DESIGN ISSUES IN FLOATING-POINT DIVISION

Stuart F. Oberman and Michael J. Flynn

Technical Report: CSL-TR-94-647

December 1994

This work was supported by NSF under contract MIP93-13701.

DESIGN ISSUES IN
FLOATING-POINT DIVISION

by

Stuart F. Oberman and Michael J. Flynn

Technical Report: CSL-TR-94-647

December 1994

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Floating-point division is generally regarded as a low frequency, high latency operation in typical floating-point applications. However, the increasing emphasis on high performance graphics and the industry-wide usage of performance benchmarks, such as SPECmarks, forces processor designers to pay close attention to all aspects of floating-point computation. This paper presents the algorithms often utilized for floating-point division, and it also presents implementation alternatives available for designers. 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.

Key Words and Phrases: Floating-point, division, benchmarks, system performance

Contents

1 Introduction **1**

2 Divide Algorithms: General Discussion **2**
2.1 Subtractive Algorithms 2
2.2 Multiplicative Algorithms 3
2.3 Comparison 5

3 System Level Study **5**
3.1 Instrumentation 5
3.2 Method of Analysis 6

4 Results **6**
4.1 Instruction Mix 6
4.2 Compiler Effects 6
4.3 Overall CPI 9
4.4 Shared Multiplier Effects 10
4.5 On-the-fly Rounding and Conversion 12
4.6 Consumers of Divide Results 13

5 Conclusion **15**

6 Acknowledgement **15**

List of Figures

1 Basic SRT Topology . . . 4
2 Instruction count distribution . . . 7
3 Functional unit stall time distribution . . . 7
4 Spice with optimization O0 . . . 8
5 Spice with optimization O3 . . . 8
6 Cumulative average interlock distance . . . 9
7 Average effective divide latency . . . 9
8 CPI and area vs divide latency . . . 11
9 CPI and area vs divide latency . . . 11
10 Excess CPI due to shared multiplier . . . 12
11 Effects of on-the-fly rounding and conversion . . . 13
12 Consumers of divide results . . . 14
13 Consumers of multiply results . . . 14

List of Tables

1 Effects of compiler optimization 10

1 Introduction

Modern computer applications have increased in their computation complexity in recent years. The development of high speed floating-point (FP) arithmetic is a requirement to meet the computation demands of modern applications which have increased in their computation complexity in recent years. The emphasis on high performance graphics rendering systems has placed further demands on the computation abilities of processors. Furthermore, the industry-wide usage of performance benchmarks, such as SPECmarks, forces processor designers to pay particular attention to floating-point computation.

Applications such as the aforementioned comprise several floating point operations, among them addition, multiplication, and division. In recent FPUs, emphasis has been placed on designing ever-faster adders and multipliers, with division receiving less attention. Typically, the range for addition latency is 2 to 4 cycles, and the range for multiplication is 2 to 8 cycles. In contrast, the latency for double precision division in modern FPUs ranges from 7 to 61 cycles [4]. This phenomenon is largely due to the perception that divide is an infrequent operation in modern floating-point applications. Because of the low frequency, it is believed that the overall performance degradation incurred by the use of a slow divider will not be large. More 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 divide. Thus, current applications and benchmarks are usually written assuming that divide is an inherently slow operation and should therefore 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 theory exists describing the theory of division. However, the implementation of division has received less attention, and very little emphasis has been placed on studying the effects of FP division on overall system performance.

This study investigates in more detail the relationship between FP divide and system performance. This relationship is studied in the context of a set of floating-point applications. The choice of applications to use when studying the performance of a system is often difficult and controversial. The application suites considered for this study included the NAS Parallel Benchmarks [5], the Perfect Benchmarks [8], and the SPECfp92 [10] benchmark suite. An initial analysis of the instruction distribution showed 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 division:

- Does a high-latency divide operation cause enough system performance degradation to warrant a separate, lower latency divide functional unit?
- How well can a compiler schedule code in order to maximize the distance between

divide result production and consumption?

- What are the effects of increasing the width of instruction issue on effective divide latency?
- If a hardware divide unit is warranted, should the divider share the FP multiplier hardware, or should it have its own dedicated functional unit?
- Is on-the-fly rounding and conversion necessary?

The remainder of this paper is organized as follows. Section 2 presents common division algorithms and implementations. Section 3 describes the method of obtaining data from the benchmarks. Section 4 presents and analyzes the results of the study. Section 5 is the conclusion.

2 Divide Algorithms: General Discussion

Many classes of algorithms exist for implementing division. These include the subtractive method, the multiplicative method, various approximation methods, and special methods such as the CORDIC and continued product methods [1]. The most commonly used algorithms in modern FPUs are the subtractive and multiplicative methods, and the analysis here is limited to these.

2.1 Subtractive Algorithms

Digit recurrence algorithms use subtraction as the iterative operator. The quotient is represented in a radix-r form and one digit of it is calculated in every iteration. This class be can be further divided into *restoring* and *nonrestoring* division. Restoring division is similar to the familiar paper and pencil division. When dividing two n-bit numbers, the division can require up to 2n + 1 adds. The Winograd bound [13] on restoring division in gate delays is therefore:

$$T = (2n + 1) \log_2 2n$$

Nonrestoring division eliminates the restoration cycles. Accordingly, the bound on nonrestoring division is given by:

$$T = n \log_2 2n$$

SRT is a nonrestoring division algorithm that is basically a trial and error process [11]. It utilizes the following relationship:

$$P_{j+1} = rP_j - q_{j+1}D$$

To calculate a next partial remainder, the divisor is multiplied by the next quotient digit, and the result is subtracted from the product of the last partial remainder, or dividend for the first iteration, and a radix r. The next quotient digit is obtained by supplying a fixed number of bits from the last partial remainder, approximately 8 bits for a radix-4 divider, to a look-up table. By choosing a radix to be a power of 2, the product of the radix and the last partial remainder can be formed by shifting. Similarly, the various products of the divisor multiplied by the next quotient digit can be formed by multiplexing different multiples of the divisor. However, the problem with this basic scheme is that it requires a full-width subtractor. Consequently, this scheme can be very slow.

In order to improve upon this basic scheme, some redundancy is often introduced into the algorithm. An extra constraint is added to provide redundancy:

$$|P_{j+1}| < kD$$

where k = n / (r - 1) and n is the number of positive allowed digits for the next quotient digit. A design tradeoff can be noted in this relationship. By using a large number of allowed digits for the next quotient digit, and thus a large value for k, a smaller look-up table is required, and thus the complexity and latency of the table look-up can be reduced. However, choosing a smaller number of allowed digits for the quotient simplifies the generation of the multiple of the divisor. Multiples that are powers of two can be formed by simply shifting. If a multiple is required that is not a power of two (e.g. three), an additional operation such as addition may also be required, which can add to the complexity and latency of the divisor multiple generating process. Thus, the complexity of the look-up table and that of generating multiples of the divisor must be balanced.

To increase the performance of the subtraction process, the partial remainders themselves are often kept in a redundant form. Instead of using full-width adders that require carry propagation to compute partial remainders, a series of carry-save adders are used to compute the next partial remainder in the delay of a single full-adder. In this way, the conversion to a non-redundant form only needs to be done after the final iteration using a full width carry propagate adder. A block diagram of a basic SRT topology is shown in figure 1. The critical path of the topology is shown by the dotted line.

2.2 Multiplicative Algorithms

Multiplication based algorithms obtain a reciprocal of the divisor and multiply the result by the dividend. Flynn [2] shows that there are two main methods of iteration to evaluate the reciprocal: the series expansion, and the Newton-Raphson iteration. The two schemes are similar, and in fact, they have the same iteration under certain situations. Accordingly, only the Newton-Raphson iteration is discussed here.

In the Newton-Raphson algorithm, a function is chosen which has a root at the desired result. As the algorithm searches for the root, it creates a higher precision approximation of the result. For division:

$$Q = a/b = a * (1/b),$$

where Q is the quotient, a is the dividend, and b is the divisor. The algorithm is used to find an approximation to the reciprocal operation and then a multiplication is performed to calculate the quotient. The following function is chosen for the algorithm which has a root at the reciprocal:

$$f(X) = 1/X - b = 0$$

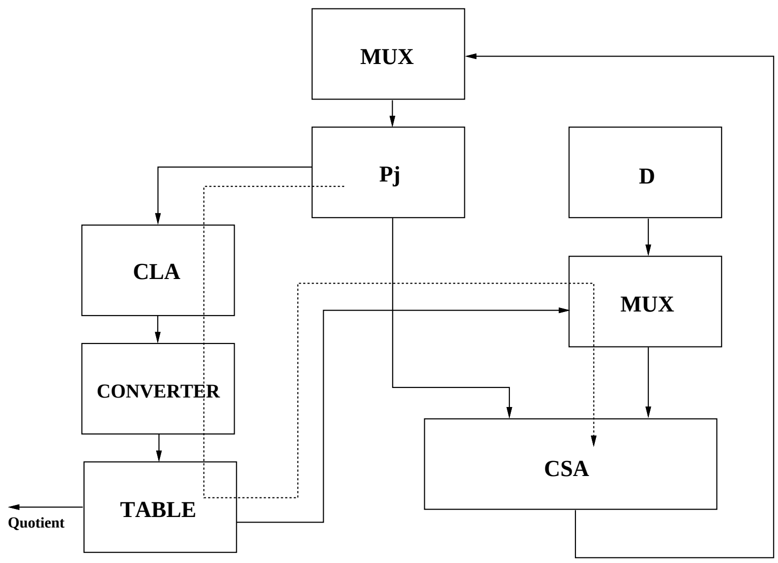

Figure 1: Basic SRT Topology

The function and its first derivative are evaluated at X_0:

$$\begin{aligned} f(X_0) &= 1/X_0 - b \\ f'(X_0) &= -1/X_0^2. \end{aligned}$$

These results are then used to find an approximation to the reciprocal:

$$\begin{aligned} X_1 &= X_0 - \frac{f(X_0)}{f'(X_0)} \\ X_1 &= X_0 + \frac{(1/X_0 - b)}{(1/X_0^2)} \\ X_1 &= X_0 * (2 - b * X_0) \\ &\vdots \\ X_{i+1} &= X_i * (2 - b * X_i) \end{aligned}$$

As can be seen from the general relationship, each iteration involves two multiplications and a subtraction. The subtraction is equivalent to the two's complement operation and is commonly replaced by it. Thus, two multiplications and one two's complement operation are performed each iteration.

As long as the second derivative of f is continuous and the approximation is sufficiently close, the Newton-Raphson algorithm converges quadratically. The bound for Newton-Raphson division is:

$$T = \lceil \log_2 n \rceil (\log_{3/2} 2n + \log_2 2n)$$

2.3 Comparison

The performance of these two methods of division computation can be compared by considering the latency for fixed length operands. A widely used standard today for floating-point representation is the IEEE 754 specification [3]. The standard has two precisions, single and double. A double precision operand is a 64-bit word, comprising a 1 bit sign, an 11 bit biased exponent, and 52 bits of mantissa, with one hidden mantissa bit. Thus, for this format, n = 53. Comparing the latency bounds for the three algorithms presented earlier, it can be seen that the bound for restoring division is 428 gates, for nonrestoring division it is 212 gates, and for Newton-Raphson it is 109 gates. From a latency perspective, the iterative Newton-Raphson algorithm seems to have provide the highest performance. However, other system issues must be considered when designing a divide functional unit besides overall latency. These issues include cycle time, area, availability of remainder, and the effects on other, possibly shared, functional units. Only when the issues are considered in the context of an entire system can design choices be accurately made.

Actual implementations of the Newton-Raphson algorithm have yielded latencies as low as 12 cycles. The fundamental issue in enhancing the performance of this form of division is to initiate the algorithm with as good an approximation (i.e. as many bits of the quotient) as possible. This is typically accomplished by the use of large look-up tables or reuse of existing multiplier hardware [9]. While Wong [14] reports very low latencies using an iterative scheme similar to Newton-Raphson, between 20 and 30 ns or about 3 to 6 cycles, the chip area requirement can be quite high, due to the very large look-up tables required. SRT implementations can have latencies of under 8 cycles. Williams [12] presents a self-timed SRT divider with a latency between 45 ns and 160 ns. In the technology of that divider, the latency translates into between 4 and 8 cycles. Oberman [6] reports how a radix-256 SRT implementation can achieve a latency under 8 cycles in a reasonable cycle time. The question still remains as to how these latencies actually affect system performance and what the trade-offs in these schemes are.

3 System Level Study

3.1 Instrumentation

As stated earlier, system performance was evaluated using the SPECfp92 benchmark suite. This suite contains 14 CPU intensive floating point applications. All but one of them are written in Fortran, and the other is written C.

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 global optimization, including code motion, code scheduling, and inlining of arithmetic statement functions. O3 performs all of O2's optimizations, but it also implements loop unrolling [7]. By varying the level of compiler optimization, two results were gained: how far apart a divide operation and the use of its result could be spaced, and also the dynamic frequency of divide operations as a percentage of the total number of operations. The compilers utilized the MIPS R3000 machine model with double precision FP latencies of add = 2 cycles, multiply = 5 cycles, and divide = 19 cycles.

The resulting binaries were then instrumented using *pixie*, which reads an executable file and partitions the program into its basic blocks. Pixie then writes a new version of the executable using this information which contains extra instructions to dynamically count the number of times each basic block is executed. Each benchmark was then executed with its standard input data set. As a result, each application executed approximately 3 billion instructions.

3.2 Method of Analysis

To determine the effects of FP divide on overall system performance, the performance degradation due to divide is written as:

$$CPI_{div} = F(f, u, l)$$

where f is the dynamic frequency of divide instructions, u is the urgency of divide results, and l is the functional unit latency of divide. It is clear that f is solely a function of the application, u is a function of the application and the compiler, and l is a function of the hardware. Thus, the system designer can only directly control l.

After the completion of execution, the program's execution characteristics were statically analyzed. The application code, in conjunction with the basic block counts from pixie, were used to compute many statistics regarding the dynamic execution.

4 Results

4.1 Instruction Mix

Figure 2 shows the arithmetic average of the frequency of divide operations in the benchmark suite relative to the total number of floating-point operations. This figure show that simply in terms of dynamic frequency, divide seems to be a relatively unimportant instruction, with about 3% of the dynamic floating-point instruction count due to divide. The dominant instructions are FP multiply and add. It should be noted that add, subtract, move, and convert operations typically utilize the FP adder hardware. Thus, FP multiply accounts for about 37% of the instructions, and the FP adder is used for about 55% of the instructions. However, in terms of latency, divide can play a much larger role. By assuming a machine model where every divide instruction takes 20 cycles to complete, and the adder and multiplier each take three cycles, a distribution of the stall time due to the FP hardware was formed as shown in figure 3. Here, FP divide accounts for about 40% of the latency, FP add accounts for about 42%, and multiply accounts for the remaining 18%. It is apparent that by improving the performance of divide, overall system performance can be improved.

4.2 Compiler Effects

In order to analyze the impact that the compiler can have on improving system performance, the *urgency* of divide results was measured as a function of compiler optimization level.

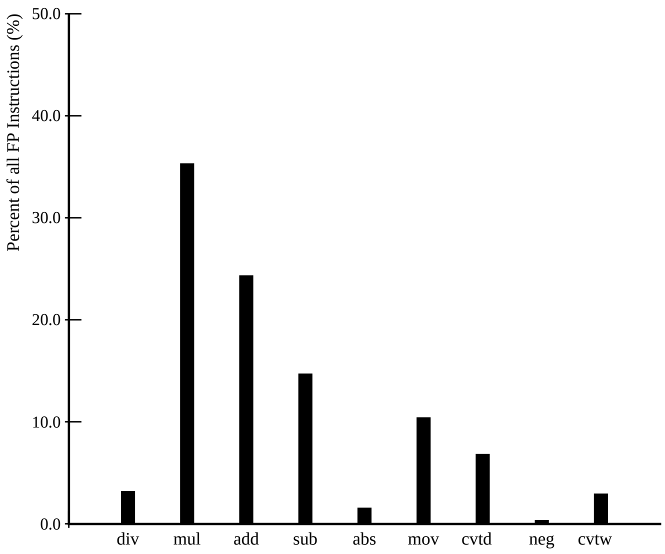

Figure 2: Instruction count distribution

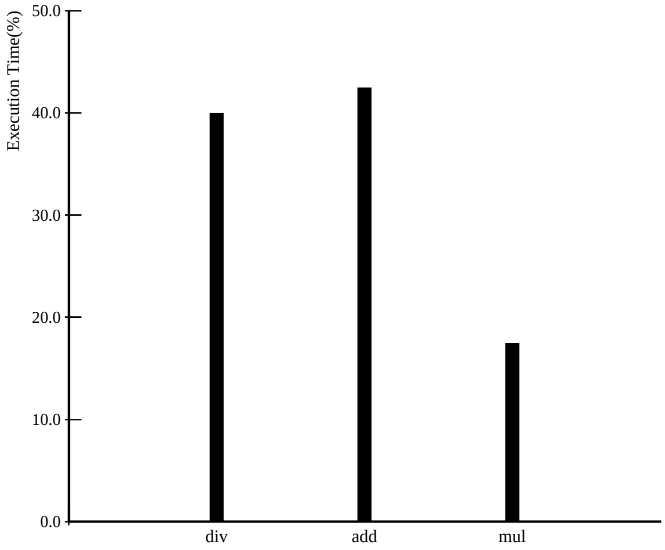

Figure 3: Functional unit stall time distribution

Figure 4 shows a histogram of the interlock distance for divide instructions at O0, as well as a graph of the cumulative interlock distance for the spice benchmark. Figure 5 shows the same data when compiled at O3. It is clear that by intelligent scheduling and loop unrolling, the compiler is able to expose instruction-level parallelism in the applications, decreasing u in the divide CPI function.

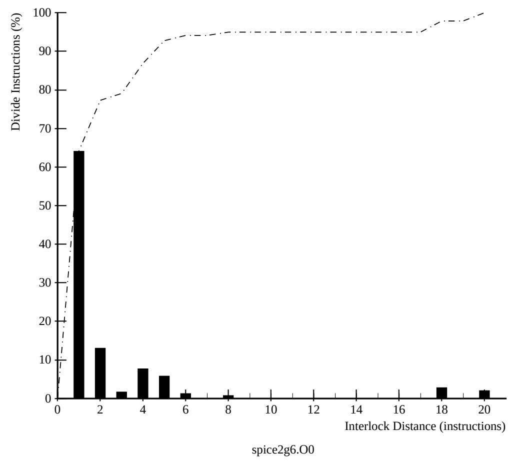

Figure 4: Spice with optimization O0

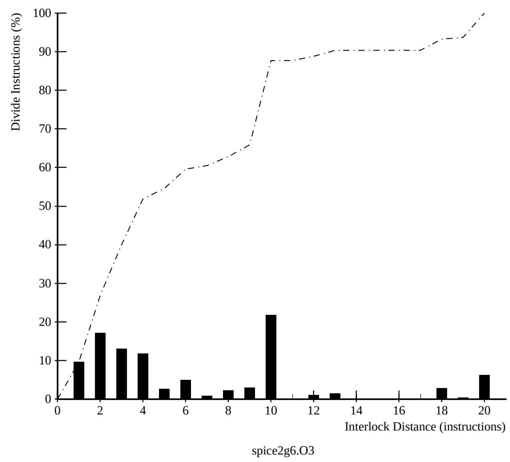

Figure 5: Spice with optimization O3

An average of the divide interlock distances from all of the benchmarks was formed, weighted by divide frequency in each benchmark. This result is shown in figure 6 for the three levels of compiler optimization. In this graph, the curves represent the cumulative number of divides at each distance.

The average number of stall cycles for a given latency divider was determined when executing these benchmarks. This result is shown in figure 7, again as a function of compiler optimization level. For this analysis, it is assumed that all instructions other than divide

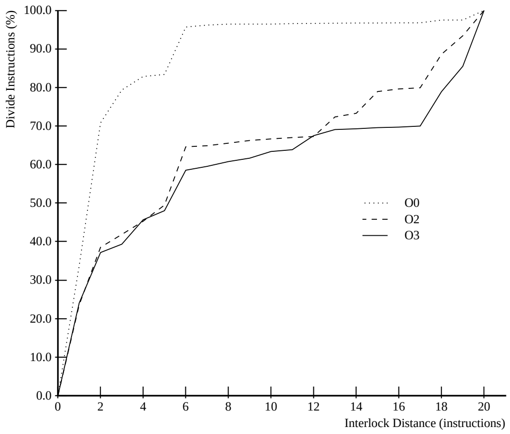

Figure 6: Cumulative average interlock distance

take one cycle, and that the memory system is perfect with no cache misses. This forms a worst case bound for the effective divide latency.

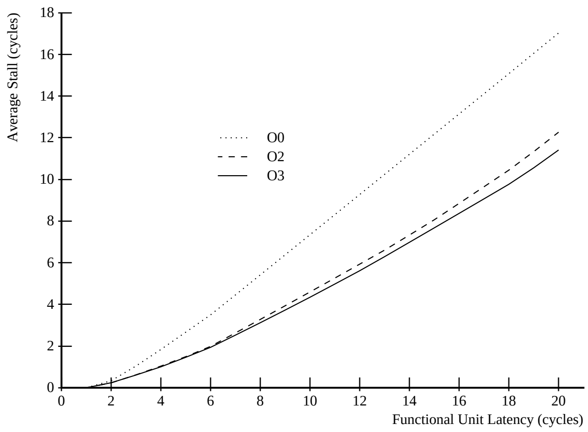

Figure 7: Average effective divide latency

4.3 Overall CPI

The effect of divide latency on overall performance is displayed in figure 8. This graph shows how excess CPI, in this case the CPI due to the divide interlocks, varies with divide unit latency between 1 and 20 cycles at an optimization level O3. Varying the optimization level also changed the total number of instructions executed, but left the number of divide instructions executed constant. As a result, the fraction of divide instructions is also a function of optimization level. While CPI due to divide actually increases from O0 to O2, the total performance at O2 and O3 would decrease because the total instruction count decreases. This effect is summarized in table 1.

Figure 8 also shows the effect of increasing the number of instructions issued per cycle on excess CPI due to divide. As the width of instruction issue increases, u increases for divide data proportionally. In the worst case, every divide result consumer could cause a stall equal to the functional unit latency.

Figure 8 also shows how area increases as the functional unit latency decreases. The data for the areas are based on layouts from [6, 12, 14], all of which have been normalized to 1.0μm MOSIS scalable CMOS layout rules. Clearly, as divide latencies decrease below 4 cycles, a large trade-off must be made. Either a very large area penalty must be incurred to achieve this latency by utilizing a large look-up table method, or large cycle times will result if an SRT method is utilized.

Optimization Level	Divide Frequency	Excess CPI for L=20
O0	0.33%	0.057
O2	0.76%	0.093
O3	0.79%	0.091

Table 1: Effects of compiler optimization

Figure 9 shows the excess CPI versus divide latency tradeoff over a larger range of divide latencies. The graphs can roughly be divided into five regions: for $l > 40$ cycles corresponds to very inexpensive 1-bit SRT schemes. They can contribute in the worst case up to 0.50 CPI in wide-issue machines. However, the area used by such schemes is small, under 2 mm^2. The second region corresponds to 2-bit SRT schemes, for $20 < l < 40$ cycles. The excess CPI in this region is $0.10 < \text{CPI} < 0.32$, with an area of approximately 2.1 mm^2. The third region corresponds to 4-bit SRT schemes, for $10 < l < 20$ cycles, with excess CPI in the range $0.04 < \text{CPI} < 0.10$, and area of approximately 2.75 mm^2. The range from $4 < l < 10$ cycles corresponds to 8-bit SRT as well as the self-timed SRT implementation of [12]. The CPI penalty here is $0.01 < \text{CPI} < 0.07$, with an area of approximately 4.5 mm^2. The final region consists of dividers with latencies less than or equal to 4 cycles. To achieve this performance, with $\text{CPI} < 0.01$, large area is typically required for very large look-up tables, often over 100 mm^2.

4.4 Shared Multiplier Effects

If a multiplicative based divide algorithm is chosen, it must be decided whether to use a dedicated multiplier for this purpose, or share the existing multiplier hardware. The area for a well-designed 3 cycle FP multiplier is around 11 mm^2, again using the 1.0μm process. Thus, 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 probably increase due to the modifications necessary to support the divide operation. Second, in some cases, as in the case of a scalar processor with a single multiplier, a multiply operation can be stalled due to a structural hazard that the divide operation has caused by sharing the

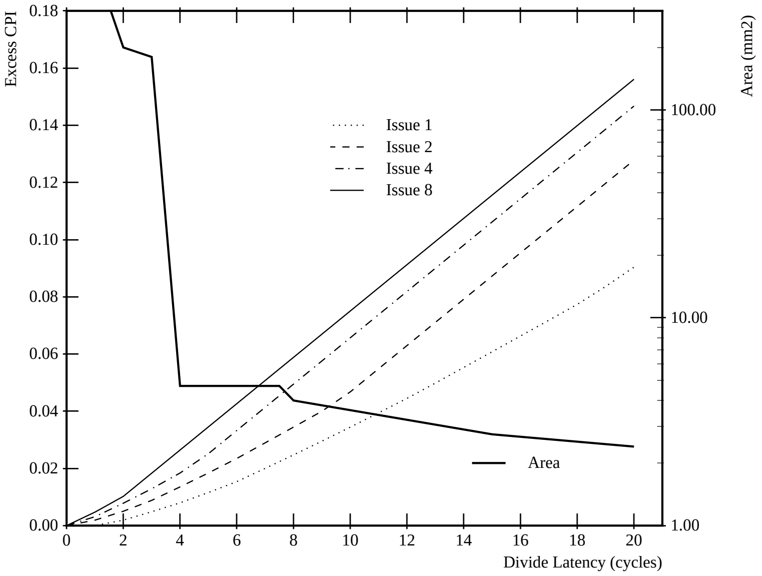

Figure 8: CPI and area vs divide latency

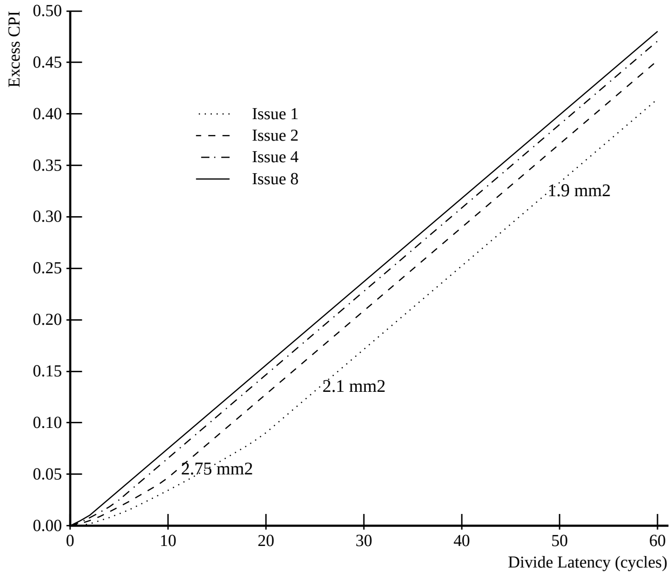

Figure 9: CPI and area vs divide latency

multiplier. The effect of this structural hazard on excess CPI is shown in figure 10. Here again, the results are based on an average of all of the applications when scheduled with O3. In all cases of the divide latency less than 20 cycles, the excess CPI is less than 0.07. For reasonable implementations of a multiplication based divide, with l approximately 12 cycles, the actual penalty is $0.02 < \text{CPI} < 0.04$. Accordingly, due to the relatively low frequency of divide operations, the penalty incurred for sharing an existing multiplier is not large.

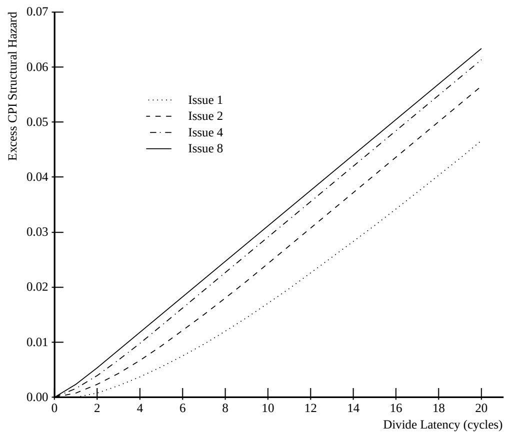

Figure 10: Excess CPI due to shared multiplier

4.5 On-the-fly Rounding and Conversion

In a nonrestoring division implementations 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. Additionally, to conform to the IEEE standard, it may be 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 [1]. The implementation of such a scheme is complex and is not discussed here. Because of its complexity, the designer may not wish to add this hardware to the divider. Figure 11 shows the performance impact of requiring an additional cycle after the divide operation completes. For divide latencies greater than 10 cycles, less than 20% of the total divide penalty in CPI is due to the extra cycle. At very low divide latencies, where l is less than or equal to 4 cycles, the penalty for requiring the additional cycle is obviously much larger, often greater than 50% of the total penalty.

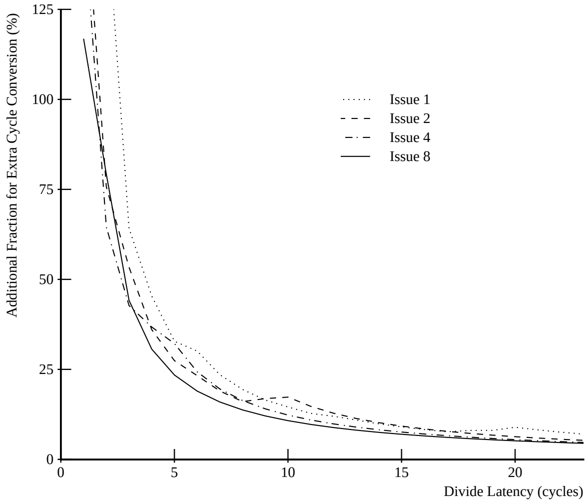

Figure 11: Effects of on-the-fly rounding and conversion

4.6 Consumers of Divide Results

In order to reduce the effective penalty due to divide, it is useful to look at which operations actually use the divide results. Figure 12 shows a histogram of instructions that use the divide results. This graph can be compared with that for multiply results, which appears in figure 13. For multiply results, the biggest users are the multiplier itself and the adder. 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 multiply-accumulate are reasonable. Because the multiply-add pattern occurs frequently in such applications, and it does not require much more hardware than a typical FP multiplier, fused multiply-adders are often used in machines today.

Looking at the consumers of divide results, the FP adder is the biggest 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 divide-store interlock, though, with other architectural implementations. A typical reason why a store would require a divide 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 divide 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 divide 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, this fused divide-add scheme could provide additional performance.

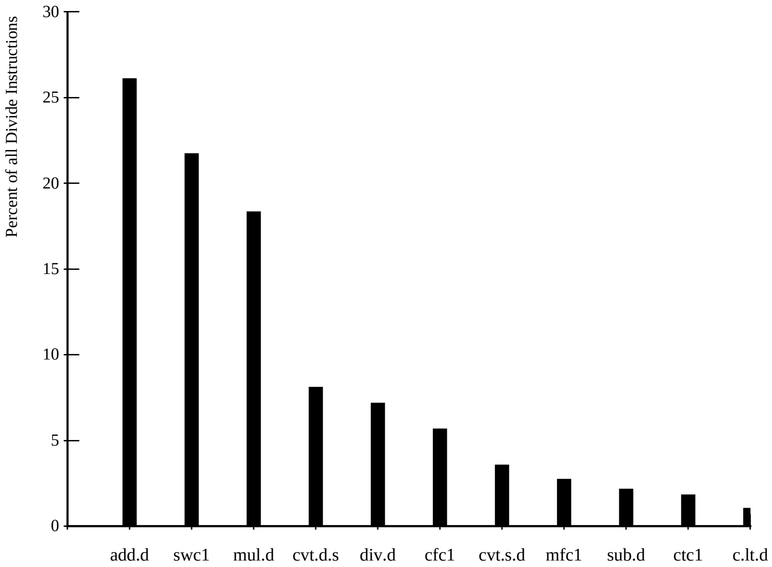

Figure 12: Consumers of divide results

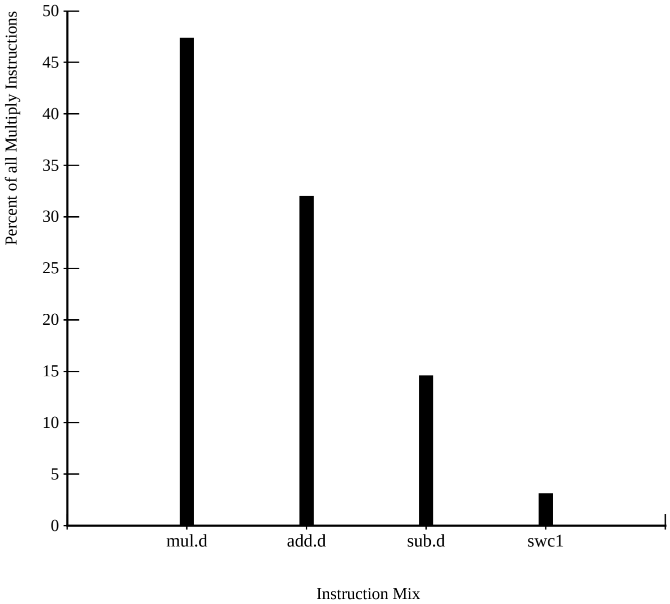

Figure 13: Consumers of multiply results

5 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 divide 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 first question asked was whether a hardware FP divide unit is necessary in a system. The data shows that for the slowest hardware divider, with $l > 60$ cycles, the CPI penalty is greater than 0.50. This indicates that to achieve reasonable system performance, some form of hardware divide is required. The compiler's ability to improve system performance due to divide was then investigated. The results showed the compiler's ability to decrease u, and so reduce the effective divide latency by 30%. Most of the performance gain was in performing basic compiler optimizations, at the level of O2. Only marginal improvement was gained by further optimization.

The effects of multiple issue on divide latency were then investigated. It was clear that increasing the number of instructions issued per cycle also increased the urgency u. On the average, increasing the number of instructions issued per cycle to 2 caused a 38% increase in CPI, increasing to 4 caused a 94% increase in CPI, and increasing to 8 caused a 120% increase in CPI. 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.

The question of whether an existing FP multiplier could be shared when using a multiplication based divide algorithm was then investigated. The results show that for a divide latency l of around 12 cycles, the CPI penalty is between 0.025 and 0.040. This result that due to the low frequency of divide operations combined with the low frequency of multiply instructions that happen to occur in-between the divide 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.

The final topic investigated was the necessity of on-the-fly rounding and conversion. For divide 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 probably not required.

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 components of implementing an FP divider in hardware.

6 Acknowledgement

The authors wish to thank N. Quach for his assistance throughout this work.

References

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

[2] M. Flynn. On division by functional iteration. *IEEE Transactions on Computers*, C-19(8), August 1970.

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

[4] *Microprocessor Report*, Various issues, 1994.

[5] NAS Parallel Benchmarks 8/91.

[6] 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 University, January 1994.

[7] MIPS compiler reference pages.

[8] Perfect Benchmarks, University of Illinois 1992.

[9] E. Schwarz. High-Radix Algorithms for High-Order Arithmetic Operations. Technical Report No. CSL-TR-93-559, Computer Systems Laboratory, Stanford University, January 1993.

[10] SPEC benchmark suite release 2/92.

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

[12] T. E. Williams and M. A. Horowitz. A Zero-Overhead Self-Timed 160-ns 54-b CMOS Divider. *IEEE Journal of Solid-State Circuits*, 26(11), November 1991.

[13] S. Winograd. On the time required to perform addition. *Journal ACM*, 12(2), 1965.

[14] D. Wong and M. Flynn. Fast Division Using Accurate Quotient Approximations to Reduce the Number of Iterations. *IEEE Transactions on Computers*, 41(8), August 1992.