A Variable Latency Pipelined Floating-Point Adder*

Stuart F. Oberman and Michael J. Flynn

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract. Addition is the most frequent floating-point operation in modern microprocessors. Due to its complex shift-add-shift-round dataflow, floating-point addition can have a long latency. To achieve maximum system performance, it is necessary to design the floating-point adder to have minimum latency, while still providing maximum throughput. This paper proposes a new floating-point addition algorithm which exploits the ability of dynamically-scheduled processors to utilize functional units which complete in variable time. By recognizing that certain operand combinations do not require all of the steps in the complex addition dataflow, the average latency is reduced. Simulation on SPECfp92 applications demonstrates that a speedup in average addition latency of 1.33 can be achieved using this algorithm while maintaining single cycle throughput.

1 Introduction

Floating-point (FP) addition and subtraction are very frequent floating-point operations. Together, they account for over half of the total floating-point operations in typical scientific applications [7]. Both addition and subtraction utilize 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.

Due to its many serial components, FP addition can have a longer latency than FP multiplication. Pipelining is a commonly used method to increase the throughput of the adder. However, it does not reduce the latency. Previous research has provided algorithms to reduce the latency by performing some of the operations in parallel. This parallelism is achieved at the cost of additional hardware. The minimum achievable latency using such algorithms in high clock-rate microprocessors has been three cycles, with a throughput of one cycle.

To further reduce the latency, it is necessary to remove one or more of the remaining serial components in the dataflow. In this study, it is observed that not all of the components are needed for all input operands. Two variable latency techniques are proposed to take advantage of this behavior and to reduce the average addition latency. To take advantage of the reduced average latency, it

* This work was supported by NSF under grant MIP93-13701.

is necessary that the processor be able to exploit a variable latency functional unit. Thus, the processor must use some form of dynamic instruction scheduling with out-of-order completion in order to use the reduced latency and achieve maximum system performance.

The remainder of this paper is organized as follows: Section 2 presents previous research in FP addition. Section 3 presents two forms of the proposed algorithm. Section 4 analyzes the performance of the algorithm. Section 5 is the conclusion.

2 FP Addition 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. Throughout this study, the analysis assumes IEEE double precision operands. An IEEE double precision operand is a 64 bit word, comprising a 1 bit sign, an 11 bit biased exponent, and a 52 bit significand, with one hidden significand bit [5].

2.1 Basic

The straightforward addition algorithm *Basic* requires the most serial operations. It has the following steps [13]:

1. Exponent subtraction: Perform subtraction of the exponents to form the absolute difference $|E_a - E_b| = d$.
2. Alignment: Right shift the significand of the smaller operand by d bits. The larger exponent is denoted E_f.
3. Significand addition: Perform addition or subtraction according to the effective operation, which is a function of the opcode and the signs of the operands.
4. Conversion: Convert the significand result, when negative, to a sign-magnitude representation. The conversion requires a two's complement operation, including an addition step.
5. Leading-one detection: Determine the amount of left shift needed in the case of subtraction yielding cancellation. Priority encode (PENC) the result to drive the normalizing shifter.
6. Normalization: Normalize the significand and update E_f appropriately.
7. Rounding: Round the final result by conditionally adding 1 unit in the last place (ulp), as required by the IEEE standard [5]. If rounding causes an overflow, perform a 1 bit right shift and increment E_f.

The latency of this algorithm is large, due to its many long length components. It contains two full-length shifts, in steps 2 and 6. It also contains three full-length significand additions, in steps 3, 4 and 7.

2.2 Two Path

Several improvements can be made to *Basic* in order to reduce its total latency. These improvements come typically at the cost of adding additional hardware. These improvements are based on noting certain characterists of FP addition/subtraction computation:

1. The sign of the exponent difference determines which of the two operands is larger. By swapping the operands such that the smaller operand is always subtracted from the larger operand, the conversion addition in step 4 and the rounding addition in step 7 become mutually exclusive operations, eliminating one of the three carry-propagate addition delays.
2. The full-length alignment shift and the full-length normalizing shift are mutually exclusive, and only one such shift need ever appear on the critical path. These two cases can be denoted *CLOSE* for $d \leq 1$, and *FAR* for $d > 1$, where each path comprises only one full-length shift [3].
3. Rather than using leading-one-detection after the completion of the significand addition, it is possible to predict the number of leading zeros in the result directly from the input operands. This leading-one-prediction (LOP) can therefore proceed in parallel with the significand addition using specialized hardware [4, 10].

An improved adder takes advantage of these three cases. It implements the significand datapath in two parts: the *CLOSE* path and *FAR* path. At a minimum, the cost for this added performance is an additional significand adder and a multiplexor to select between the two paths for the final result. Adders based on this algorithm have been used in several commercial designs [1, 2, 6]. A block diagram of the improved *Two Path* algorithm is shown in Fig. 1(a).

2.3 Pipelining

To increase the throughput of the adder, a standard technique is to pipeline the unit such that each pipeline stage comprises the smallest possible atomic operation. While an FP addition may require several cycles to return a result, a new operation can begin each cycle, providing maximum throughput. Figure 1(a) shows how the adder is typically divided in a pipelined implementation. It is clear that this algorithm fits well into a four cycle pipeline for a high-speed processor with a cycle time between 10 and 20 gates. The limiting factors on the cycle time are the delay of the significand adder (SigAdd) in the second and third stages, and the delay of the final stage to select the true result and drive it onto a result bus. The first stage has the least amount of computation; the *FAR* path has the delay of at least one 11 bit adder and two multiplexors, while the *CLOSE* path has only the delay of the 2 bit exponent prediction logic and one multiplexor. Due to the large atomic operations in the second stage, the full-length shifter and significand adder, it is unlikely that the two stages can be merged, requiring four distinct pipeline stages.

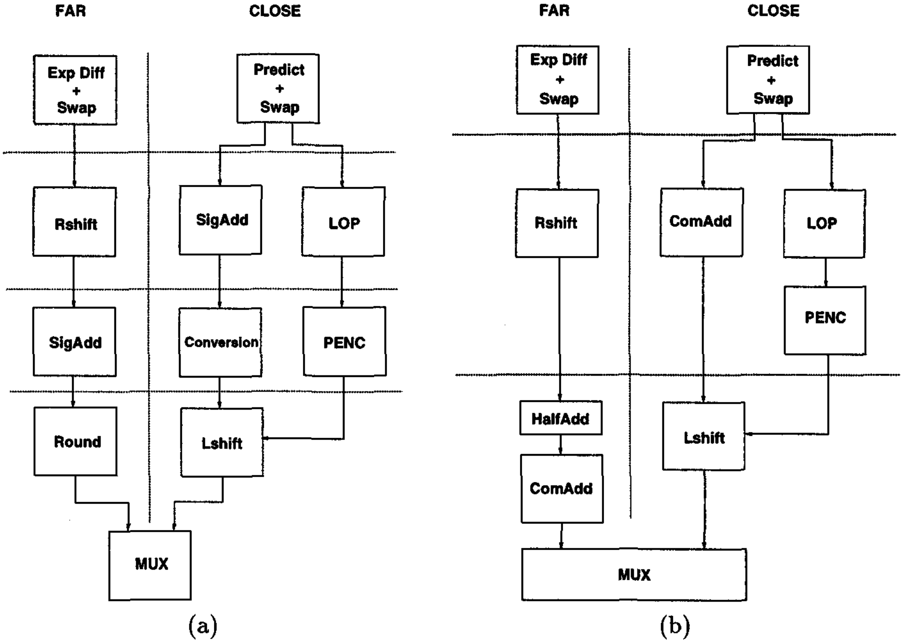


Fig. 1. Block diagram of (a) two path (b) combined rounding

When the cycle time of the processor is significantly larger than that required for the FP adder, it is possible to combine pipeline stages, reducing the overall latency in machine cycles but leaving the latency in time relatively constant. Commercial superscalar processors often have larger cycle times, resulting in a reduced FP addition latency in machine cycles when using the *Two Path* algorithm. In contrast, superpipelined processors have shorter cycle times and have at least a four cycle FP addition latency. For the rest of this study, it is assumed that the FP adder cycle time is limited by the delay of the largest atomic operation within the adder, such that the pipelined implementation of *Two Path* requires four stages.

2.4 Combined Rounding

A further optimization can be made to the *Two Path* algorithm to reduce the number of serial operations, as shown in Fig. 1(b). This optimization is based upon the realization 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, as described by Quach [8, 9]. Specifically, 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 *Two Path* requires that each significand adder 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. The rounding bits are the sign, LSB, guard, and sticky bits. This optimization removes one significand addition step. For pipelined implementations, this can reduce the number of pipeline stages from four to three. The cost of this improvement is that the significand adders in both paths must be modified to produce both *sum* and *sum+1*.

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 must be 1 for rounding to be required. The explicit addition *sum+2* is required for correct rounding in the event of overflow requiring a 1 bit normalizing right shift for RP and RM where the guard bit need not be 1. In [9], it is proposed to use a row of half-adders above the *FAR* path significand adder. These adders allow for the conditional pre-addition of the additional ulp to produce *sum+2*. The critical path in this implementation is in the third stage consisting of the delays of the half-adder, compound adder, multiplexor, and drivers.

3 Variable Latency Algorithm

From Fig. 1, it can be seen that 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 exponents. As previously stated, 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 operands, and a multiplexor is required to perform the swap.

3.1 Two Cycle

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. As it has been shown that the first stage in the *CLOSE* path completes very early relative to the *FAR* path, the addition of the second stage hardware need not result in an increase in cycle time.

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 a considerable reduction in the average latency can be achieved.

A result can be driven onto the result bus in either stage 2 or stage 3. Therefore, some logic is required to control the tri-state buffer in the second stage to ensure that it only drives a result when there is no result to be driven in stage 3. In the case of a collision with a pending result in stage 3, the stage 2 result is simply piped into stage 3. While this has the effect of increasing the *CLOSE* path latency to three cycles in these instances, it does not affect throughput. As only a single operation is initiated every cycle, it is possible to retire a result every cycle.

Scheduling the use of the results of an adder implementing *Two Cycle* is not complicated. At the end of the first cycle, the *FAR* path hardware will have determined the true exponent difference, and thus the correct path will be known. Therefore, a signal can be generated at that time to inform the scheduler whether the result will be available at the end of one more cycle or two more cycles. Typically, one cycle is sufficient to allow for the proper scheduling of a result in a dynamically-scheduled processor.

3.2 One Cycle

Further reductions in the latency of the *CLOSE* path can be made after certain observations. First, it can be seen that 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 $d \leq 2$, 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 adder is shown in Fig. 2.

The *One Cycle* algorithm allows a result to be driven onto the result bus in any of the three stages. As in the *Two Cycle* algorithm, additional control for the tri-state buffers is required to ensure that only one result is driven onto the bus in any cycle. In the case of a collision with a pending result in any of the other two stages, the earlier results are simply piped into their subsequent stages. This guarantees the correct FIFO ordering on the results. While the average latency may increase due to collisions, throughput is not affected.

Scheduling the use of the results from a *One Cycle* adder is somewhat more complicated than for *Two Cycle*. In general, the instruction scheduling hardware needs some advance notice to schedule the use of a result for another functional unit. It may not be sufficient for this notice to arrive at the same time as the data. Thus, an additional mechanism may be required to determine as soon as possible before the end of the first cycle whether the result will complete

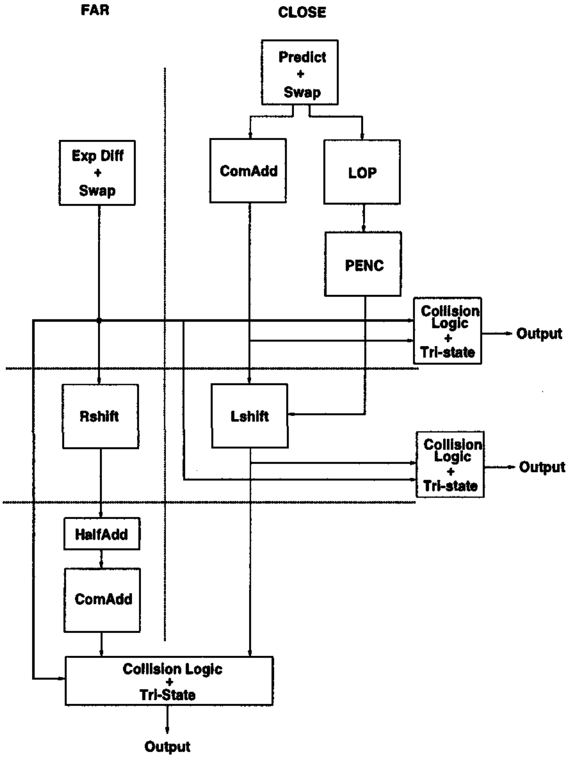
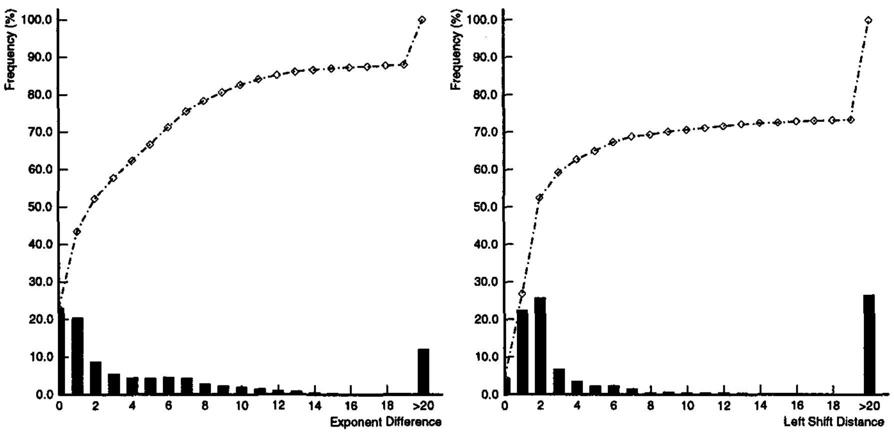


Fig. 2. Variable latency adder

either 1) in the first cycle or 2) the second or third cycles. A proposed method is as follows. First, it is necessary to determine quickly whether the correct path is the *CLOSE* or *FAR* path. This can be determined from the absolute difference of the exponents. If all bits of the difference except for the LSB are 0, then the absolute difference is either 0 or 1 depending upon the LSB, and the correct path is the *CLOSE* path. To detect this situation fast, an additional small leading-one-predictor is used in parallel with the exponent adder in the *FAR* path to generate a CLOSE/FAR signal. This signal is very fast, as it does not depend on exactly where the leading one is, only if it is in a position greater than the LSB.

Predicting early in the first cycle whether or not a *CLOSE* path operation can complete in one or two cycles may require additional hardware. Effective additions require no other information than the CLOSE/FAR signal, as all *CLOSE* path effective additions can complete in the first cycle. In the case of effective subtractions, an additional specialized leading-one-predictor can be included in the significand portion of the *CLOSE* path to predict quickly whether the leading one will be in any of the high order three bits. If it will be in these bits, then it generates a one cycle signal; otherwise, it generates a two cycle signal.



(a)

(b)

Fig. 3. Histogram of (a) exponent difference and (b) left-shift distance

An implementation of this early prediction hardware should produce a one cycle signal in less than 8 gate delays, or about half a cycle.

4 Performance Results

To demonstrate the effectiveness of these two algorithms in reducing the average latency, the algorithms were simulated using operands from actual applications. The data for the study was acquired using the ATOM instrumentation system [12]. ATOM was used to instrument 10 applications from the SPECfp92 [11] benchmark suite. These applications were then executed on a DEC Alpha 3000/500 workstation. The benchmarks used the standard input data sets, and each executed approximately 3 billion instructions. 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.

Figure 3(a) is a histogram of the exponent differences for the observed operands, and it also is a graph of the cumulative frequency of operations for each exponent difference. This figure shows the distribution of the lengths of the initial aligning shifts. It should be noted that 57% of the operations are in the *FAR* path with $E_d > 1$, while 43% are in the *CLOSE* path. An implementation of the *Two Cycle* algorithm therefore utilizes the two cycle path 43% of the time, yielding an average latency of 2.57 cycles. Thus, an implementation of the *Two Cycle* algorithm has a speedup in average addition latency of 1.17.

Implementations of the *One Cycle* algorithm reduce the average latency even further. 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% effec-

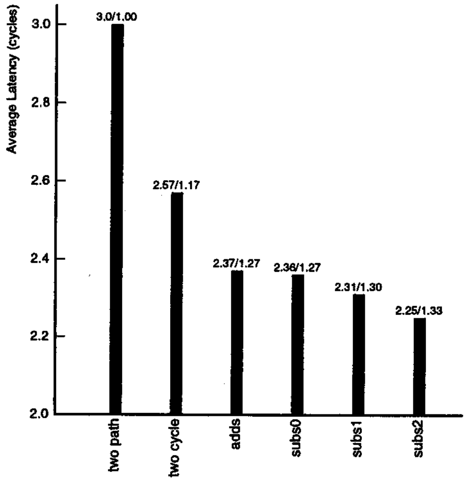


Fig. 4. Performance summary of proposed techniques

tive subtraction. As effective additions do not require any normalization in the close path, they complete in the first cycle. An implementation allowing effective addition to complete in the first cycle is referred to as *adds* has an average latency of 2.37 cycles, for a speedup of 1.27.

Figure 3(b) is a histogram of the normalizing left shift distances for effective subtractions in the *CLOSE* path. From Fig. 3(b), it can be seen that the majority of the normalizing shifts occur for distances of less than three bits. Only 4.4% of the effective subtractions in the *CLOSE* path require no normalizing shift. However, 22.4% of the subtractions require a 1 bit normalizing left shift, and 25.7% of the subtractions require a 2 bit normalizing left shift. In total, 52.5% of the *CLOSE* path subtractions require a left shift less than or equal to 2 bits. The inclusion of separate hardware to handle these frequent short shifts provides a performance gain.

Three implementations of the *One Cycle* algorithm could be used to exploit this behavior. They are denoted *subs0*, *subs1*, and *subs2*, which allow completion in the first cycle for effective subtractions with maximum normalizing shift distances of 0, 1, and 2 bits respectively. The most aggressive implementation *subs2* reduces the average latency to 2.25 cycles, for a speedup of 1.33.

The performance of the proposed techniques is summarized in Fig. 4. For each technique, the average latency is shown, along with the speedup provided over the base *Two Path* FP adder with combined rounding that has a fixed latency of three cycles.

5 Conclusion

This study has presented two techniques for reducing the average latency of FP addition. Previous research has shown techniques to guarantee a maximum latency of three cycles in high clock-rate processors. This study shows that additional performance can be achieved in dynamic instruction scheduling processors by exploiting the distribution of operands that use the *CLOSE* path. It has been shown that 43% of the operands in the SPECfp92 applications use the *CLOSE* path, resulting in a speedup of 1.17 for the *Two Cycle* algorithm. By allowing effective additions in the *CLOSE* path to complete in the first cycle, a speedup of 1.27 is achieved. For even higher performance, an implementation of the *One Cycle* algorithm achieves a speedup of 1.33 by allowing effective subtractions requiring very small normalizing shifts to complete in the first cycle. These techniques do not add significant hardware, nor do they impact cycle time. They provide a reduction in average latency while maintaining single cycle throughput.

References

1. B. J. Benschneider, et al. A pipelined 50-Mhz CMOS 64-bit floating-point arithmetic processor. *IEEE Journal of Solid-State Circuits*, 24(5):1317–1323, October 1989.
2. M. Birman, A. Samuels, G. Chu, T. Chuk, L. Hu, J. McLeod, and J. Barnes. Developing the WTL 3170/3171 Sparc floating-point co-processors. *IEEE Micro*, 10(1):55–63, February 1990.
3. M. P. Farmwald. *On the Design of High Performance Digital Arithmetic Units.* PhD thesis, Stanford University, August 1981.
4. E. Hokenek and R. K. Montoye. Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating-point execution unit. *IBM Journal of Research and Development*, 34(1):71–77, January 1990.
5. ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.
6. P. Y. Lu, A. Jain, J. Kung, and P. H. Ang. A 32-mflop 32b CMOS floating-point processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 28–29, 1988.
7. S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. In press *IEEE Transactions on Computers*, 1996.
8. N. Quach and M. Flynn. Design and implementation of the SNAP floating-point adder. Technical Report No. CSL-TR-91-501, Computer Systems Laboratory, Stanford University, December 1991.
9. N. T. Quach and M. J. Flynn. An improved algorithm for high-speed floating-point addition. Technical Report No. CSL-TR-90-442, Computer Systems Laboratory, Stanford University, August 1990.
10. N. T. Quach and M. J. Flynn. Leading one prediction - implementation, generalization, and application. Technical Report No. CSL-TR-91-463, Computer Systems Laboratory, Stanford University, March 1991.
11. SPEC Benchmark Suite Release 2/92.
12. A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.
13. S. Waser and M. Flynn. *Introduction to Arithmetic for Digital Systems Designers.* Holt, Rinehart, and Winston, 1982.