Floating-Point Arithmetic on a Reduced-Instruction-Set Processor

Thomas Gross
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Current single chip implementations of reduced-instruction-set processors do not support hardware floating-point operations. Instead, floating point operations have to be provided either by a co-processor or by software. This paper discusses issues arising from a software implementation of floating point arithmetic for the MIPS processor, an experimental VLSI architecture. Measurements indicate that an acceptable level of performance is achieved, but this approach is no substitute for a hardware accelerator if higher precision results are required. This paper includes instruction profiles for the basic floating point operations and evaluates the usefulness of some aspects of the instruction set.

1. Introduction

A reduced-instruction-set processor like the Stanford MIPS does not provide instructions for floating point arithmetic [10, 2]. There are several reasons for the exclusion of floating point operations from the hardware level instruction set. First, the execution of a floating point operations takes significantly longer than the execution of an integer addition or load-register operation. Therefore, floating point operations do not fit well the pipeline structure of MIPS; control of the MIPS pipeline relies heavily on the fact that all instructions take the same time to execute. Second, floating point operations are encountered only infrequently in the projected applications [2]. Furthermore, the limitation of resources available demands to focus attention on the most frequent instructions. And the size of the silicon area that can be fabricated places hard constraints on the complexity of the processor. The relevance of these arguments will change over time (for example, advances in fabrication technology allow implementation of larger designs), but when the design of MIPS started (1981), full 32-bit integer multiplication or floating point operations could not be accomodated.

This research is supported in part by the Defense Advanced Research Projects Agency under contract # MDA903-79-C-0680.

Similar reasons motivated the designers of the Berkeley RISC machine to exclude floating point arithmetic [9]. They felt that the available area of silicon was better spend on a large register file to speed up function and procedure call. The advantages of reduced-instruction-set architectures are discussed extensively in recent papers [3, 12, 8].

In the absence of hardware instructions for floating point operations, other implementations have to be found. There are two alternatives: either attach a floating point co-processor to a streamlined instruction set processor, or implement floating point arithmetic by software routines. The first approach has the advantage that it provides high-speed floating point operations. On the other hand, the development of such a co-processor requires a major commitment of resources, if no commercially available co-processor fullfills the performance goals or (more likely) the communication requirements.

We think that a floating point co-processor is the best approach to provide fast and high precision floating point operations, but there are also good reasons to consider software implementations. First, not all classes of programs involve a high degree of floating point operations. These operations might not be frequent enough to justify the additional hardware. Second, a software implementation of floating point operations can be achieved in less time and with much less effort than a hardware implementation. Lastly, software routines for floating point operations allow further evaluation of the architecture. These routines provide an interesting test for a streamlined processor's ability to synthesize complex operations from basic primitives.

For these reasons we undertook a software implementation for the MIPS processor; a similar development is reported for the Berkeley RISC machine [13]. The primary purpose of this effort is to study the consequences of a streamlined (or reduced) instruction set. There are two topics that are of concern to us: the necessary properties for floating point operations (that is, those properties of the instruction repertoire of the processor that are required to get the job done), and the usefulness of other operations that faciliate the implementation but could be

substituted. Before we address these points in detail, we describe briefly the floating point format in the next section.

2. Floating point format

Several different major formats exist for the representation of floating point numbers. Two of these formats are good candidates for our implementation: the representation chosen for DEC's family of VAX computers [7], and the IEEE floating point standard [5]. Most of our environment is based on a VAX-11/780, our simulator for MIPS executes on a VAX-11/780. Adopting the VAX representation would ease migration between these processors. Furthermore, the VAX floating point hardware could be used as a "co-processor" during simulation. However, the IEEE floating point standard offers a portable base between different microprocessors, and the advent of high-speed floating point multiplier and adder chips for this format simplifies the design of a custom co-processor. We therefore decided to use the IEEE floating point standard. Figure 2-1 shows the layout of the single precision format.

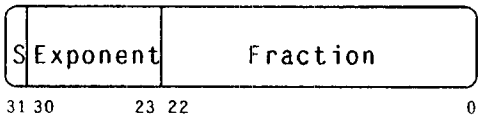

Figure 2-1: Single precision format

The fraction, F, is stored in bits 22 through 0; the leading '1" is hidden for normalized numbers. Bits 30 to 23 hold the exponent E, which is stored with a bias of -127. Bit 31 is the sign bit. Zero is represented by $E = 0$ and $F \neq 0$. Thus the number n is defined by $n = (-1)^{n.S} \times 2^{n.E-127} \times (1 + 2^{-23}\ n.F)$.

The value $E = 255$ is reserved to represent NotANumber (NAN) and (signed) infinity. The combinations of E=0 and F ≠ 0 an be used to encode denormalized numbers. One advantage of this format over the VAX representation is that it places the sign of the floating point numbers in the leftmost bit. This position simplifies to test if a number is greater or less than zero. Table 2-1 shows some further data that characterize our floating point. The precision of this floating point representation is $\varepsilon = 2^{-23} = 1.19 \times 10^{-7}$, that is the seven leading decimal digits can be trusted at best.

Maximum positive number	1.7×10^{38}
Minimum positive number	1.2×10^{-38}
Minimum positive number,denormalized	1.4×10^{-45}

Table 2-1: Range of the floating point format

The current implementation uses the IEEE floating point *data format* but does not fulfill the requirements of the IEEE floating point *standard*. Only one rounding mode, round to nearest, is supported. A trap is taken whenever underflow, overflow, or gradual underflow occurs, but the trap handler maintains only counts of the frequency of these events[1]. That is, Not-a-Number and denormalized numbers are detected, but they are not treated as required by the standard. Adding the functionality required by the standard would involve some tedious programming. But the instruction frequencies reported in this paper would hardly change since exceptions occur rarely.

No implementation of double precision or extended arithmetic is available at this time, but Section 4.4 contains estimates for double precision arithmetic. The primary purpose of our study is to investigate the characteristics of a software realization of floating point arithmetic for the MIPS processor. The inclusion of extended single precision arithmetic does not provide any new insights but should be included in a stabilized version [4]. Our first goal is to provide a basic level of support for those programs that require *some* floating point computation. The execution of floating point intensive programs is not a target for a software based implementation. Double precision software routines can be added when the need arises.

3. The MIPS instruction set

The architecture and instruction set of the MIPS processors has been described in previous papers [2, 10, 1]. It is useful for the understanding of the next section to keep in mind that MIPS is a load/store architecture. Only register operands can be used for arithmetic or logic operations. MIPS provides addition and subtraction for 32-bit integers. To implement multiplication and division, a two-bit Booth multiply-step or a one-bit non-restoring divide-step can be executed repeatedly. The Booth operations require that the operands reside in two special registers; these registers are called the High (Hi) and the Low (Lo) register.

The datapath of the MIPS processor contains a barrel shifter that implements arithmetic shifts, logical shifts, and rotation. The shift amount (in the range 0 to 31) is either an immediate value taken from the instruction or supplied by the Lo register. That is, a general purpose register cannot hold the shift amout, this value has to be moved to the special register.

The MIPS hardware instruction set allows the comparision of signed and unsigned integers in a "test-and-branch" instruction. Unfortunately this hardware cannot be used directly to compare two floating point numbers, which are in a signed magnitude format. A quick look at the format (see Figure 2-1) convinces that it is permissible to interpret two floating point numbers A and B as signed integers unless both A and B are less than zero. If both arguments of a comparision are less than zero, the bits of the

[1] So far the floating point routines are used with performance benchmarks - and these programs do not contain exceptions.

exponent and mantissa must be negated. This operation produces two numbers A' and B', which deliver the correct result when tested as integers by the hardware.

The data in the subsequent sections are gathered with a simulator. We assume in our translation from cycles to real time that the fabricated chip executes at a rate of two million instructions per second [11].

4. Floating point operations

Our implementation of the floating point operations consists of code sequences for single precision addition, subtraction, multiplication, and division. Furthermore, we have routines to convert between integers and floating point numbers. These code sequences contain conditional branches and therefore the execution time (for example, for an addition) depends on the data values of the input. The data presented in this section are averages obtained from the execution of three sample programs:

MatrixMult multiplies two matrices of dimension 40 by 40: the matrix elements are random numbers

Euler computes the Euler's constant e with a Taylor series

FFT a 256-point complex FFT.

The difference in the averages that are obtained from the individual programs is negligible, but not all operations are used in every benchmark.

All floating point routines require the use of registers to carry out the computations. Since we intend to use a floating point co-processor when it is available but do not want to maintain two different code generators for the two different environments, all responsibility for storing and restoring registers rests with the floating point operations. The code generator is unaware of the presence (or absence) of floating point hardware. If no floating point co-processor is available, a filter program converts a floating point operation into the call of the appropriate subroutine.

Although this arrangement places no demands on the rest of the compiler, it creates a high overhead. If two floating point operations are executed in sequence, everyone of them saves and restores the set of working registers. Two options exist to reduce the overhead. One option calls for a simple peephole optimizer that removes redundant restore/save operations. An alternative is to reconsider the decision to handle floating point operations independent of their implementation. This option requires that the code generator frees some registers for use of the floating point routines.

Table 4-1 gives the basic execution times for single precision operations. It provides a breakdown in time spend computing the result and time required to save and restore the temporary registers.

Operation	Generic Operation	Overhead
addition	25 μs	9 μs
subtraction	27 μs	9 μs
muliplication	14 μs	7 μs
division	19 μs	7 μs
conversion (integer to float)	15 μs	7 μs
conversion (float to integer)	19 μs	4 μs

Table 4-1: Execution times for operations

There are several reasons why addition and subtraction are more expensive than multiplication. For addition and subtraction, computing the sign bit and the result exponent are more complex, and the leading one (of the sum) can be in any bit position.

The next table shows the contribution of the floating point operations to overall program execution for the three benchmark programs. The first row excludes the overhead, which is separately given in the second row. The final row shows the overall contribution of floating point operations (including overhead) to total program execution. We note that about 20% of the total execution time is spend on overhead.

	FFT	MatrixMul	Euler
add/sub	40.41 %	33.58 %	24.05 %
overhead for add/sub	13.85 %	11.40 %	7.90 %
mul	12.85 %	19.38 %	0.00 %
overhead for mul	6.51 %	8.86 %	0.00 %
div	2.11 %	1.22 %	17.03 %
overhead for div	0.77 %	0.44 %	6.14 %
conversion	1.65 %	0.99 %	14.04 %
overhead for conv.	0.72 %	0.44 %	6.14 %
total FP	78.90 %	76.31 %	75.30 %

Table 4-2: Contribution of floating point operations [of total program execution]

The basic execution times given in Table 4-1 represent only one viewpoint. Measurement of total execution time is another important aspect. Table 4-3 presents the execution times for the Pascal versions of the benchmark programs.

Program	Time
Euler(1000 times)	1.14 sec
MatrixMult	5.05 sec
FFT	5.85 sec

Table 4-3: Execution times for benchmarks

Execution of the C version of the MatrixMult program on RISC-I takes 22.5 sec [13] and was timed at 57.7 sec on an 8 MHz M68000. For comparision, a VAX 11-780 with a floating point accelerator executes this program in 2.1 sec with double precision arithmetic.

In the remainder of this section, we now discuss the implementation of the five basic operations in some detail. The code for each of the floating point operations can be broken down into four parts. First, the registers that are needed to carry out the computation (and to hold intermediate results) are saved. Then, the exponent and fraction are moved into separate registers. Next, after some preliminary checks (i.e., divisior ≠ 0)), the computation of the result takes place. Then, the result is assembled and exponent and fraction are placed into one register. Finally, the registers saved in the first part are restored, and execution resumes in the main program. Since the overhead (the first and the last part) are due to our decision to provide transparency to the code generator, we exclude these parts from our summary.

4.1. Addition/Subtraction

The implementation of addition and subtraction proves to be the most difficult. Both operations are implemented by the same routine; the sign of the subtrahend is changed for subtraction. These operations take the longest amount of time (compare Figure 4-1), and they also require the most instructions statically (89 instructions, 24 of which count the leading zeroes and are also used in other parts of the floating point system). Seven registers are needed to compute the result and one register holds the returnaddress; that is, eight registers have to be saved and restored. The pre-addition alignment and post-addition normalization are responsible for the number of registers needed. 64% of the instructions are dedicated to this phase. Computation of the fraction and exponent is a mere 21%; the rest of the space (15%) is divided among the field extraction at the beginning (9%) and negation of the operand for subtraction (6%). Table 4-4 gives the actual instruction usage for a series of additions and subtractions.

Instruction group	Static	Dynamic
Add/subtract	20.22 %	20.82 %
Bit operations	6.74 %	7.03 %
Booth operations	0.00 %	0.00 %
Byte insert/extract	7.87 %	7.95 %
Load/store Hi/Lo register	11.24 %	7.79 %
Load/store normal register	3.37 %	4.50 %
Rotate operations	6.74 %	5.22 %
Shift operations	22.45 %	23.23 %
Total ALU operations	78.65 %	76.55 %
Branch uncond.	4.49 %	2.25 %
Branch conditionally	12.36 %	15.22 %
Load direct	2.25 %	2.99 %
Trap conditionally	2.25 %	2.99 %

Table 4-4: Instruction frequencies for addition/subtraction

The Load-direct instructions are used exclusively to retrieve masks and constants. Traps test for illegal results, for example an underflow, and transfer control to the exception and interrupt handler. The contribution of branches (conditional and unconditional) is much higher for this routine than for the multiplication or division routines. The reason is the extensive case analysis that must be done in this routine. This topic is discussed again in Section 5.3.

4.2. Multiplication

The core of the multiplication routine requires 49 instructions. The implementation of this operation is greatly faciliated by the special features for integer multiplication and division [6]. The Booth multiply instructions allow the multiplication of two 24-bit numbers in six words, since two ALU operations can be performed in one instruction cycle. This routine requires six temporary registers. Computation of the fraction consumes 30% of the 49 instructions, rounding and result composition 44%, computation of the exponent (and test for zero operands) 12%, and field extraction 14%. Table 4-5 gives the detailed breakdown of instructions used for the multiplication routine.

Instruction group	Static	Dynamic
Add/subtract	10.20 %	10.16 %
Bit operations	6.12 %	4.40 %
Booth operations	26.53 %	26.44 %
Byte insert/extract	10.20 %	10.83 %
Load/store Hi/Lo register	18.36 %	18.30 %
Load/store normal register	2.04 %	2.03 %
Rotate operations	8.16 %	8.80 %
Shift operations	6.12 %	6.10 %
Total ALU operations	87.76 %	87.13 %
Branch uncond.	2.04 %	2.03 %
Branch conditionally	4.08 %	4.73 %
Load direct	2.04 %	2.03 %
Trap conditionally	4.08 %	4.07 %

Table 4-5: Instruction frequencies for multiplication

22.38% of all instructions are Booth multiply-steps, which compute two bits of the product; the large number of multiply-steps overshadows the other ALU operations. Loading the Lo register is the next frequent operation (10.17%). This register is loaded for two purposes; the multiplier must be moved into this register, and it contains the byte selector for insert/extract byte operations.

4.3. Division

The software routines for floating point division on MIPS takes 30% more time than multiplication, but division is still faster than addition or subtraction. Division is about 50% shorter than addition; it contains 61 instructions. The one-bit divide-step instruction is useful in forming the quotient. 57% of the total space is dedicated to the computation of the fraction, 13% are used for field extraction and operand test, and 27% for quotient assembly and adjustment of the exponet. The division routine needs six working registers.

Instruction group	Static	Dynamic
Add/subtract	6.56 %	5.21 %
Bit operations	3.28 %	1.72 %
Booth operations	44.26 %	46.51 %
Byte insert/extract	8.20 %	8.62 %
Load/store Hi/Lo register	9.84 %	10.33 %
Load/store normal register	3.28 %	3.45 %
Rotate operations	1.64 %	1.72 %
Shift operations	11.48 %	10.38 %
Total ALU operations	88.52 %	87.94 %
Branch uncond.	1.64 %	1.72 %
Branch conditionally	3.28 %	3.45 %
Load direct	1.64 %	1.72 %
Trap conditionally	4.92 %	5.17 %

Table 4-6: Instruction frequencies for division

The divide-step instruction produces only one bit of the quotient in each step. 44.79% of all instructions are divide-steps. Load Lo register and shift right with sign extension are the two next frequent operations (6.89%).

4.4. Double precision operations

Our library of single precision operations allows us to estimate the timing of double precision operations. Double precision addition or subtraction takes approximately 35μs (plus 12μs overhead), and a double precision multiplication requires 42μs and an additional 11μs overhead to restore temporary registers. The multiplication routine is dominated by mstep operations that compute four partial products, which are combined to form the 53-bit fraction.

Double precision division is more expensive: the divide-step instruction cannot be used directly in the computation of a quotient A + B. Division must be decomposed in the evaluation of the reciprocal 1/B, followed with a multiplication by A. The evaluation of the reciprocal can be based on the well-known Newton-Raphson algorithm with a seed obtained by successive divide-step operations. A seed that is accurate in 30 bits requires only one Newton-Raphson iteration step to yield the 53-bit fraction of the reciprocal, and computation of the quotient is complete after 131μs (plus 12μs overhead).

5. Experience with floation point operations for MIPS

The measurements allow us to evaluate some aspects of the instruction set of the MIPS processor. Of special interest are those instructions that proved difficult to implement [10] and the potential benefits of the count-leading-zero instruction. This latter instruction was proposed but removed from the final architecture.

5.1. Instruction packing

The MIPS processor executes two ALU operations in one cycle, and the compiler (reorganizer) attempts to rearrange instructions so that two useful operations are executed. Often the instructions formats used prohibit full utilization of the hardware resources; encoding constraints disallow packing. Table 5-1 shows the dynamic density of the assembled floating point routines. A density of 115.48% indicates that 115.48 instructions are executed per 100 machine cycles. The density of the multiplication and division routines is significantly higher than the density found in other benchmark programs; the high density is one reason for the short execution time.

Operation	#Instructions	#MIPS Words	Density
Addition/Subtraction	89	75	115.48 %
Multiplication	39	32	152.89 %
Division	61	40	152.66 %

Table 5-1: Packing of floating point routines

5.2. Unused ALU instructions

The floating point programs use a wider spectrum of ALU operations than our set of integer Pascal benchmarks. Shift instructions occur in all routines, their contribution ranges from 23% (addition) to 6% (multiplication).

Nevertheless, several ALU instructions are not used at all by the floating point routines. Bit-wise "or" and "not" are two examples of bit operations that do not occur in these routines. Shift-logical, which extends the shifted argument with zeros, is used only with an immediate shift-amount. That is, the shift-amount is never supplied from the Lo register for logical shifts (both right and left).

The routine for addition and subtraction requires that the shift-amount is determined at run time, depending on the input data value. Two instructions read the shift-amount from the Lo register; they occur during the alignment of operands and in the

normalization phase. These instructions amount to 3.00% of all instruction executions in the addition subroutine. Both of these operations could be replaced with a sequence of tests and shifts to obtain the same result. This substitution increases the static size of the subroutine by 42 words (that is 45%), but the dynamic effect is less severe. Using the data about the frequency and range of shift operations gathered by Sweeney [14], we estimate that the execution time of floating point addition increases by 14 cycles. This is an increase of 7 μs - an increase of about 27%. The inclusion of dynamic shift amounts in the data path and instruction set cannot be justified in light of this isolated performance loss.

5.3. Count leading zeros

The first architecture document of the MIPS processor included an instruction to count the number of leading zeros of an operand. The implementation turned out to be cumbersome, and the designers decided against wasting a great deal of effort for this instruction. Instead, the plan called for a software routine to implement the desired functionality. This evaluation indicates that this decision was correct.

The code to count the leading zeros has a length of 23 MIPS instruction words. On the average, 18 cycles (9 μs) are spend in this section of the addition routine. The code sequence uses extensive case analysis: 50% of all conditional branches in the addition routine are found in this section. A single cycle count-leading-zeros instruction would decrease the execution time of addition to 16 μs (and subtraction to 18 μs), but the effect on overall program execution would be less pronounced. The execution of FFT would be shortened by 12%, the effect on MartrixMult is 10%.

5.4. Summary

Table 5-2 summarizes the influence of these architectural features on the exeuction times for the floating point primitives. The execution times (as measured by the number of instruction executed) are scaled relative the MIPS baseline as implemented; a number greater than 1 indicates that the operation is slowed down by the proposed change. Similarily, a factor less than 1 represents a potential speed up — if the feature can be included without a negative effect on the basic cycle time of the processor.

	Addition	Multiplication	
MIPS baseline	1.00	1.00	1.00
Elapsed time [Table 4-1]	(34μs)	(21μs)	(26μs)
Only 1 instruction/word	1.12	1.35	1.39
No shift amount in register	1.20	1.00	1.00
Only 1 bit Booth multiply step	1.00	1.19	1.00
Include "Count leading zeros"	.71	1.00	1.00
Compiler optimizes overhead	.74	.71	.77

Table 5-2: Contribution of architectural features

6. Concluding remarks

We have demonstrated that single precision arithmetic can be implemented on a reduced-instruction-set processor. This implementation illustrates again that it is advantagous to synthesize complex instructions from a set of simple operations. The performace of this implementation compares favorably against software implementations for other processors but cannot replace special arithmetic units for floating point operations.

The current linkage conventions for floating point arithmetic occur a high overhead but simplify the code generator. This organization should be reconsidered in an environment that uses these routines extensively.

The inclusion of logical and arithmetical shift instructions, two-bit multiply-step instruction, and one-bit divide-step instructions simplified the implementation of the software routines. The ability to obtain the shift-amount from a register (and not only from an immmediate field) is not essential; it saved 7 μs in the routine for addition. The penalty imposed by the absence of a single-cycle count-leading-zero instruction is in the same range (9 μs).

Instead of adding new instructions, other alternatives provide a higher speed-up. For example, if the code generator frees the working registers for the software routines, each floating point operation is shortened (on the average) by 6 μs. And the overall improvement will be higher than any single modifiction of the instruction set can achieve.

Acknowledgments

Many persons contributed to the MIPS project, and John Hennessy, John Gill, Norman Jouppi, Steven Przybylski, and Christopher Rowen are major contributors. John Burnett started the first implementation of floating point operations for MIPS. John Hennessy, David Patterson, and Steven Przybylski provided comments and advice on earlier versions of this paper.

References

1. Gill, J., Gross, T., Hennessy, J., Jouppi, N., Przybylski, S., and Rowen, C. Summary of MIPS Instructions. Technical Note 83-237, Stanford University, November, 1983.

2. Hennessy, J.L., Jouppi, N., Baskett, F., and Gill, J. MIPS: A VLSI Processor Architecture. Proc. CMU Conference on VLSI Systems and Computations, October, 1981, pp. 337-346.

3. Hennessy, J.L., Jouppi, N., Baskett, F., Gross, T.R., and Gill, J. Hardware/Software Tradeoffs for Increased Performance. Proc. SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, Palo Alto, March, 1982, pp. 2 - 11.

4. Hough, D. "Applications of the Proposed Standard for Floating-Point Arithmetic". *IEEE Computer 14*, 3 (Mar 1981), 70-74.

5. IEEE Computer Society Working Group. "A Proposed Standard for Binary Floating-Point Arithmetic". *Computer 14*, 3 (March 1981), 51 - 62.

6. Jouppi, N. Multiplication and Division Features in MIPS. Project Report, Stanford University.

7. Levy, H.M., and Eckhouse, R.H.. *Computer Programming and Architecture -- The VAX-11.* Digital Press, 1980.

8. Patterson, D.A. and Ditzel, D.R. "The Case for the Reduced Instruction Set Computer". *Computer Architecture News 8*, 6 (October 1980), 25 - 33.

9. Patterson, D.A. and Sequin, C.H. "A VLSI RISC". *Computer 15*, 9 (September 1982), 8-22.

10. Przybylski, S., Gross, T., Hennessy, J., Jouppi, N. and Rowen, C. "Organization and VLSI Implementation of MIPS". *Journal of VLSI and Computer Systems 1*, 3 (Fall 1984).

11. Przybylski, S. The Design Verification and Testing of MIPS. Proceedings, Conference on Advanced Research in VLSI, Boston, January, 1984, pp. 100-109.

12. Radin, G. The 801 Minicomputer. Proc. SIGARCH/SIG PLAN Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, Palo Alto, March, 1982, pp. 39 - 47.

13. Sippel, T.N. Floating RISCs: Implementation and analysis of floating point on RISC I. Research Project at the University of California, Berkeley.

14. Sweeney, D.W. "An Analysis of Floating Point Addition". *IBM Systems Journal 4*, 1 (1965), 31-42.