The SNAP Project: Design of Floating Point Arithmetic Units

Stuart F. Oberman, Hesham Al-Twaijry, and Michael J. Flynn

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{oberman, hesham, flynn}@umunhum.stanford.edu

Abstract

In recent years computer applications have increased in their computational complexity. The industry-wide usage of performance benchmarks, such as SPECmarks, and the popularity of 3D graphics applications forces processor designers to pay particular attention to implementation of the floating point unit, or FPU. This paper presents results of the Stanford subnanosecond arithmetic processor (SNAP) research effort in the design of hardware for floating point addition, multiplication and division. We show that one cycle FP addition is achievable 32% of the time using a variable latency algorithm. For multiplication, a binary tree is often inferior to a Wallace-tree designed using an algorithmic layout approach for contemporary feature sizes (0.3μm). Further, in most cases two-bit Booth encoding of the multiplier is preferable to non-Booth encoding for partial product generation. It appears that for division, optimum area-performance is achieved using functional iteration, and we present two techniques to further reduce average division latency.

1 Introduction

A floating point number representation can simultaneously provide a large range of numbers and a high degree of precision. As a result, a portion of modern microprocessors is often dedicated to hardware for floating point computation. Previously, silicon area constraints have limited the complexity of the floating point unit, or FPU. Advances in integrated circuit fabrication technology have resulted in both smaller feature sizes and increased die areas. Together, these trends have provided a larger transistor budget to the processor designer. It has therefore become possible to implement more sophisticated arithmetic algorithms to achieve higher FPU performance.

The IEEE 754 floating point standard [1] is the most common floating point representation used in modern microprocessors. It dictates the precisions, accuracy, and arithmetic operations that must be implemented in conforming processors. The required precisions of the significands include single precision (24 bits) and double precision (53 bits), and support for double extended precision (≥ 64 bits) is recommended. The arithmetic operations include addition, multiplication, and division. The Stanford subnanosecond arithmetic processor (SNAP) research effort has studied the design of algorithms and implementations for high performance IEEE conforming floating point adders, multipliers, and dividers.

1.1 Design Space

The performance and area of a functional unit depend upon circuit style, logic implementation, and choice of algorithms. The space of current circuit styles ranges from fully-static CMOS designs to hand-optimized self-timed dynamic circuits. Logic design styles range from automatically-synthesized random logic to custom, hand-selected gates. We investigate performance and area tradeoffs at all levels.

The three primary parameters in FP functional unit design are latency, cycle time, and area. The functional unit latency is the time required to complete a computation, typically measured in machine cycles. Designs can be either *Fixed Latency (FL)* or *Variable Latency (VL)* [S1]. In a FL design, each step of the computation completes in lock-step with a system clock. Further, any given operation completes after a fixed quantity of cycles. The cycle time in a FL design is the maximum time between the input of operands from registers and the latching of new results into the next set of registers. In contrast, VL designs complete after a variable quantity of cycles. This allows a result to be returned possibly sooner than the maximum latency, reducing the average latency. They achieve their variability through either the choice of algorithm (VLA) or choice of circuit design (VLC). VLA designs operate in synchronization with a system clock. However, the total number of cycles required to complete the operation varies depending upon other factors, such as the actual values of the input operands. VLC designs need not have any internal synchronization with the rest of the system. Instead, such a design accepts new inputs at one time, and it produces results sometime later, independent of the system clock.

1.2 Organization

The following sections detail performance and area tradeoffs for floating point unit design. Section 2 is a case study of a VLA technique for increasing the performance of FP addition. Section 3 investigates how multiplication performance and area vary with technology. Specifically, the use of different partial product generating algorithms are analyzed, as are the choices for partial product reduction. Section 4 investigates the performance of floating point division. Section 5 summarizes the results.

2 Floating Point Addition

The most frequent FP operations are addition and subtraction, and together they account for over half of the total FP operations in typical scientific applications [S2]. Both addition and subtraction use the FP adder. Techniques to reduce the latency and increase the throughput of the FP adder have therefore been the subject of much previous research.

To further reduce the latency, we observe that not all of the components are needed for all input operands. Two VLA techniques are proposed to take advantage of this to reduce the average addition latency [S3]. To effectively use average latency, the processor must be able to exploit a variable latency functional unit. The processor might use some form of dynamic instruction scheduling with out-of-order execution in order to use the reduced latency and achieve maximum system performance.

2.1 Current Algorithms

FP addition comprises several individual operations. Higher performance is achieved by reducing the maximum number of serial operations in the critical path of the algorithm. In this study, the analysis assumes double precision operands.

A block diagram of a state-of-the-art FP adder is shown in Fig. 1. Adders similar to this architecture have been implemented in several commercial microprocessors [2], [3], [4]. This architecture exploits many aspects of the FP addition dataflow. It implements the significand datapath in

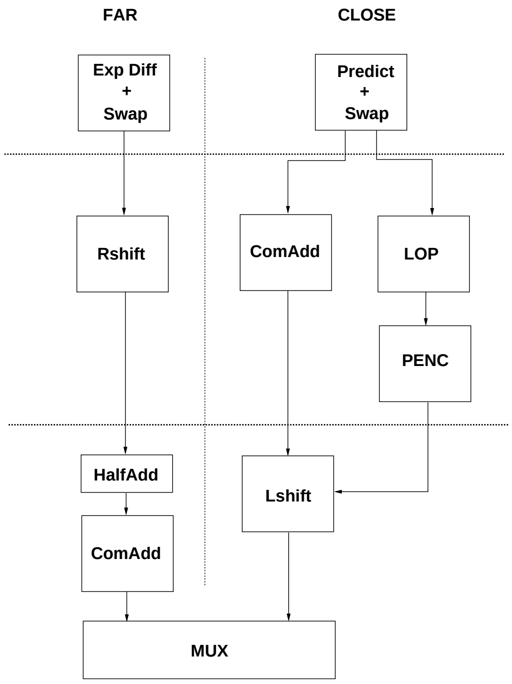

Figure 1. Three cycle pipelined adder with combined rounding

two parts: the *CLOSE* path and *FAR* path. For subtraction, when the exponents differ by more than 1 (FAR path), massive cancellation can not occur. Rather, there can be at most a 1 bit left-shift. Similarly, when the exponents differ by at most 1 (CLOSE path), massive cancellation may occur requiring a large normalizing left-shift, but no initial large aligning right shift is required. This allows the aligning right shift and the normalizing left-shift to be mutually exclusive, with only one such shift ever appearing on the critical path [5].

Another optimization made in this algorithm reduces the number of serial operations. In a straightforward implementation of the addition dataflow, rounding would be implemented by a separate series incrementer after all other operations. However, the realization can be made that the rounding step occurs very late in the computation, and it only modifies the result by a small amount. By precomputing all possible required results in advance, rounding and conversion can be reduced to the selection of the correct result [S4]. For the IEEE *round to nearest* (RN) rounding mode, the computation of $A + B$ and $A + B + 1$ is sufficient to account for all possible rounding and conversion possibilities. Incorporating this optimization into the algorithm requires that the significand adders in each path compute both *sum* and *sum+1*, typically through the use of a compound adder (ComAdd). Selection of the true result is accomplished by analyzing the rounding bits, and then selecting either of the two results. This optimization removes one significand addition step.

For the two directed IEEE rounding modes *round to positive* and *minus infinity* (RP and RM), it is also necessary to compute $A + B + 2$. The rounding addition of 1 ulp may cause an overflow, requiring a 1 bit normalizing right-shift. This is not a problem in the case of RN, as the guard bit (next less significant bit below the LSB) must be 1 for rounding to be required. Accordingly, the addition of 1 ulp will be added to the guard bit, causing a carry-out into the next most significant bit which, after normalization, is the LSB. However, for the directed rounding modes, the guard bit need not be 1. Thus, the explicit addition *sum+2* is required for correct rounding in the event of overflow requiring a 1 bit normalizing right shift. This additional result can be produced by using a row of half-adders above the *FAR* path compound adder, or by simply duplicating the two compound adders, where one adder computes *sum* and *sum+1* assuming that there is no carry-out and the other adder computes the same results assuming that a carry-out will occur.

Assuming that the mantissas are conditionally swapped based upon the true exponent difference, the smaller mantissa is always subtracted from the larger mantissa, except possibly in the *CLOSE* path for cases where the exponents are equal. However, in these cases, since there is no initial aligning right shift, the result is exact and no rounding is required. Further, by again precomputing both *sum* and *sum+1* in the significand adder, recomplementation can also be reduced to selection. The true subtraction of $A - B$ is accomplished by selecting *sum+1*, as the subtraction is implemented by $A + \overline{B} + 1$. If the carry-out of this addition is 0, then the result is negative requiring recomplementation. The complemented result is formed by bitwise inversion of *sum*, as

$$-(A - B) = \overline{A + \overline{B}}$$

Accordingly, recomplementation is reduced to a MUX and bitwise inversion.

Further performance improvement is achieved by computing the normalizing left-shift distance in the *CLOSE* path in parallel with the compound adder, rather than in series, using leading-one-prediction (LOP) and priority-encoding (PENC). An adder employing all of these optimizations in a high clock-rate microprocessor typically has a latency of three cycles. The critical path in this implementation is in the third stage consisting of the delays of the half-adder, compound adder, multiplexor, and drivers.

2.2 Variable Latency Algorithm

From Fig. 1, the long latency operation in the first cycle occurs in the *FAR* path. It contains hardware to compute the absolute difference of two exponents and to conditionally swap the mantissas. For IEEE double precision operands, the minimum latency in this path comprises the delay of an 11 bit adder and two multiplexors. The *CLOSE* path, in contrast, has relatively little computation. A few gates are required to inspect the low-order 2 bits of the exponents to determine whether or not to swap the mantissas, and a multiplexor is required to perform the swap.

Rather than letting the *CLOSE* path hardware sit idle during the first cycle, it is possible to take advantage of the duplicated hardware and initiate *CLOSE* path computation one cycle earlier. This is accomplished by moving both the second and third stage *CLOSE* path hardware up to their preceding stages. Since the first stage in the *CLOSE* path completes early relative to the *FAR* path, the addition of the second stage hardware need not result in an increase in cycle time. This is similar to the first cycle in the second generation DEC 21164 FP adder [4].

The operation of the proposed algorithm is as follows. Both paths begin speculative execution in the first cycle. At the end of the first cycle, the true exponent difference is known from the *FAR* path. If the exponent difference dictates that the *FAR* path is the correct path, then computation continues in that path for two more cycles, for a total latency of three cycles. However, if the *CLOSE* path is chosen, then computation continues for one more cycle, with the result available after a total of two cycles. While the maximum latency of the adder remains three cycles, the average latency is reduced due to the faster *CLOSE* path. If the *CLOSE* path is a frequent path, then up to 1/3 reduction in the average latency can be achieved.

Further reductions in the latency of the *CLOSE* path can be made after certain observations. First, the normalizing left shift in the second cycle is not required for all operations. A normalizing left shift can only be required if the effective operation is subtraction. Since additions never need a left shift, addition operations in the *CLOSE* path can complete in the first cycle. Second, in the case of effective subtractions, small normalizing shifts, such as those of two bits or less, can be separated from longer shifts. While longer shifts still require the second cycle to pass through the full-length shifter, short shifts can be completed in the first cycle through the addition of a separate small multiplexor. Both of these cases have a latency of only one cycle, with little or no impact on cycle time. If these cases occur frequently, the average latency is reduced. A block diagram of the variable latency adder is shown in Fig. 2.

2.3 Performance

This algorithm was simulated using operands from actual applications to determine its effectiveness. The data for the study was acquired using the ATOM instrumenta-

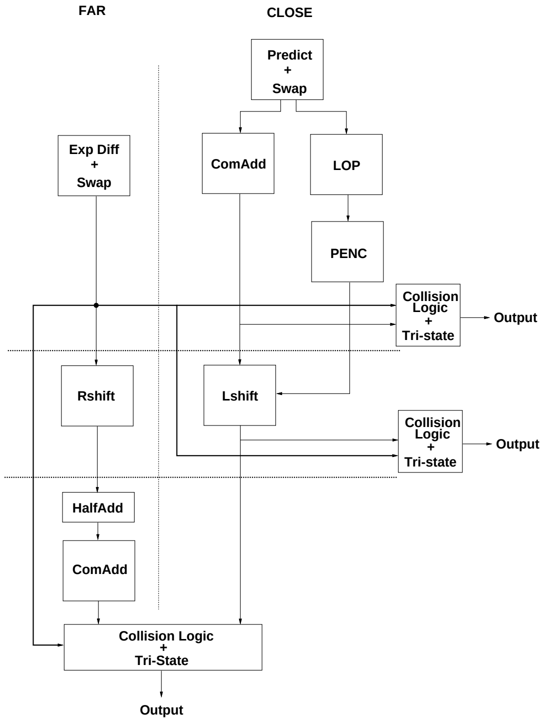

Figure 2. One, two, or three cycle variable latency adder

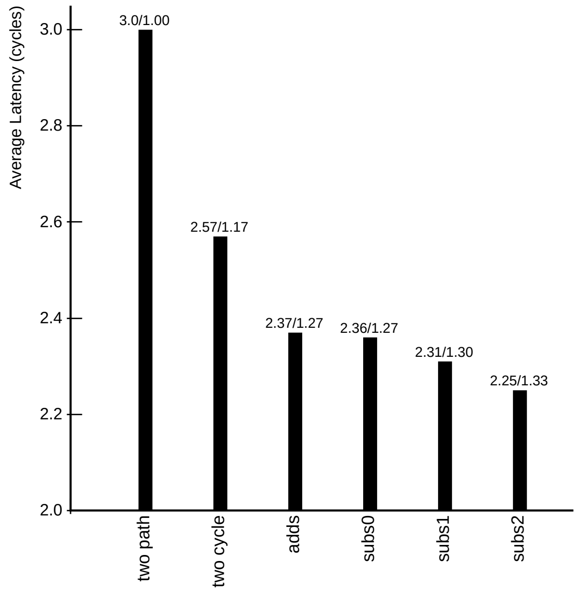

Figure 3. FP addition performance

tion system [6]. ATOM was used to instrument 10 applications from the SPECfp92 [7] benchmark suite which were then executed on a DEC Alpha 3000/500 workstation. The benchmarks used the standard input data sets. All double precision floating point addition and subtraction operations were instrumented. The operands from each operation were used as input to a custom FP adder simulator. The simulator recorded the effective operation, exponent difference, and normalizing distance for each set of operands.

The results show that 57% of the operations are in the *FAR* path and require three cycles, while 43% are in the *CLOSE* path and require at most two cycles. A comparison with a different study of floating point addition operands [8] on a much different architecture using different applications provides validation for these results. In that study over 30 years ago, six problems were traced on an IBM 704, tracking the aligning and normalizing shift distances. There 45% of the operands required aligning right shifts of 0 or 1 bit, while 55% required more than a 1 bit right shift. The similarity in the results suggests a fundamental distribution of floating point addition operands in scientific applications.

An analysis of the effective operations in the *CLOSE* path shows that the total of 43% can be broken down into 20% effective addition and 23% effective subtraction. A left shift less than or equal to 2 bits is required for 52.5% of the *CLOSE* path subtractions. In total, $20\% + (0.525) \times 23\% = 32\%$ of the operations can complete in the first cycle. The performance of the proposed techniques is summarized in Fig. 3. For each technique, the average latency is shown, along with the speedup provided over the base *Two Path* FP adder with a fixed latency of three cycles. By allowing effective additions in the *CLOSE* path to complete in the first cycle (adds), a speedup of 1.27 is achieved. For even higher performance, the most aggressive implementation (subs2) achieves a speedup of 1.33 by allowing all effective addition and those effective subtractions requiring normalizing shifts of two bits or less to complete in the first cycle. These techniques do not add significant hardware, nor do they impact cycle time. They demonstrate how a VLA architecture provides a reduction in average latency while maintaining single cycle throughput.

3 Floating Point Multiplication

3.1 Background

The speed of the FP multiplier is critical to the performance of an FPU. Multiplication is the process of adding the partial products. Multiplication algorithms differ in how they generate the partial products and how the partial products are added together to produce the final result.

Research on multiplier design has included techniques for partial product generation [9] and partial product reduction [10], [11], [12], [13], [14]. Most previous analyses of the partial product reduction trees use as the basis for their design a simple compressor delay model where the delay from each input of a compressor to each output is equal. Also, the delay due to interconnection is typically ignored. Unfortunately, such simple models do not accurately reflect the performance of actual implementations where not all inputs have the same delay and where the added delay due to interconnect is significant, especially for minimum feature sizes below 0.5μm. However, a simple delay model is sufficient for the design of a binary tree using 4-2 compressors, as the delay for all inputs of a 4-2 compressor are approximately equal.

Designing an optimized partial product array using (3,2) counters requires taking into account all delay components. Further, organizing the counters in order to minimize worst-case delay is not trivial. Therefore, an algorithmic approach to the design, using a sophisticated delay model that takes into account the interconnect delay due to counter placement and the different path delays, is extremely useful. We have implemented such an algorithm, based upon the approach of Oklobdzija [15]. The algorithm is essentially the same as that proposed by Oklobdzija, but it also takes into account interconnect delay due to counter placement and the different path delays. Our algorithm uses a complex delay model for the (3,2) counter, and it is further constrained by the availability of wiring tracks for the routing of each column of the partial product array [S5]. The number of wiring tracks available in a column is a function of the fabrication process and the floorplan of the multiplier. It is a fixed parameter for each column, and it limits the possible interconnections.

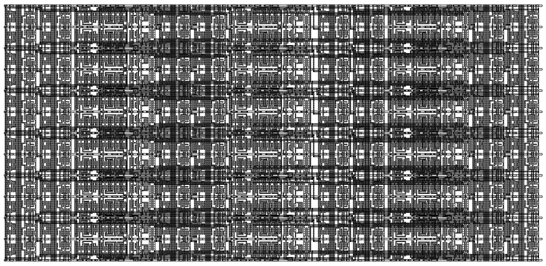

Figure 4. Multiplier layout from MAGIC for 16 bit slice of binary tree

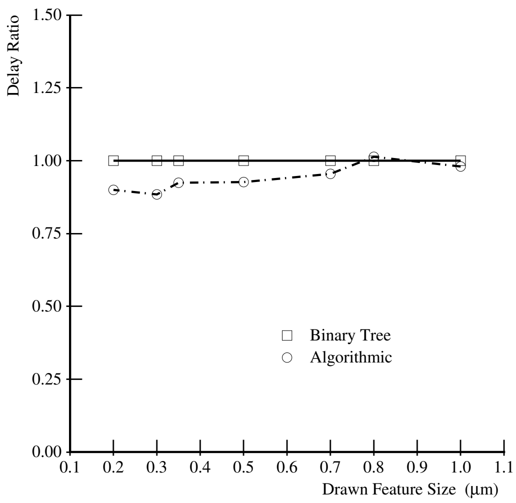

Figure 5. Relative delay: algorithmic layout to binary tree for non-Booth double precision

3.2 Methodology

In this study, we examined multiplier performance and area tradeoffs over combinations of several parameters: feature size (f=$1.0\mu m$ to $0.2\mu m$), counter configuration (3,2 and “4-2”), encoding scheme (non-Booth, Booth 2, and Booth 3), and significand precision (24b through 113b). For each category, we implemented a custom layout of a binary-tree multiplier using the MAGIC layout tool. Additionally, a unique (3,2) array was designed for every combination of feature size, encoding scheme, and significand precision. A portion of a binary-tree layout is shown in Fig. 4. Using extracted parasitics, we performed SPICE timing simulations for each combination of parameters. Each simulation included delays due to transistors as well as interconnect. The scalable SPICE model of McFarland [S6] was used to project results down to 0.2μm.

The relative performance of the algorithmically generated partial product array to binary trees for double precision non-Booth encoded multipliers is shown in Fig. 5. The graph shows that the latency of the binary tree and the algorithmically generated arrays are comparable for large feature sizes. This is because at the larger feature sizes interconnect does not significantly affect the total delay for the binary tree. The algorithmic approach is also better able to hide the interconnection delay. However, at smaller feature sizes, the algorithmic layouts outperform the binary tree. The same reasoning applies to multipliers built using Booth 2 encoding for quad precision (113 bits) because the number of partial products is approximately equal in both cases.

The relative performance of the algorithmically generated partial product array to binary trees for quad precision non-Booth encoded multipliers is shown in Fig. 6. The graph shows that the latency of the binary tree is smaller than that for the algorithmically generated arrays for large feature sizes. This is because the quad format has a large number of partial products and therefore requires a larger number of counters in the critical path. At these feature sizes, the contribution to delay due to interconnect is small. As a result of the large number of counters, the critical path

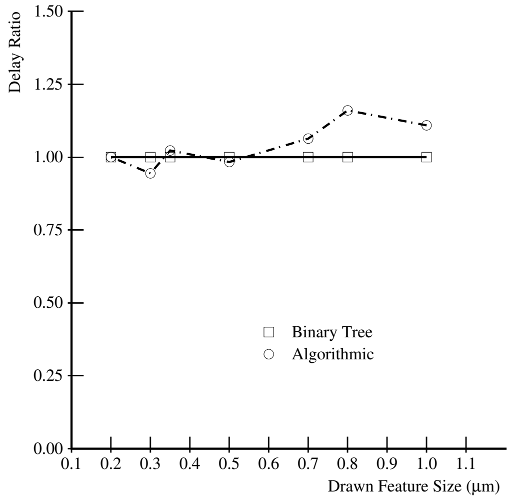

Figure 6. Relative delay: algorithmic layout to binary tree for non-Booth quad precision

Significand Length (bits)	Encoding Scheme		
	Non-Booth	Booth 2	Booth 3
Single (24)	0.85	0.85	0.85
Double (53)	0.88	0.85	0.85
Ext (64)	0.96	0.82	0.88
Ext+4 (68)	0.79	0.77	0.88
Quad (113)	0.95	0.90	0.86

Table 1. Relative delay of Algorithmic Reduction to Binary tree for $0.3\mu m$

contains a large number of buffers. The binary tree is better able to match the delays due to the buffers due to the simplicity of our algorithm for (3,2) array design, yielding overall better performance. However, at smaller feature sizes, interconnect has a significant effect on the total delay. Our simple algorithmic approach is able to provide comparable latency to the binary tree because of its ability to hide interconnect delay by connecting the slow inputs due to the longer wires with the fast inputs for the counters.

Table 1 presents performance results from Fig. 5 and Fig. 6 along with several other common significand precisions and possible encoding schemes for a $0.3\mu m$ process. In this table, the delays are for the algorithmic array relative to those of the binary tree. The results show that an algorithmically-designed array usually results in a lower latency than does the binary tree. Therefore, we recommend further research on the design of more sophisticated tools and algorithms for (3,2) counter-based partial product array generation and interconnection.

Significand Length (bits)	PP Reduction Method			
	Algorithmic		Binary Tree	
	Non-Booth	Booth 3	Non-Booth	Booth 3
Single (24)	1	1.15	0.98	1.12
Double (53)	1.18	1.14	1.14	1.15
Ext (64)	1.25	1.12	1.07	1.04
Ext+4 (68)	1.22	1.16	1.19	1.02
Quad (113)	1.23	1.13	1.18	1.19

Table 2. Relative latency of encoding scheme to Booth 2 for $0.3\mu m$

Significand Length (bits)	PP Reduction Method			
	Algorithmic		Binary Tree	
	Non-Booth	Booth 3	Non-Booth	Booth 3
Single (24)	1.02	1.11	0.99	1.15
Double (53)	1.50	0.99	1.35	1.02
Ext (64)	1.63	0.96	1.31	0.92
Ext+4 (68)	1.60	0.97	1.45	0.90
Quad (113)	1.73	0.95	1.54	1.04

Table 3. Relative latency $\times$ area product of encoding scheme to Booth 2 for $0.3\mu m$

Table 2 summarizes the performance of the different encoding schemes relative to the performance of Booth 2 for a $0.3\mu m$ process. From this table, as the length of the significand increases, Booth 2 becomes the choice which minimizes latency. In most of the cases, the reduction in the number of summands achieved when moving from Booth 2 to Booth 3 encoding is not large enough to offset the extra delay needed to generate the hard (3x) multiple required for Booth 3. We therefore recommend the use of Booth 2 encoding for the generation of partial products when minimum latency is desired.

Not all multiplier implementations require minimum latency. For these cases, an optimized design balances both latency and area. Table 3 summarizes the choice of encoding scheme which minimizes the latency $\times$ area product.

For single precision, both the latency and area of non-Booth and Booth 2 encoding are approximately the same. As a result, the delay $\times$ area product is the same for both. Non-Booth encoding is recommended in this case due to its simplicity of implementation. For other precisions, Booth 3 encoded multipliers are 10-15% smaller and 5-20% slower than Booth 2 encoded multipliers. Accordingly, if area is of primary concern, Booth 3 encoding is recommended for these precisions.

4 Floating Point Division

The emphasis in recent FPUs has been in designing ever-faster adders and multipliers, with division receiving less attention. Current applications and benchmarks are often written assuming that division is an inherently slow operation and should be used sparingly. While division is an infrequent operation even in floating point intensive applications, ignoring its implementation can result in system performance degradation. Thus, a high performance FPU requires a fast and efficient adder, multiplier, and divider. Choosing an optimal FP divider design in terms of performance and area is difficult, as the design space of FP dividers is large, comprising five different classes of division algorithms: digit recurrence, functional iteration, very high radix, table look-up, and variable latency [S7]. This section investigates the performance requirements of FP division and proposes several techniques for achieving them through a combination of FL and VLA techniques.

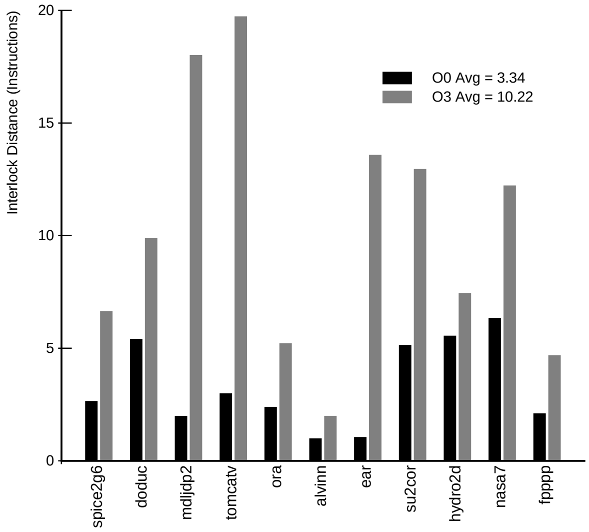

Figure 7. Interlock distances [S2]

4.1 Performance and Area Tradeoffs

We have investigated in detail the relationship between FP division latency and system performance [S2]. System performance was evaluated using 11 applications from the SPECfp92 benchmark suite. The applications were each compiled on a DECstation 5000 using the MIPS C and Fortran compilers at O3 optimization. The results presented were obtained on the MIPS architecture, primarily due to the availability of the flexible program analysis tools pixie and pixstats. The compiler utilized the MIPS R3000 machine model for all schedules assuming double precision FP latencies of 2 cycles for addition, 5 cycles for multiplication, and 19 cycles for division.

In order to analyze the impact that the compiler can have on improving system performance, we measured the interlock distances of division results as a function of compiler optimization level. Fig. 7 shows the average interlock distances for all of the applications at both O0 and O3 levels of optimization. By intelligent scheduling and loop unrolling, the compiler is able to expose instruction-level parallelism in the applications, increasing the interlock distances. Fig. 7 shows that the average interlock distance can be increased by a factor of three by compiler optimization to over 10 instructions. Accordingly, for scalar processors, a division latency of 10 cycles or less can be tolerated.

To determine the effects of division latency on overall system performance, the performance degradation due to division was determined. This degradation is expressed in terms of excess CPI, or the CPI due to the result interlock. The performance degradation due to division latency between 1 and 20 cycles is displayed in Fig. 8. In this figure, designs above 8 cycles are SRT implementations, the design between 4 and 8 cycles is a self-timed SRT design, and those designs below 4 cycles are very-high radix designs requiring large initial approximation tables.

Fig. 8 also shows the effect of increasing the number of instructions issued per cycle on excess CPI due to division. To determine the effect of varying instruction issue rate on excess CPI due to division, a model of an underlying architecture must be assumed. In this study, an optimal superscalar processor is assumed, such that the maximum issue rate is sustainable. The issue rate is then used to appropriately reduce the interlock distances. Fig. 8 also shows how area increases as the functional unit latency decreases. The estimation of area is based on several reported layouts, all of which have been normalized to 1.0μm scalable CMOS layout rules.

Dividers with latencies lower than 4 cycles do not provide significant system performance benefits, and their areas are typically too large to be justified. However, increasing the number of instructions issued per cycle also increases the urgency of division results. Thus, for future microprocessors, it is recommended that the divider have a latency no greater than 10 cycles, and wide issue processors should attempt to reduce latency even further.

4.2 Achieving High Division Performance

Many recent floating point divider implementations have implemented SRT division [S7]. SRT division is a competitive division algorithm mainly for shorter operand lengths, as the algorithm retires only a fixed number of quotient bits in each iteration. For longer floating point formats, such as double extended and quad, the demand for greater than linear convergence becomes apparent. Further, many impor-

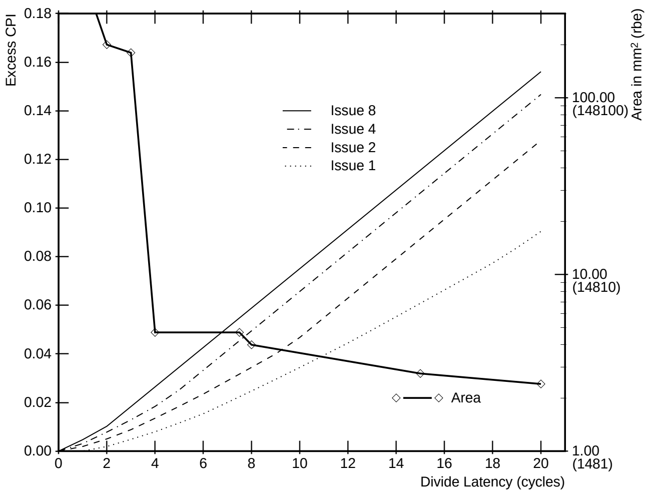

Figure 8. CPI and area vs division latency [S2]

tant special-purpose applications, such as 3D graphics, have a high division throughput requirement. A sequential SRT divider, though, allows only one operation to be in progress at any time. Thus, SRT division may not be appropriate for high throughput, long format implementations. The application of aggressive circuit techniques to low radix stages (radix 2 and radix 4) may allow SRT division to remain competitive for shorter formats, including single and double precision, when high throughput is not required [S8]. In Fig. 8, typical SRT algorithms are represented by small areas and latencies greater than 15 cycles.

Low latency, high throughput division for longer formats can be achieved most readily by using an implementation of division by functional iteration. Such implementations may reuse existing floating point multiplication hardware. Thus, these implementations are attractive from an area-performance perspective. Both the Newton-Raphson and series expansion (Goldschmidt) iterations are effective means of implementing faster division by functional iteration [S7]. The iterations in both algorithms comprise two multiplications and a two's complement operation, which is often replaced by the simpler and faster one's complement. The multiplications in Newton-Raphson are dependent operations, whereas in the series expansion iteration the two multiplications are independent operations and may occur concurrently. A series expansion implementation can therefore take advantage of a pipelined multiplier to obtain higher performance in the form of lower latency per operation. In the Newton-Raphson iteration, unused cycles in a pipelined multiplier can be used to allow for more than one division operation to proceed concurrently, providing higher division throughput.

4.3 Faster Functional Iteration

The number of iterations required for a divider using functional iteration is directly coupled to the accuracy of the initial approximation. Special tables are typically used to obtain a very accurate initial approximation [16], [17]. The challenge in table design is to maximize the approximation accuracy while minimizing its total size. As a result, this continues to be a very important active area of research.

A VLA technique that can be applied to functional iteration is the use of *reciprocal caches*. The use of *result caches* is discussed by Richardson [18]. The use of a small reciprocal cache for an integer divider is discussed in [19]. It has been shown that a reciprocal cache is an efficient technique of reducing average floating point division latency when implementing division by functional iteration [S9]. This technique uses the redundant nature of reciprocal operations present in many applications by trading execution time for increased memory storage. Once a reciprocal is calculated, it is stored in a reciprocal cache. When a division operation is initiated, the reciprocal cache is simultaneously accessed to check for a previously computed result. If a previous reciprocal is available, the result is simply retrieved from the cache and multiplied by the dividend to form the quotient. Otherwise, the operation continues in the divider, and the reciprocal is written into the cache upon completion of the iterations. In the study of [S9], applications from the SPECfp92 and NAS suites are analyzed. It is shown that the use of a reciprocal cache with a total storage of approximately eight-times that of a standard 8-bit initial approximation table, or 16 Kbits, can yield a two-times speedup in average floating point division performance.

The main disadvantage of division by function iteration is the lack of a final remainder, making exact rounding difficult. For exact rounding of the quotient, it is typically necessary to use an additional multiplication of the quotient and the divisor and then to subtract the product from the dividend to form the final remainder. Accordingly, quadratically-converging algorithms can incur a latency penalty of one back-multiplication and a subtraction in order to produce exactly rounded quotients. A VLA technique that can be used to reduce the latency penalty involves keeping several extra guard bits in an appropriately biased pre-rounded result. Schwarz [20] proposes using 1 additional guard bit in the pre-rounded result. In this way, the back-multiplication and subtraction is only required in half of the cases on average. An extension to this technique [S1] further reduces the latency penalty. An exhaustive analysis of the possible cases for various numbers of guard bits and rounding modes demonstrates that by using m bits of extra precision in the pre-rounded result, a back-multiplication and subtraction are required for only 2^{-m} of all cases, reducing the average latency for exactly-

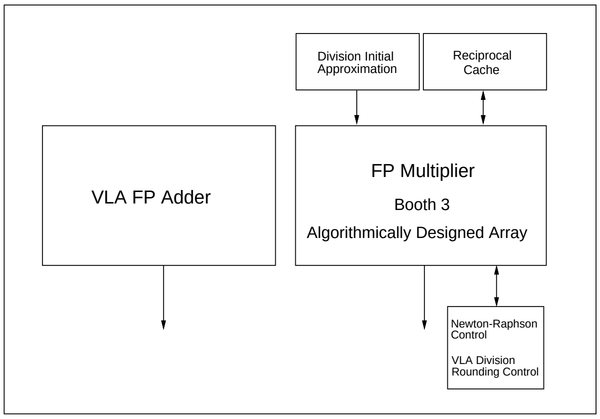

Figure 9. Possible FPU implementation

rounded quotients formed using functional iteration. For RN, the back-multiplication is required for cases very near the halfway point between two representable machine numbers to determine on which side of the halfway point the infinitely precise result lies, while the other cases do not require the multiplication. For RZ, RM, and RP, the back-multiplication is required for cases very near a machine number to determine on which side of the machine number the infinitely precise result lies, while the other cases do not require the multiplication.

The use of small and accurate initial approximation tables, coupled with a reciprocal cache and a few extra guard bits in the pre-rounded result can result in a significant reduction in the latency of floating point division.

5 Summary

Bringing together many of the themes of this paper, one possible organization of a high performance FPU is shown in Fig. 9.

The pipelined FP adder in this FPU uses the VLA addition algorithm of Fig. 2. A pipelined multiplier is shown that uses Booth 3 encoding and an algorithmically-designed array of (3,2) counters. Such a multiplier implementation provides a good balance of performance and area. Division is computed through the Newton-Raphson iteration. This allows for more than one division operation to be in progress simultaneously, providing higher division throughput. Dedicated hardware is included to provide a very accurate initial reciprocal approximation. A small reciprocal cache returns frequently-computed reciprocals at a much lower latency. VLA exact rounding is supported in this implementation by using a multiplier with wider precision than is strictly required in order to compute several extra guard bits in the pre-rounded quotient estimate and thus reduce average latency.

This paper demonstrates that VL functional units provide a means for achieving higher performance than can be obtained through FL-only implementations. The next generation of performance-oriented processors require flexible and robust micro-architectures to best exploit the performance achievable with VL FPUs. As superscalar processors become more complex and move to higher widths of instruction issue, it becomes even more imperative that processors incorporate VL functional units due to the increased exposure of the latencies of individual FP functional units.

Acknowledgments

This work was supported by the NSF under grant MIP93-13701 and a fellowship from the Saudi National Guard.

List of Recent SNAP Publications

The following is a list of recent publications related to the SNAP project. These and other publications and information on the SNAP project and researchers may be obtained through the World Wide Web using the URL http://umunhum.stanford.edu.

[S1] S. F. Oberman, *Design Issues in High Performance Floating Point Arithmetic Units*, Ph.D. thesis, Stanford University, Nov. 1996.

[S2] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *IEEE Trans. Computers*, vol. 46, no. 2, pp. 154–161, Feb. 1997.

[S3] S. F. Oberman and M. J. Flynn, "A variable latency pipelined floating-point adder," in *Proc. Euro-Par'96, Springer LNCS vol. 1124*, pp. 183–192, Aug. 1996.

[S4] N. T. Quach and M. J. Flynn, "An improved algorithm for high-speed floating-point addition," Technical Report No. CSL-TR-90-442, Stanford University, Aug. 1990.

[S5] H. Al-Twaijry and M. J. Flynn, "Optimum placement and routing of multiplier partial product trees," Technical Report: CSL-TR-96-706, Stanford University, Sept. 1996.

[S6] G. McFarland and M. Flynn, "Limits of scaling MOSFETs," Technical Report: CSL-TR-95-662 Revised, Stanford University, Nov. 1995.

[S7] S. F. Oberman and M. J. Flynn, "Division algorithms and implementations," *to appear in IEEE Trans. Computers*, 1997.

[S8] D. L. Harris, S. F. Oberman and M. A. Horowitz, "SRT division architectures and implementations," in *Proc. 13th IEEE Symp. Computer Arithmetic*, this volume, July 1997.

[S9] S. F. Oberman and M. J. Flynn, "Reducing division latency with reciprocal caches," *Reliable Computing*, vol. 2, no. 2, pp. 147–153, Apr. 1996.

References

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

[2] D. Greenley et al., "UltraSPARC: the next generation superscalar 64-bit SPARC," in *Digest of Papers. COMPCON 95*, pp. 442–451, Mar. 1995.

[3] L. Kohn and S. W. Fu, "A 1,000,000 transistor microprocessor," in *Digest of Technical Papers, IEEE Int. Solid-State Circuits Conf.*, pp. 54–55, 1989.

[4] J. A. Kowaleski et al., "A dual-execution pipelined floating-point CMOS processor," in *Slide Supplement to Digest of Technical Papers, IEEE Int. Solid-State Circuits Conf.*, pp. 287, 1996.

[5] M. P. Farmwald, *On the Design of High Performance Digital Arithmetic Units*, Ph.D. thesis, Stanford University, Aug. 1981.

[6] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," in *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 196–205, June 1994.

[7] SPEC Benchmark Suite Release 2/92.

[8] D. W. Sweeney, "An analysis of floating-point addition," *IBM Systems Journal*, vol. 4, pp. 31–42, 1965.

[9] O. L. McSorley, "High speed arithmetic in binary computers," *Proc. IRE*, vol. 49, no. 1, pp. 67–91, Jan. 1961.

[10] C. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electronic Computers*, pp. 14–17, Feb. 1964.

[11] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, Mar. 1965.

[12] D. T. Shen and A. Weinberger, "4-2 carry-save adder implementation using send circuits," *IBM Technical Disclosure Bull.*, vol. 20, no. 9, Feb. 1978.

[13] M. Santoro and M. Horowitz, "A pipelined 64X64b iterative array multiplier," in *Digest of Technical Papers, IEEE Int. Solid-State Circuits Conf.*, pp. 35–36, Feb. 1988.

[14] N. Ohkubo et al., "A 4.4 ns CMOS 54*54-b multiplier using pass-transistor multiplexor," *IEEE J. Solid-State Circuits*, vol. SC-30, no. 3, pp. 251–257, Mar. 1995.

[15] V. G. Oklobdzija, D. Villeger and S. S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *IEEE Trans. Computers*, vol. C-45, no.3, pp. 294–305, Mar. 1996.

[16] D. DasSarma and D. Matula, "Faithful bipartite ROM reciprocal tables," in *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 12–25, July 1995.

[17] M. Ito, N. Takagi, and S. Yajima, "Efficient initial approximation and fast converging methods for division and square root," in *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 2–9, July 1995.

[18] S. E. Richardson, "Exploiting trivial and redundant computation," in *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 220–227, July 1993.

[19] D. Eisig et al., "The design of a 64-bit integer multiplier/divider unit," in *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 171–178, July 1993.

[20] E. Schwarz, "Rounding for quadratically converging algorithms for division and square root," in *Proc. 29th Asilomar Conf. on Signals, Systems, and Computers*, pp. 600–603, Oct. 1995.