Design Issues in Division and Other Floating-Point Operations

Stuart F. Oberman, *Student Member*, *IEEE*, and Michael J. Flynn, *Fellow*, *IEEE*

Abstract—Floating-point division is generally regarded as a low frequency, high latency operation in typical floating-point applications. However, in the worst case, a high latency hardware floating-point divider can contribute an additional 0.50 CPI to a system executing SPECfp92 applications. This paper presents the system performance impact of floating-point division latency for varying instruction issue rates. It also examines the performance implications of shared multiplication hardware, shared square root, on-the-fly rounding and conversion, and fused functional units. Using a system level study as a basis, it is shown how typical floating-point applications can guide the designer in making implementation decisions and trade-offs.

Index Terms—Benchmarks, computer arithmetic, division, floating-point, multiplication, square root, system performance.

1 Introduction

In recent years computer applications have increased in their computational complexity. The industry-wide usage of performance benchmarks, such as SPECmarks, forces processor designers to pay particular attention to implementation of the floating-point unit, or FPU. Special purpose applications, such as high performance graphics rendering systems, have placed further demands on processors. High speed floating-point hardware is a requirement to meet these increasing demands.

Modern applications comprise several floating point operations, among them addition, multiplication, division, and square root. In recent FPUs, emphasis has been placed on designing ever-faster adders and multipliers, with division and square root receiving less attention. The typical range for addition latency is two to four cycles, and the range for multiplication is two to eight cycles. In contrast, the latency for double precision division ranges from six to 61 cycles, and square root is often far larger [13]. Most emphasis has been placed on improving the performance of addition and multiplication. As the performance gap widened between these two operations and division, floating-point algorithms and applications have been slowly rewritten to account for this gap by mitigating the use of division. Thus, current applications and benchmarks are usually written assuming that division is an inherently slow operation and should be used sparingly.

While the methodology for designing efficient high-performance adders and multipliers is well understood, the design of dividers still remains a serious design challenge, often viewed as a "black-art" among system designers. Extensive literature exists describing the theory of division. Subtractive methods, such as nonrestoring SRT division which was independently proposed by and subsequently named for Sweeney, Robertson, and Tocher, are described in detail in [3], [4], [7], [17], [21], [22]. Multiplication-based algorithms such as functional iteration are presented in [1], [9], [11], [23]. Various division and square root implementations have been reported in [1], [2], [6], [24]. However, little emphasis has been placed on studying the effects of FP division on overall system performance.

This study investigates in detail the relationship between FP functional unit latencies and system performance. The application suites considered for this study included the NAS Parallel Benchmarks [15], the Perfect Benchmarks [5], and the SPECfp92 [20] benchmark suite. An initial analysis of the instruction distribution determined that the SPEC benchmarks had the highest frequency of floating-point operations, and they were therefore chosen as the target workload of the study to best reflect the behavior of floating-point intensive applications.

These applications are used to investigate several questions regarding the implementation of floating-point units:

- Does a high-latency division/square root operation cause enough system performance degradation to justify dedicated hardware support?
- How well can a compiler schedule code in order to maximize the distance between floating-point result production and consumption?
- What are the effects of increasing the width of instruction issue on effective division latency?
- If hardware support for division and square root unit is warranted and a multiplication-based algorithm is utilized, should the FP multiplier hardware be shared, or should a dedicated functional unit be designed?
- Should square root share the division hardware?
- What operations most frequently consume division results?
- Is on-the-fly rounding and conversion necessary?

The organization of this paper is as follows. Section 2 describes the method of obtaining data from the applications. Section 3 presents and analyzes the results of the study. Section 4 is the conclusion.

- *The authors are with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305. E-mail: oberman@umunhum.stanford.edu.*

Manuscript received July 1, 1995; revised May 16, 1996.
For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C96326.

2 System Level Study

2.1 Instrumentation

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 each of three levels of optimization: no optimization, O2 optimization, and O3 optimization. O2 performs common subexpression elimination, code motion, strength reduction, code scheduling, and inlining of arithmetic statement functions. O3 performs all of O2's optimizations, but it also implements loop unrolling and other code-size increasing optimizations [18]. Among other things, varying the level of compiler optimization varies the total number of executed instructions and the distance between a division operation and the use of its result. The compilers utilized the MIPS R3000 machine model for all schedules assuming double precision FP latencies of two cycles for addition, five cycles for multiplication, and 19 cycles for division.

In most traditional computer architectures, a close match exists between high-level-language semantics and machine-level instructions for floating-point operations [12]. Thus, the results obtained on a given architecture are applicable to a wide range of architectures. The results presented were obtained on the MIPS architecture, primarily due to the availability of the flexible program analysis tools pixie and pixstats [19]. Pixie reads an executable file and partitions the program into its basic blocks. It then writes a new version of the executable containing extra instructions to dynamically count the number of times each basic block is executed. The benchmarks use the standard input data sets, and each executes approximately 3 billion instructions. Pixstats is then used to extract performance statistics from the instrumented applications.

2.2 Method of Analysis

To determine the effects of a floating-point operation on overall system performance, the performance degradation due to the operation needs to be determined. This degradation can be expressed in terms of excess CPI, or the CPI due to the result interlock. Excess CPI is a function of the dynamic frequency of the operation, the urgency of its results, and the functional unit latency. The dynamic frequency of an operation is the number of times that a particular operation is executed in the application. The urgency of a result is measured by how soon a subsequent instruction needs to consume the result. To quantify the urgency of results, interlock distances were measured for division results. The interlock distance is the distance between the production of a division result and its consumption by a subsequent instruction. It is clear that the dynamic frequency is solely a function of the application, urgency is a function of the application and the compiler, and functional unit latency depends upon the hardware implementation. The system designer has the most control over the functional unit latency. Through careful design of the processor architecture, though, the designer has some limited influence on the urgency. Adding extra registers and providing for out-of-order instruction execution are two means by which the system designer can influence urgency.

3 Results

3.1 Instruction Mix

Fig. 1 shows the average frequency of division and square root operations in the benchmark suite relative to the total number of floating-point operations, where the applications have been compiled using O3 optimization. This figure shows that simply in terms of dynamic frequency, division and square root seem to be relatively unimportant instructions, with about 3% of the dynamic floating-point instruction count due to division and only 0.33% due to square root. The most common instructions are FP multiply and add. It should be noted that add, subtract, move, and convert operations typically use the FP adder hardware. Thus, FP multiply accounts for 37% of the instructions, and the FP adder is used for 55% of the instructions. However, in terms of latency, division can play a much larger role. By assuming a machine model of a scalar processor, where every division operation has a latency of 20 cycles and the adder and multiplier each have a three cycle latency, a distribution of the stall time due to the FP hardware was formed, shown in Fig. 2. Here, FP division accounts for 40% of the latency, FP add accounts for 42%, and multiply accounts for the remaining 18%. It is apparent that the performance of division is significant to the overall system performance.

3.2 Compiler Effects

In order to analyze the impact that the compiler can have on improving system performance, the urgency of division results was measured as a function of compiler optimization level. Fig. 3 shows a histogram of the interlock distances for division instructions at O0, as well as a graph of the cumulative interlock distance for the spice benchmark. Fig. 4 shows the same data when compiled at O3. Fig. 5 shows the average interlock distances for all of the applications at both O0 and O3 levels of optimization. It is clear that by intelligent scheduling and loop unrolling, the compiler is able to expose instruction-level parallelism in the applications, reducing the urgency of division results. Fig. 5 shows that the average interlock distance can be increased by a factor of three by compiler optimization.

An average of the division interlock distances from all of the benchmarks was formed, weighted by division frequency in each benchmark. This result is shown in Fig. 6 for the three levels of compiler optimization. In this graph, the curves represent the cumulative percentage of division instructions at each distance. The results from Fig. 5 and Fig. 6 show that the average interlock distance can be increased to only approximately 10 instructions. Even if the compiler assumed a larger latency, there is little parallelism left to exploit that could further increase the interlock distance and therefore reduce excess CPI. If the compiler scheduled assuming a low latency, the excess CPI could only increase for dividers with higher latencies than that for which the compiler scheduled. This is because the data shows the maximum parallelism available when scheduling for a latency of 19 cycles. If the compiler scheduled for a latency much less than 19 cycles, then it would not be as aggressive in its scheduling, and the interlock distances would be smaller, increasing urgency and therefore excess CPI.

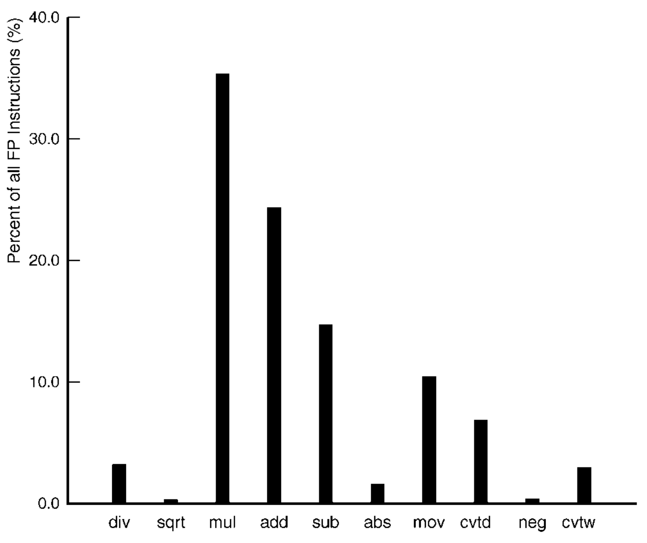

Fig. 1. Distribution of floating point instructions.

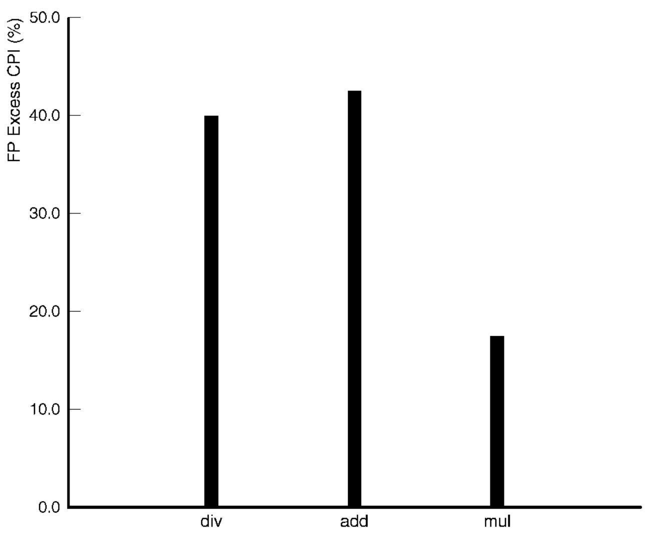

Fig. 2. Distribution of functional unit stall time.

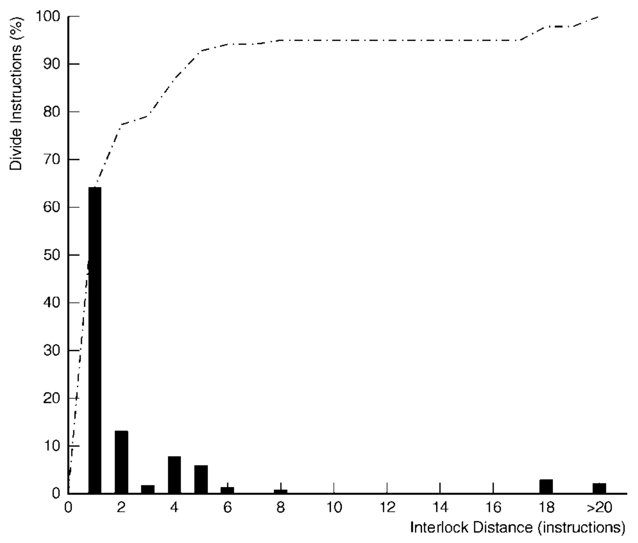

Fig. 3. Spice with optimization O0.

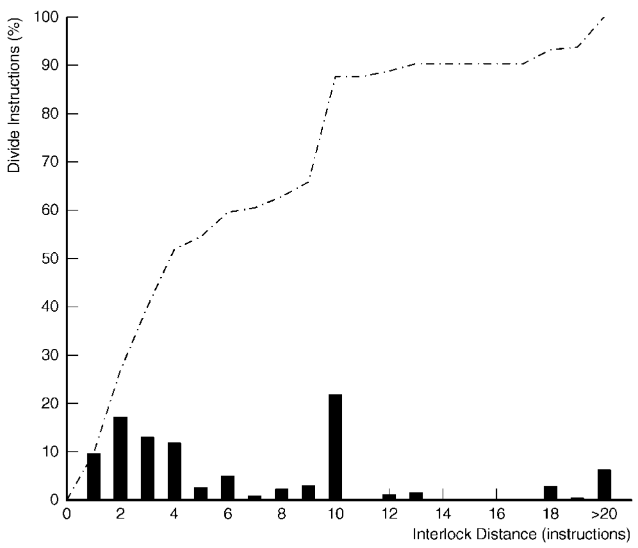

Fig. 4. Spice with optimization O3.

The results for division can be compared with those of addition and multiplication, shown in Fig. 7 and Fig. 8.

3.3 Performance and Area Tradeoffs

The excess CPI due to division is determined by summing all of the stalls due to division interlocks, which is the total penalty, and dividing this quantity by the total number of instructions executed. The performance degradation due to division latency is displayed in Fig. 9. This graph shows how the excess CPI due to the division interlocks varies with division unit latency between one and 20 cycles for O3 optimization. Varying the optimization level also changed the total number of instructions executed, but left the number of division instructions executed constant. As a result, the fraction of division instructions is also a function of optimization level. While CPI due to division actually increases from O0 to O2, the overall performance at O2 and O3 increases because the total instruction count decreases. This effect is summarized in Table 1, where the division latency is taken to be 20 cycles.

TABLE 1
EFFECTS OF COMPILER OPTIMIZATION

Optimization Level	Division Frequency	Excess CPI
O0	0.33%	0.057
O2	0.76%	0.093
O3	0.79%	0.091

Fig. 9 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. This model simplifies the analysis while providing an upper bound on the performance deg-

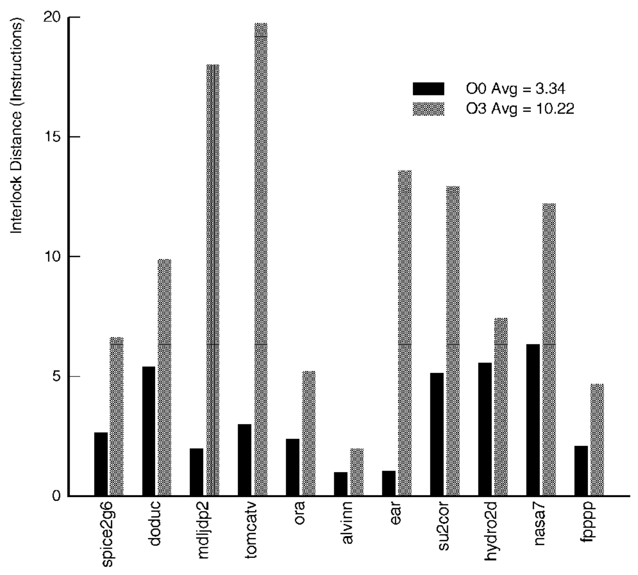

Fig. 5. Interlock distances: by application.

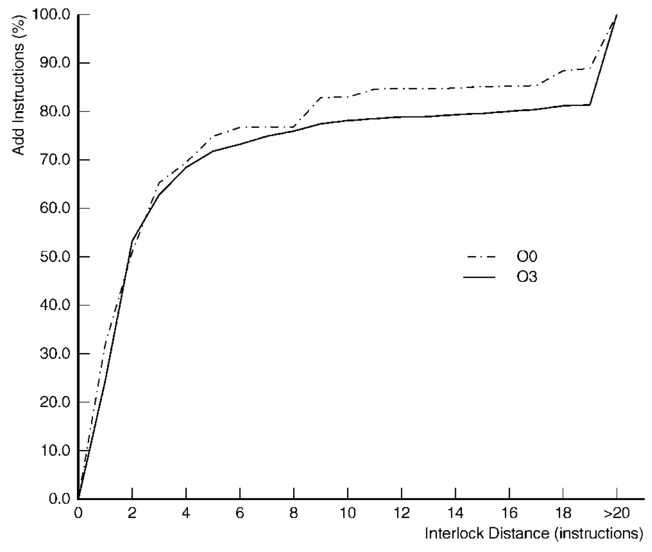

Fig. 7. Cumulative average addition interlock distances.

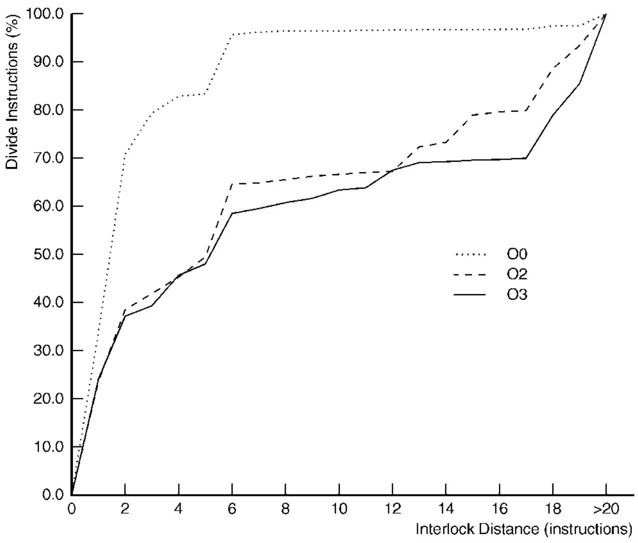

Fig. 6. Interlock distances: cumulative average.

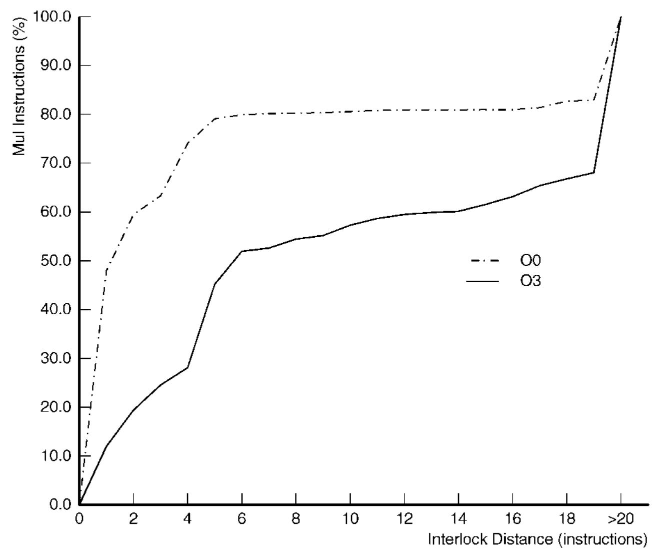

Fig. 8. Cumulative average multiplication interlock distances.

radation due to division. The issue rate is used to appropriately reduce the interlock distances. As the width of instruction issue increases, urgency of division data increases proportionally. In the worst case, every division result consumer could cause a stall equal to the functional unit latency. The excess CPI for the multiple issue processors is then calculated using the new interlock distances.

Fig. 9 also shows how area increases as the functional unit latency decreases. The estimation of area is based on reported layouts from [16], [24], [25], all of which have been normalized to 1.0 μm scalable CMOS layout rules. As division latencies decrease below four cycles, a large tradeoff must be made. Either a very large area penalty must be incurred to achieve this latency by utilizing a very large look-up table method, or large cycle times may result if an SRT method is utilized.

In order to make the comparison of chip areas technology independent, the register bit equivalent (**rbe**) model of Mulder [14] was used. In this model, one **rbe** equals the area of a one bit storage cell. For the purposes of this study, an **rbe** unit is referenced to a six-transistor static cell with high bandwidth, with an area of $675f^2$, where f is the minimum feature size. The area required for 1 static RAM bit, as would be used used in an on-chip cache, is about 0.6 **rbe**. Since all areas are normalized to an f = 1.0 μm process, 1 mm^2 = 1481 ***rbe***.

Fig. 10 shows excess CPI versus division latency over a larger range of latencies. This graph can roughly be divided into five regions. Table 2 shows that inexpensive 1-bit SRT schemes use little area but can contribute in the worst case up to 0.50 CPI in wide-issue machines. Increasing the radix of SRT implementations involves an increase in area, but

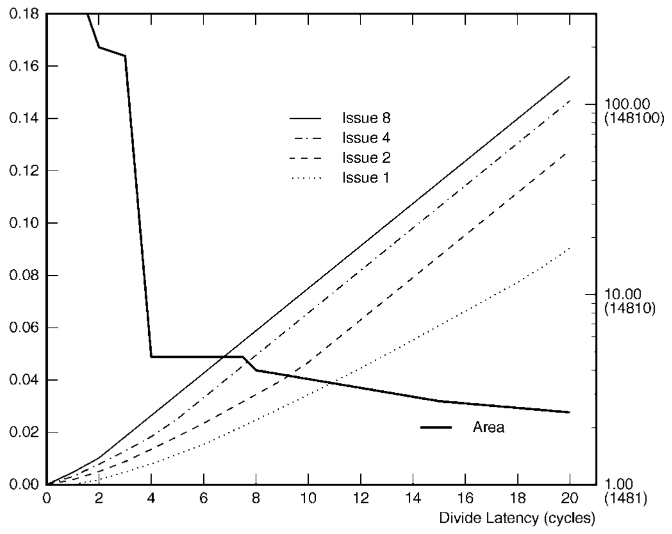

Fig. 9. CPI and area vs. division latency—low latency.

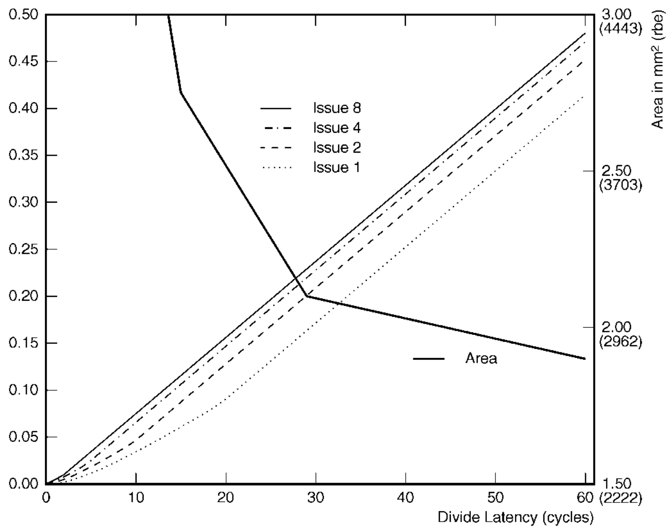

Fig. 10. CPI and area vs. division latency—full range.

TABLE 2
FIVE REGIONS OF DIVISION LATENCY

Divider Type	Latency (cycles)	Excess CPI	Area (rbe)
1-Bit SRT	> 40	< 0.5	< 3000
2-Bit SRT	[20, 40]	[0.10, 0.32]	3110
4-Bit SRT	[10, 20]	[0.04, 0.10]	4070
8-Bit SRT and Self-Timed	[4, 10]	[0.01, 0.07]	6665
Very-High Radix	< 4	< 0.01	> 100,000

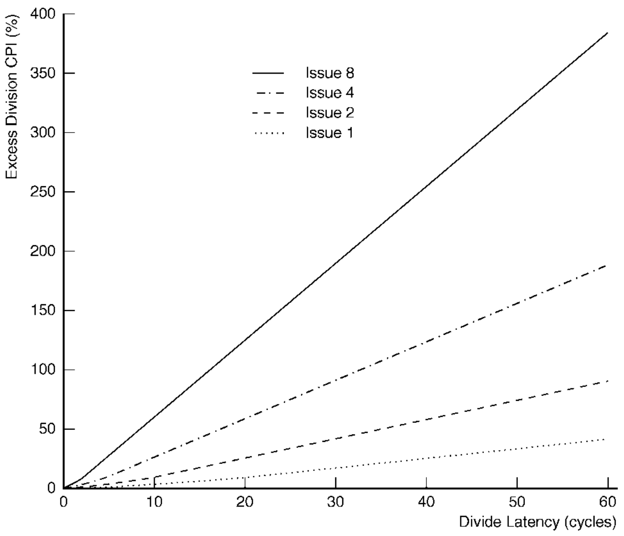

Fig. 11. Excess CPI as a percentage of base CPI for multiple issue processors.

with a decrease in excess CPI. The region corresponding to 4-bit SRT schemes also represents the performance of typical multiplication-based division implementations, such as Newton-Raphson or series expansion [6]. The additional area required in such implementations is difficult to quantify, as implementations to date have shared existing multipliers, adding control hardware to allow for the shared functionality. At a minimum, such implementations require a starting approximation table that provides at least an 8 bit initial approximation. Such a table occupies a minimum of 2 Kbits, or 1230 **rbe**. The final region consists of very-high radix dividers of the form presented in [8] and [25]. To achieve this performance with CPI < 0.01, large area is required for very large look-up tables, often over 500,000 **rbe**.

To better understand the effects of division latency on the system performance of multiple issue processors, the excess CPI due to division can be expressed as a percentage of the base processor CPI. Fig. 11 shows this relationship quantitatively. As instruction width increases, the degradation of system performance markedly increases. Not only does increasing the width of instruction issue reduce the average interlock distance, but the penalty for a division interlock relative to the processor issue rate dramatically increases. A slow divider in a wide-issue processor can easily reduce system performance by half.

An expansion of this study examines the performance/area tradeoffs between additional FP functional units and other CPU components such as larger caches and branch target buffers. This represents ongoing research by colleagues at Stanford [10]. Interested readers may consult our world-wide-web server[1] for recent technical reports.

3.4 Shared Multiplier Effects

If a multiplication-based division algorithm is chosen, such as Newton-Raphson or series expansion, it must be decided whether to use a dedicated multiplier or to share the existing multiplier hardware. The area of a well-designed three cycle FP multiplier is around 11 mm^2, again using the 1.0μm process. Adding this much area may not be always desirable. If an existing multiplier is shared, this will have two effects. First, the latency through the multiplier will likely increase due to the modifications necessary to sup-

1. http://umunhum.stanford.edu

port the division operation. Second, multiply operations may be stalled due to conflicts with division operations sharing the multiplier.

The effect of this structural hazard on excess CPI is shown in Fig. 12. The results are based on an average of all of the applications when scheduled with O3. In all cases for a division latency less than 20 cycles, the excess CPI is less than 0.07. For reasonable implementations of multiplication-based division, with a latency of approximately 13 cycles, the actual penalty is $0.02 < CPI < 0.04$. For these applications, due to the relatively low frequency of division operations, the penalty incurred for sharing an existing multiplier is not large. For special classes of applications, such as certain graphics applications, division and multiplication frequencies could be higher, requiring a separate division unit to achieve high performance.

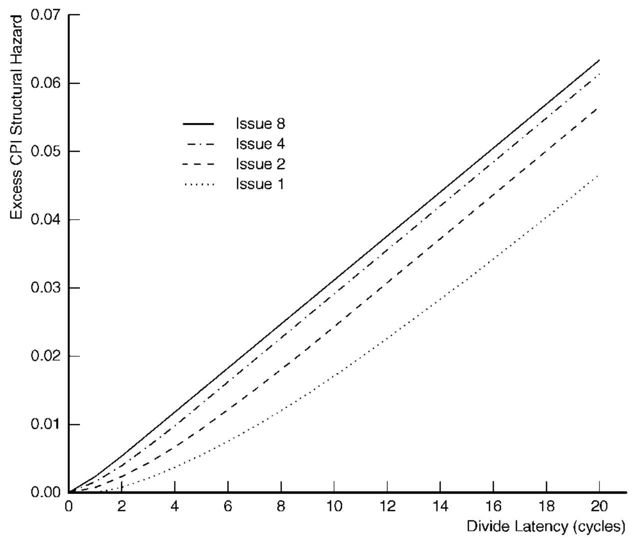

Fig. 12. Excess CPI due to shared multiplier.

3.5 Shared Square Root

The recurrence equation for square root is very close in form to that of division for both subtractive and multiplicative algorithms. Accordingly, division hardware can be implemented with additional functionality to perform square root computation. The design tradeoff then becomes whether a possible increase in hardware complexity and/or cycle time can be justified by an increase in overall performance.

The results of this study show that floating point square root on the average accounts for 0.087% of all executed instructions. This is a factor of 9.1 less than division. To avoid significant performance degradation due to square root, the latency of square root should be no worse than a factor 9.1 greater than division. However, any hardware implementation of square root will more than likely meet the requirement of this bound. Even a simple 1 bit per iteration square root would contribute only 0.05 CPI for a scalar processor. Accordingly, these results suggest that the square root implementation does not need to have the same radix as the divider, and the sharing of division hardware is not crucial to achieving high system performance. Only if the additional area is small and the cycle time impact is negligible should division and square root share the same hardware.

3.6 On-the-Fly Rounding and Conversion

In a nonrestoring division implementation such as SRT, an extra cycle is often required after the division operation completes. In SRT, the quotient is typically collected in a representation where the digits can take on both positive and negative values. Thus, at some point, all of the values must be combined and converted into a standard representation. This requires a full-width addition, which can be a slow operation. To conform to the IEEE standard, it is necessary to round the result. This, too, can require a slow addition.

Techniques exist for performing this rounding and conversion "on-the-fly," and therefore the extra cycle may not be needed [7]. Because of the complexity of this scheme, the designer may not wish to add the additional required hardware. Fig. 13 shows the performance impact of requiring an additional cycle after the division operation completes. For division latencies greater than 10 cycles, less than 20% of the total division penalty in CPI is due to the extra cycle. At very low division latencies, where the latency is less than or equal to four cycles, the penalty for requiring the additional cycle is obviously much larger, often greater than 50% of the total penalty.

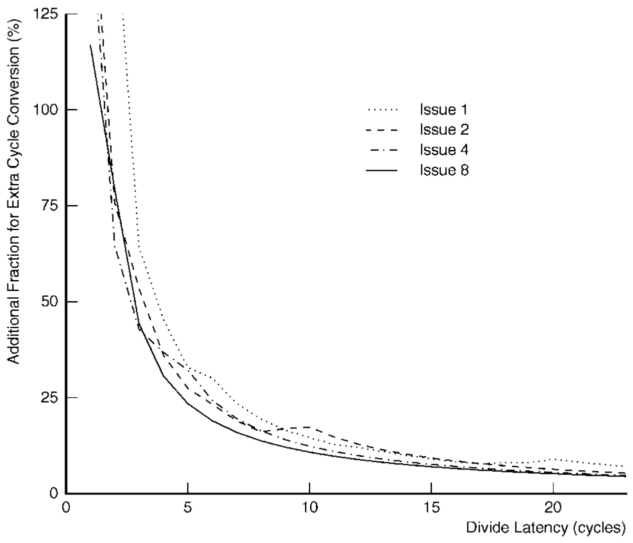

Fig. 13. Effects of on-the-fly rounding and conversion.

3.7 Consumers of Division Results

In order to reduce the effective penalty due to division, it is useful to consider which operations actually use division results. Fig. 14 is a histogram of instructions that consume division results. This can be compared with the histogram for multiply results, shown in Fig. 15. For multiply results, the biggest users are multiply and add instructions. It should be noted that both *add.d* and *sub.d* use the FP adder. Thus, the FP adder is the consumer for nearly 50% of the multiply results. Accordingly, fused operations such as

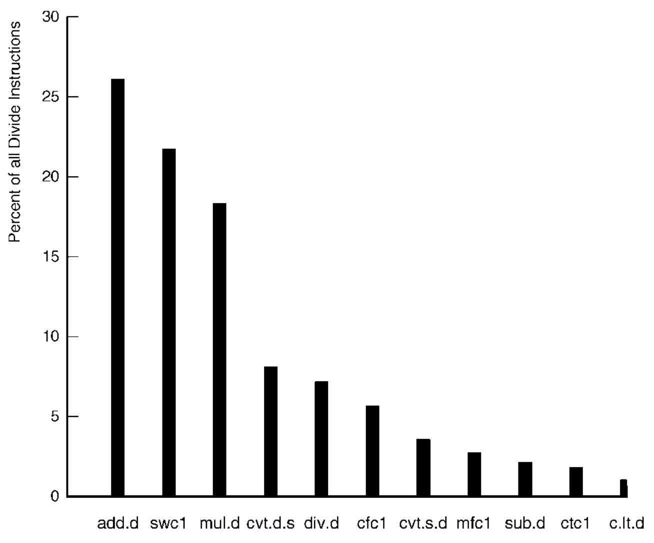

Fig. 14. Consumers of division results.

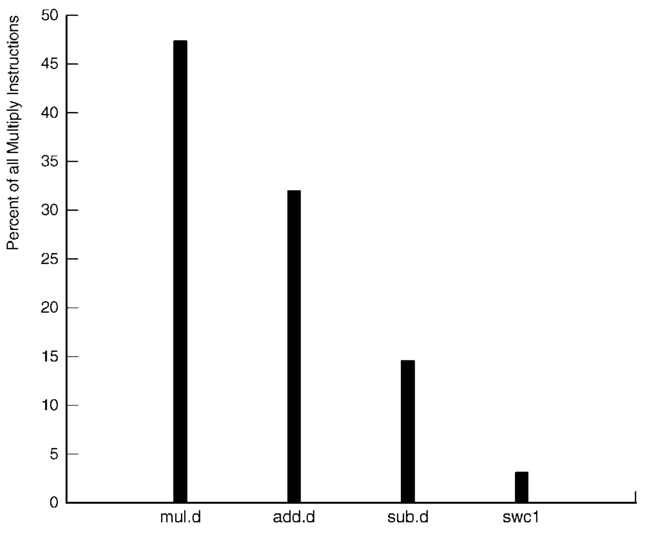

Fig. 15. Consumers of multiply results.

multiply-accumulate are useful. Because the multiply-add pattern occurs frequently in such applications and it does not require much more hardware than the separate functional units, fused multiply-adders are often used in modern processors.

Looking at the consumers of division results, the FP adder is the largest consumer with 27% of the results. The second biggest consumer is the store operation with 23% of the results. It is possible to overcome the penalties due to a division-store interlock, though, with other architectural implementations. A typical reason why a store would require a division result and cause an interlock is because of register pressure, due to a limited number of registers. By either adding registers or register renaming, it may be possible to reduce the urgency due to store.

While the percentage of division results that the adder consumes is not as high as for multiply results, it is still the largest quantity. A designer could consider the implementation of a fused divide-add instruction to increase performance. In division implementations where on-the-fly conversion and rounding is not used, an extra addition cycle exists for this purpose. It may be possible to make this a three-way addition, with the third operand coming from a subsequent add instruction. Because this operand is known soon after the instruction is decoded, it can be sent to the the three-way adder immediately. Thus, a fused divide-add unit could provide additional performance.

4 Conclusion

This study has investigated the issues of designing an FP divider in the context of an entire system. The frequency and interlock distance of division instructions in SPECfp92 benchmarks have been determined, along with other useful measurements, in order to answer several questions regarding the implementation of a floating-point divider.

The data shows that for the slowest hardware divider, with a latency greater than 60 cycles, the CPI penalty can reach 0.50. To achieve good system performance, some form of hardware division is required. However, at very low divider latencies, two problems arise. The area required increases exponentially or cycle time becomes impractical. This study shows a knee in the area/performance curve near 10 cycles. Dividers with lower latencies do not provide significant system performance benefits, and their areas are too large to be justified.

The results show the compiler's ability to decrease the urgency of division results. Most of the performance gain is in performing basic compiler optimizations, at the level of O2. Only marginal improvement is gained by further optimization. The average interlock distance increases by a factor of three by using compiler optimization. Accordingly, for scalar processors, this study shows that a division latency of 10 cycles or less can be tolerated.

Increasing the number of instructions issued per cycle also increases the urgency of division results. On the average, increasing the number of instructions issued per cycle to two causes a 38% increase in excess CPI, increasing to four causes a 94% increase in excess CPI, and increasing to eight causes a 120% increase in excess CPI. Further, as the width of instruction issue increases, the excess CPI due to division expressed as a percentage of base CPI increases even faster. Wide issue machines utilize the instruction-level parallelism in applications by issuing multiple instructions every cycle. While this has the effect of decreasing the base CPI of the processor, it exposes the functional unit latencies to a greater degree and accentuates the effects of slow functional units.

In most situations, an existing FP multiplier can be shared when using a multiplication-based division algorithm. The results show that for a division latency of around 13 cycles, the CPI penalty is between 0.025 and 0.040. Thus, due to the low frequency of division operations combined with the low frequency of multiply instructions that occur in-between the division result production and consumption, the structural hazard is also very infrequent. While the CPI penalty is low when the multiplier is shared and modified to also perform division, the designer must also consider latency effects through the multiplier which could have an impact on cycle time.

On-the-fly rounding and conversion is not essential for all division implementations. For division latencies greater than 10 cycles, the lack of on-the-fly rounding and conversion does not account for a significant fraction of the excess CPI, and, as a result, is not necessarily required. However, for very high radix implementations where the area and complexity are already large, this method is a practical means of further reducing division latency.

Addition and store operations are the most common consumers of division results. Accordingly, the design of a fused divide-add unit is one means of achieving additional system performance.

While division is typically an infrequent operation even in floating-point intensive applications, ignoring its implementation can result in system performance degradation. By studying several design issues related to FP division, this paper has attempted to clarify the important tradeoffs in implementing an FP divider in hardware.

ACKNOWLEDGMENTS

The authors would like to thank Nhon Quach for his valuable discussions and comments. The authors would also like to thank Grant McFarland for reading and commenting on an earlier draft of this paper.

This research was supported by the National Science Foundation under grant MIP93-13701.

REFERENCES

[1] S.F. Anderson, J.G. Earle, R.E. Goldschmidt, and D.M. Powers, “The IBM System/360 Model 91: Floating-Point Execution Unit,” *IBM J. Research and Development*, vol. 11, pp. 34-52, Jan. 1967.

[2] T. Asprey, G. Averill, E. DeLano, R. Mason, B. Weiner, and J. Yetter, “Performance Features of the PA7100 Microprocessor,” *IEEE Micro*, vol. 13, no. 3, pp. 22-35, June 1993.

[3] D.E. Atkins, “Higher-Radix Division Using Estimates of the Divisor and Partial Remainders,” *IEEE Trans. Computers*, vol. 17, no. 10, pp. 925-934, Oct. 1968.

[4] N. Burgess and T. Williams, “Choices of Operand Truncation in the SRT Division Algorithm,” *IEEE Trans. Computers*, vol. 44, no. 7, pp. 933-937, July 1995.

[5] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, “Supercomputer Performance Evaluation and the Perfect Benchmarks,” *Proc. Int’l Conf. Supercomputing*, pp. 254-266, June 1990.

[6] M. Darley, B. Kronlage, D. Bural, B. Churchill, D. Pulling, P. Wang, R. Iwamoto, and L. Yang, “The TMS390C602A Floating-Point Coprocessor for Sparc Systems,” *IEEE Micro*, vol. 10, no. 3, pp. 36-47, June 1990.

[7] M.D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, 1994.

[8] M.D. Ercegovac, T. Lang, and P. Montuschi, “Very High Radix Division with Selection by Rounding and Prescaling,” *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 112-199, July 1993.

[9] M. Flynn, “On Division by Functional Iteration,” *IEEE Trans. Computers*, vol. 19, no. 8, pp. 702-706, Aug. 1970.

[10] S. Fu, N. Quach, and M. Flynn, “Architecture Evaluator’s Work Bench and Its Application to Microprocessor Floating Point Units,” Technical Report no. CSL-TR-95-668, Computer Systems Laboratory, Stanford Univ., June 1995.

[11] R.E. Goldschmidt, “Applications of Division by Convergence,” MS thesis, Dept. of Electrical Eng., Massachusetts Inst. of Technology, June 1964.

[12] J.C. Huck and M.J. Flynn, *Analyzing Computer Architectures*. Washington, D.C.: IEEE CS Press, 1989.

[13] *Microprocessor Report*, various issues, 1994-1996.

[14] J.M. Mulder, N.T. Quach, and M. Flynn, “An Area Model for On-Chip Memories and Its Application,” *IEEE J. Solid-State Circuits*, vol. 26, no. 2, pp. 98-105, Feb. 1991.

[15] NAS Parallel Benchmarks 8/91.

[16] S. Oberman, N. Quach, and M. Flynn, “The Design and Implementation of a High-Performance Floating-Point Divider,” Technical Report no. CSL-TR-94-599, Computer Systems Laboratory, Stanford Univ., Jan. 1994.

[17] S.F. Oberman and M.J. Flynn, “Measuring the Complexity of SRT Tables,” Technical Report no. CSL-TR-95-679, Computer Systems Laboratory, Stanford Univ., Nov. 1995.

[18] DEC Fortran Language Reference Manual, 1992.

[19] M.D. Smith, “Tracing with Pixie,” Technical Report no. CSL-TR-91-497, Computer Systems Laboratory, Stanford Univ., Nov. 1991.

[20] SPEC Benchmark Suite Release 2/92.

[21] K.G. Tan, “The Theory and Implementation of High-Radix Division,” *Proc. Fourth IEEE Symp. Computer Arithmetic*, pp. 154-163, June 1978.

[22] G.S. Taylor, “Radix 16 SRT Dividers with Overlapped Quotient Selection Stages,” *Proc. Seventh IEEE Symp. Computer Arithmetic*, pp. 64-71, June 1985.

[23] S. Waser and M. Flynn, *Introduction to Arithmetic for Digital Systems Designers*. New York: Holt, Rinehart, and Winston, 1982.

[24] T.E. Williams and M.A. Horowitz, “A Zero-Overhead Self-Timed 160-ns 54-b CMOS Divider,” *IEEE J. Solid-State Circuits*, vol. 26, no. 11, pp. 1,651-1,661, Nov. 1991.

[25] D. Wong and M. Flynn, “Fast Division Using Accurate Quotient Approximations to Reduce the Number of Iterations,” *IEEE Trans. Computers*, vol. 41, no. 8, pp. 981-995, Aug. 1992.

Stuart F. Oberman received the BS degree in electrical engineering from the University of Iowa, Iowa City, in 1992. He received the MS and PhD degrees in electrical engineering from Stanford University, Stanford, California, in 1994 and 1997, respectively.

From 1993-1996, he participated in the design of several commercial floating-point units. He is currently a consultant for the California Microporcessor Division of Advanced Micro Devices, Milpitas, California. His current research interests include computer arithmetic, computer architecture, and VLSI design.

Dr. Oberman is a Tau Beta Pi Fellowship recipient and a member of Tau Beta Pi, Eta Kappa Nu, Sigma Xi, ACM, and the IEEE Computer Society.

Michael J. Flynn is a professor of electrical engineering at Stanford University. His experience includes 10 years at IBM Corporation working in computer organization and design. He was also a faculty member at Northwestern University and Johns Hopkins University, and the director of Stanford’s Computer Systems Laboratory from 1977 to 1983.

Dr. Flynn has served as vice president of the Computer Society and was founding chairman of CS’s Technical Committee on Computer Architecture, as well as ACM’s Special Interest Group on Computer Architecture. He has served two terms on the IEEE Board of Governors. He was the 1992 recipient of the ACM/IEEE Eckert–Mauchly Award for his contributions to processor classification and computer arithmetic. He was the 1995 recipient of the IEEE-CS Harry Goode Memorial Award in recognition of his outstanding contribution to the design and classification of computer architecture. He is the author of three books and more than 200 technical papers.

Dr. Flynn is a fellow of the IEEE and the ACM.