consumption in the *Normal* datapath. In contrast, synchronous design requires significant control overhead to attain fine-grain clock gating.

Once the mantissa has been correctly aligned using variable left or right shift block, a subsequent rounding operation may be required to increment the 53-bit mantissa by one. We utilize ripple-carry 1-of-4 encoded increment logic to implement rounding. An expensive increment logic topology would have been futile since the output from variable shift blocks arrives in bitwise fashion. The rounding logic is shared between the *Denormal* and *Underflow* datapaths as shown in Figure 10 to further minimize the area overhead of supporting these special case operations. The *Rnd* block receives incoming *guard*, *round*, *sticky*, and *rounding mode* bits from both special case datapaths. It selects the correct set of inputs to determine whether to increment the mantissa or not.

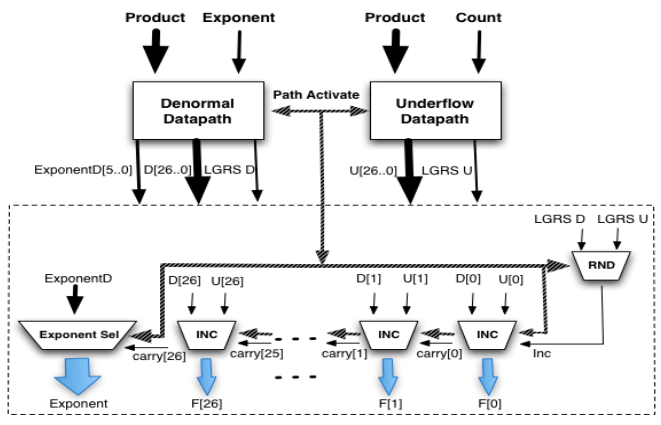


Fig. 10. Unified rounding hardware for denormal/underflow cases

Prior to the final *Pack* pipeline, there is a merge pipeline stage, which selects the output from either the *Normal* or the *Denormal/Underflow* datapath. Since these special case inputs happen very infrequently as shown in Figure 11, the throughput degradation due to the use of serial shifters does not effect the average FPM throughput.

### A. Zero-input Operands

Operand profile of floating-point multiplication instructions reveals that a few application benchmarks have a significant proportion of zero input operands. These primarily include applications with sparse matrix manipulations, such as *447.deal* and *437.leslie3d* [2], despite their use of specialized sparse matrix libraries. For other benchmarks, the zero-input percentage varies widely as shown in Figure 11. In most state-of-the-art synchronous FPM designs that we came across [21,26,28], the zero-input operands flow through the full FPM datapath. They yield similar latency and consume same power as any other non-zero operand computation. This is highly non optimum since if one or both of the FPM operands are zero, the final zero output could be produced much earlier and at much reduced energy consumption by skipping most of the compute intensive power consuming logic blocks such as the multiplier array, carry propagation adder, normalization, and rounding unit.

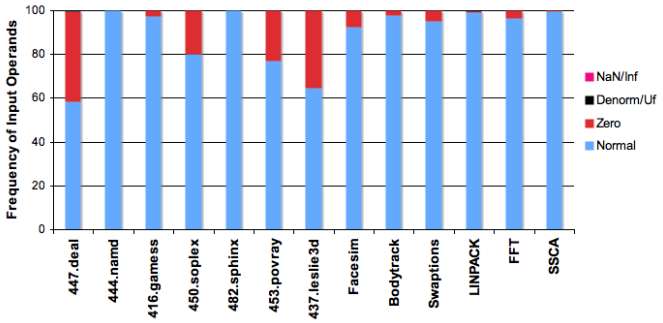


Fig. 11. Operand profile of floating-point multiplication instructions

We provide a zero bypass path in the FPM datapath to optimize its latency and energy consumption in the case of zero operands. To activate the bypass path, the FPM utilizes the zero flag control output from *Unpack* stage, which checks if any of the input operands is zero. But this information is not available in time before the start of pipeline stages pertaining to Booth control and *3Y* multiple generation. One possible solution was to delay these pipeline stages until the zero flag is computed and then use it to divert the tokens to either the regular or the bypass path. Since this solution incurs a latency hit for non-zero operands, it was discarded. In our design, instead of delaying the multiplier array, we inhibit the flow of tokens much deeper in the datapath. As a result, in our design the energy footprint of zero operand computations includes the overhead of computing Booth control token as well as some parts of the *3Y* multiple computation. But this still yields roughly 82% reduction in energy consumption for each zero operand computation, while preserving same latency and throughput for non-zero operand operations.

## VIII. Floating-Point Multiplier Evaluation

This section presents the SPICE simulation results of our proposed FPM datapath. The transistors in the FPM were sized using standard transistor sizing techniques [27]. To meet high performance targets and to minimize charge sharing problems, each NMOS stack was restricted to a maximum of four transistors in series. Since HSIM/HSPICE simulations do not account for wire capacitances, we included an additional wire load equivalent to a wire length of 8.75 µm in the SPICE file for every gate in the circuit. Our simulations use 65nm bulk CMOS process at 1V nominal $V_{DD}$ and typical-typical (TT) process corner.

For non-zero operands, the FPM registers a highest throughput of 1.53 GHz. In applications with a considerable percentage of zero operands, the average FPM throughput rises to as high as 1.78 GHz, since zero input operations skip throughput constraining N-Inverter templates in the multiplier array. The FPM energy per operation results across all application benchmarks are shown in Figure 12. Applications with considerable zero-input operands consume significantly less energy per operation as zero-input operations skip various logic blocks.

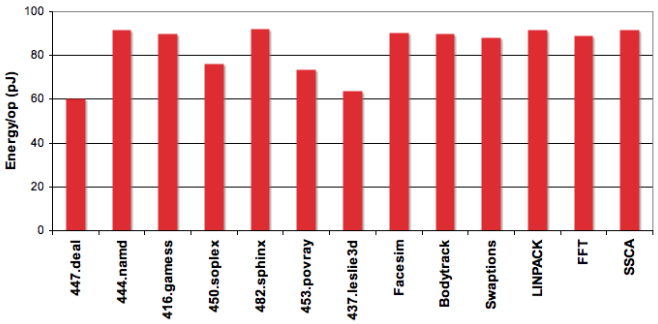


Fig. 12. FPM energy per operation across various floating-point applications

In Table II, we compare our proposed asynchronous FPM design against a custom FPM design by Quinnell et al. [21] in 65nm SOI process at 1.3V nominal $V_{DD}$. The energy, throughput, and latency results include only non-zero operand operations in order to provide the worst-case comparison. Despite using 65 nm bulk process, our FPM design consumes 3X less energy per operation while operating at 2.3X higher throughput. Both designs have similar latency at 1.3V. However, the custom FPM latency results do not include any internal pipeline latches, which account for a significant proportion of overall latency especially in high throughput designs. Our asynchronous FPM design compares quite favorably against the custom synchronous FPM implementation despite employing radix-8 Booth-encoded multiplier, which has an average 5.9% higher latency than a radix-4 Booth-encoded multiplier design.

TABLE II
ASYNCHRONOUS FPM VS SYNCHRONOUS FPM

| Design | Energy/op | Throughput | Latency @1.3V |
|---|---|---|---|
| Proposed FPM | 92.1 pJ | 1.53 GHz | 705 ps |
| Quinnell FPM | 280.8 pJ | 666 MHz | 701 ps |

For frequently occurring zero input operations in sparse matrix applications, our proposed FPM yields an even lower latency and energy per operation. The results for zero input operands are shown in Table III, which highlights the efficacy of zero bypass path.

TABLE III
ZERO OPERAND FEATURES

| Design | Energy/op | Latency |
|---|---|---|
| Proposed FPM | 15.8 pJ | 464 ps @ 1V |
| Quinnel FPM | 280.8 pJ | 701 ps @ 1.3V |

Since leakage power has become an important design constraint, our simulations model sub-threshold and gate leakage effects in detail. The total leakage power of our FPM in idle mode was estimated at 1.62 mW using typical-typical process corner at 90°C and a $V_{DD}$ of 1V.

## IX. SUMMARY

We presented the detailed design of an asynchronous high-performance energy-efficient IEEE 754 compliant double-precision floating-point multiplier. We provide thorough analysis of the trade-offs involved in using radix-4 and radix-8 array multiplier designs. The radix-8 design was preferred since it further reduced the total FPM energy consumption by 19.8% while preserving the average throughput. The full FPM datapath with numerous operand-dependent and pipeline optimizations is fully quantified using 65nm bulk process. When compared against a custom synchronous FPM design [21] in 65nm SOI process, it consumes 3X less energy per operation while operating at 2.3X higher throughput.

## REFERENCES


[1] Exascale computing study: Technology challenges. www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf.

[2] SPEC benchmark suite. www.spec.org.

[3] A. D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, June 1951.

[4] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4), July-August 1999.

[5] B. S. Cherkauer and E. G. Friedman. A hybrid radix-4/radix-8 low power signed multiplier architecture. *IEEE Transactions on Circuits and Systems*, 44(8), August 1997.

[6] W. J. Dally and J. Poulton. *Digital Systems Engineering*. Cambridge University Press, Cambridge, UK, 1998.

[7] A. Efthymious, W. Suntiamorntut, J. Garside, and L. E. M. Brackenbury. An asynchronous, iterative implementation of the original booth multiplication algorithm. In *Proceedings of the International Symposium on Asynchronous Circuits and Systems*, 2004.

[8] M. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan-Kaufmann, 2004.

[9] J. Hensley, A. Lastra, and M. Singh. A scalable counterflow-pipelined asynchronous radix-4 booth multiplier. In *Proceedings of the International Symposium on Asynchronous Circuits and Systems*, 2005.

[10] M. Horowitz. Scaling, power and the future of CMOS. In *Proceedings of the 20th International Conference on VLSI Design*, 2007.

[11] Z. Huang and M. D. Ercegovac. High-performance low-power left-to-right array multiplier design. *IEEE Transactions on Computers*, 54(3), March 2005.

[12] D. Kearny and N. W. Bergmann. Bundled data asynchronous multipliers with data dependent computation times. In *Proceedings of the Advanced Research in Asynchronous Circuits and Systems*, 1997.

[13] Y. Liu and S. Furber. The design of a low power asynchronous multiplier. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2004.

[14] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2005.

[15] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. V. Cummings, and T.-K. Lee. The design of an asynchronous MIPS R3000. In *Proceedings of Conference on Advanced Research in VLSI*, 1997.

[16] A. Naini, A. Dhablania, W. James, and D. D. Sarma. 1-ghz hal sparc65 dual floating point unit with RAS features. In *Proceedings of the International Symposium on Computer Arithmetic*, 2001.

[17] J. R. Noche and J. C. Araneta. An asynchronous IEEE floating-point arithmetic unit. *Proceedings of Science Diliman*, 19(2), 2007.

[18] S. F. Oberman, H. Al-Twaijry, and M. Flynn. The SNAP project: Design of floating point arithmetic units. In *Proceedings of the International Symposium on Computer Arithmetic*, 1997.

[19] The Institute of Electrical and Inc. Electronic Engineers. IEEE standard for binary floating-point arithmetic. ansi/ieee std 754, 1985.

[20] N. Ohkubo, M. Suzuki, T. Shinbo, T. Yamanaka, A. Shimizu, K. Sasaki, and Y. Nakagome. A 4.4 ns CMOS 54 x 54-b multiplier using pass-transistor multiplexor. *IEEE Journal of Solid-State Circuits*, 30(3), March 1995.

[21] E. Quinnell, Jr E. E. Swartzlander, and C. Lemonds. Floating-point fused multiply-add architectures. In *Proceedings of the Fortieth Asilomar Conference on Signals, Systems, and Computers*, 2007.

[22] E. M. Schwarz, R. M. Averill, and L. J. Sigal. A radix-8 cmos s/390 multiplier. In *Proceedings of the International Symposium on Computer Arithmetic*, 1997.

[23] E. M. Schwarz, M. Schmookler, and S. D. Trong. FPU implementations with denormalized numbers. *IEEE Transactions on Computers*, 54(7), July 2005.

[24] B. R. Sheikh and R. Manohar. An operand-optimized asynchronous IEEE 754 double-precision floating-point adder. In *Proceedings of IEEE International Symposium on Asynchronous Circuits and Systems*, 2010.

[25] B. R. Sheikh and R. Manohar. Energy-efficient pipeline templates for high-performance asynchronous circuits. *ACM Journal on Emerging Technologies in Computing Systems*, 7(4), November 2011.

[26] S. D. Trong, M. Schmookler, E. M. Schwarz, and M. Kroener. P6 binary floating-point unit. In *Proceedings of the International Symposium on Computer Arithmetic*, 2007.

[27] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, 2004.

[28] R. K. Yu and G. B. Zyner. 167 mhz radix-4 floating point multiplier. In *Proceedings of the International Symposium on Computer Arithmetic*, 1995.