A Software-Oriented Floating-Point Format for Enhancing Automotive Control Systems

Yoji Yamada* Daniel A. Connors† Wen-mei W. Hwu†
*Mazda Motor Corporation
†Department of Electrical and Computer Engineering
The Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801
Email: {yamada, dconnors, hwu}@crhc.uiuc.edu

December 8, 1998

In recent years, the software of automotive control systems has become increasingly large and complex to meet numerous requirements for system functionality and performance. As a result, it is very difficult to develop large programs, especially for RISC architectures, in assembly language. Consequently, high level languages (primarily C) and compilers have become more involved in the development of control programs for embedded control systems. One of the advantages of compiler involvement in software development is the ability to initiate aggressive global program-level optimizations. We propose one such optimization for increasing the performance of real number computation in automotive control systems by designing a new software-oriented floating-point format. In our proposed format, the number representation is specifically designed for automotive control programs that use software emulation rather than hardware resources. Since the proposed format is integrated within compiler technology, it offers a seamless method for creating new performance opportunities for an increasing number of embedded systems. We present an analysis of the characteristics of our proposed floating-point format and examine the effectiveness of the format integrated in an existing compiler to improve performance.

In automotive control systems, a large portion of execution time is dominated by real number computations and comparisons. Traditionally, real numbers are incorporated into these systems by either using fixed-point arithmetic or emulating floating-point standards. These methods primarily exist due to the hardware cost limitations on automotive micro-controllers that prevent expensive FPU (Floating Point Units) from being included on the system. The lack of such hardware illustrates the principle way that embedded systems differ from general purpose systems.

In fixed-point arithmetic, integer computations are performed with the convention that the binary point is assumed somewhere other than just to right of the least-significant digit. In this case, real numbers are converted into fixed-point numbers using some rule which is point of all fixed-point numbers in program in order to avoid overflow and underflow. The extra management not only reduces the performance of software but also affects its reliability. For these reasons, a more practical method of including real number computation in automotive systems is to emulate IEEE standard format floating-point representations [1][2]. The IEEE format is very sophisticated and very effective in most applications, especially for numerical computation. However, it is somewhat complicated due to the sophistication. For example, the IEEE standard supports various unusual numbers such as overflow, underflow, an NaN (Not a Number). Similarly, concepts such as rounding, biased exponents, and an implied leading bit in the mantissa require additional handling and

Operation	Cycles
Compare	20
Add/Sub	50
Multiply	80
Divide	130

Table 1: Execution cycles for floating-point primitives.

Operation	Size (Instructions)
Compare	70
Add/Sub	290
Multiply	180
Divide	210

Table 2: Code size for floating-point primitives.

manipulation. For automotive controllers without FPU hardware, the IEEE floating point computation is performed by software emulation. The emulation requires many cycles of processing overhead to carry out the intricate details of the standard. In addition, the emulation code requires many instructions and occupies the instruction memory resource of the control processor. Table 1 and Table 2 respectively show the approximate execution cycles and code size for the floating-point primitives, For example, it takes more than 20 cycles to compare the magnitude of floating-point numbers using the IEEE format and the emulation code is about 70 instructions.

It is important to mention that for embedded systems, there is not a requirement to use the IEEE floating-point format standard. In fact, various fixed-point formats are used in control systems for application-specific domains. One of the reasons for using the IEEE standard is that most compilers have already adopted the representation. Accordingly, our approach investigates using a new software-oriented floating-point format by changing an existing compiler infrastructure. The approach improves the execution performance of real-number processing in a set of automotive control applications and reduces the code size requirements of the floating-point emulation support.

Proposed Software-Oriented Floating-Point Format

The proposed software-oriented approach to floating-point representation allows the compiler to customize the format's range and precision for automotive control systems. At the same time, key format decisions can reduce the computation complexity of real number arithmetic operation emulation. Figure 1 shows an overview of the proposed floating-point format.

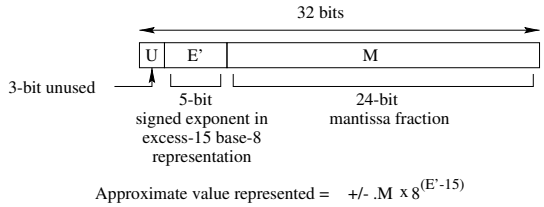

Figure 1: Proposed software-oriented floating-point format.

In short, the format uses a 24-bit mantissa, and a 5-bit signed exponent in excess-15, base-8 representation. Details on the design decisions and modifications to the IEEE floating-point standard of this format are summarized by the following topics:

- Elimination of the NaN, overflow, and underflow representations
- Unifications of positive,negative zero representation
- Use of explicit leading bit for mantissa
- Monotonous increase of number format
- Normalization by multiple bits

The first step of our proposed approach is to simplify the floating-point format based on the analysis of the computational features of control programs. It is rare that special numbers such as NaN and overflow are used for computation in control system. When these special cases happen, it is not necessary to continue the operations using the special representations, but appropriate to instead execute failure-mode routines. Another special representation is underflow. Unlike numerical computation, the precision of numbers near zero is not very significant for most automotive control systems. In this case, underflow can be regarded as zero. These special representations can be omitted to reduce the operation overhead of processing

operands. Another complicated matter is the representation of zero. In the IEEE standard format, zero can be represented in two ways: positive and negative. Although it may be important from the view point of mathematics to distinguish positive zero from negative zero, it is not necessary for automotive control systems. By using a single zero representation, the extra processing of two zero representations becomes unnecessary.

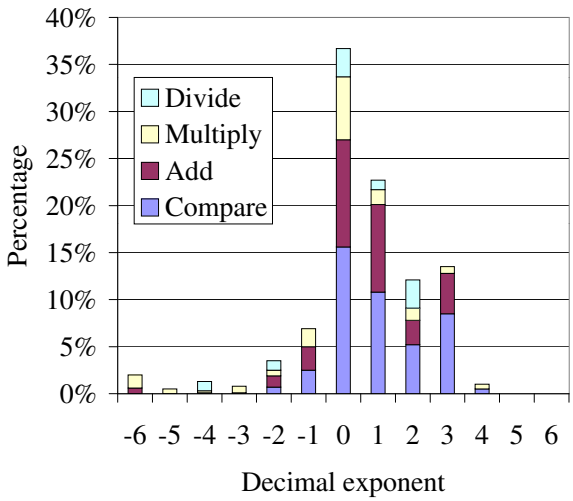

Figure 2: Distribution of decimal exponents.

Another feature of IEEE format is the representation of mantissa. In the IEEE standard, the complete mantissa, called the significand, is actually 1.M, where the 1 to the left of the binary point is an implicit or hidden leading bit which is not stored in the word. Therefore, although the number of mantissa bits for single precision is 23, the precision of the actual mantissa is effectively increased by 1 bit to 24 bits. This is very sophisticated, but complicated for emulation in software. Before any actual operation, the significand of each operand must be recovered by combining the hidden bit and mantissa, and after the operation completes, the leading bit must be removed from the significand of the result. These steps require extra execution cycles, and delay the overall completion of a real number computation. In order to reduce the execution overhead of operations by software, it is better to remove the additional processing steps by using an explicit leading bit rather than implicit leading bit. Using an explicit leading bit for the mantissa causes a one bit reduction of exponent bits. However, this is acceptable for current automotive control systems as illustrated by the Figure 2 which shows the decimal exponents of run-time operands for floating-point primitive operations of an engine control program. Figure 2 indicates that only a few powers of decimal exponents are necessary for automotive systems.

Another part of the IEEE floating-point standard that complicates floating-point operation emulation is normalization. Normalization is done by shifting the mantissa so that the MSB (Most Significant Bit) becomes 1 in order to keep the same precision for all floating-point numbers. Since it is complicated to find the left-most 1 in mantissa by software, normalization takes many execution cycles. For example, five comparisons are necessary to find the left-most 1 in 32-bit mantissa using a binary search algorithm. Another issue due to normalization appears for addition and subtraction operations. It is necessary to equalize the exponents of the two operands for these computations. Therefore, some extra processing to equalize the exponents is necessary. A solution to the issues above is to normalize multiple bits rather than a single bit. It is necessary for this method to represent the MSB of mantissa explicitly. If the normalization is done every multi-bits, the process for normalization can be simplified, reducing the execution overhead of floating point operations. For example, if the mantissa is normalized every 8-bits, only two comparisons are necessary for a 32-bit mantissa. Also, since more numbers will be represented with the same exponent, operands for operations are more likely to have the same exponent, eliminating the extra exponent equalization steps. This was an important reason for using a base-8 exponent that normalized every 3 bits of mantissa.

The software-oriented floating-point format for engine control systems is specified by examining the features of engine control. The main motivation of designing a new format is that floating-point computation and compare operations are a large percentage of execution time, nearly 50%. A series of automotive control programs were examined, and the characteristic frequencies of the floating primitives were determined. The breakdown of the floating-point primitive operation execution is illustrated in Figure 3. The characteristics show that the

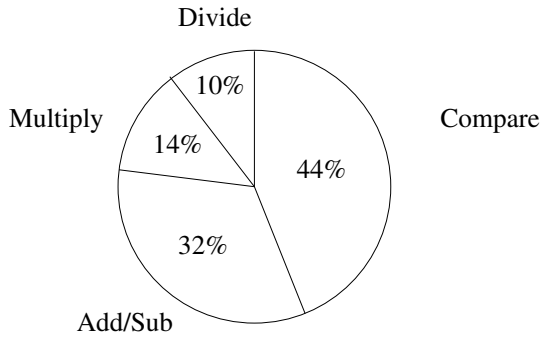

Figure 3: Breakdown of floating-point computation primitive execution.

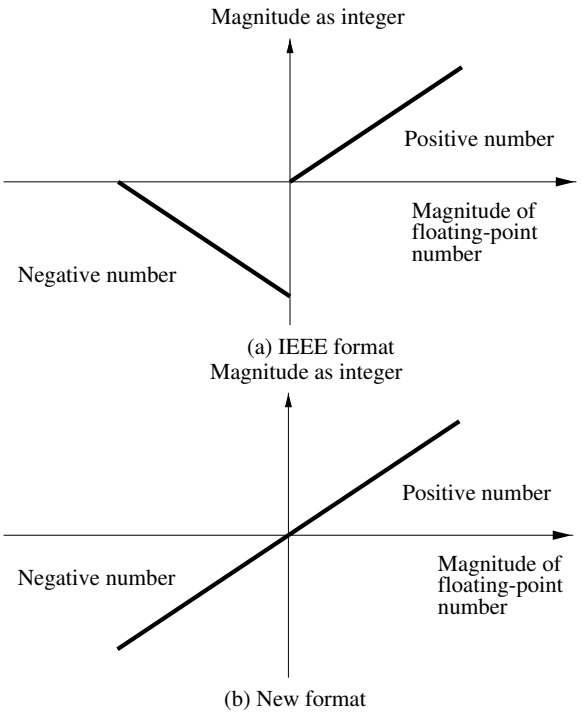

Figure 4: Magnitude of floating-point representations (a) IEEE format, (b) proposed format.

breakdown of floating point operations are: compare (44%), addition/subtraction (32%), multiplication (14%), and divide (10%). As mentioned earlier and illustrated in Figure 3, floating-point compare operations dominate floating-point computation. The current use of array data maps as control parameters in control program account for the large percentage of compare operations. Due to the high execution distribution, the design of the proposed floating-point format concentrated on improving floating-point compare operation execution, and it is important to examine some of the IEEE representation.

The IEEE standard uses a signed magnitude representation. The bits for sign, exponent, and mantissa are assigned in a word in this order. Figure 4a shows the magnitude of floating point numbers treated as integers for the IEEE format. For positive numbers, a floating-point number with larger exponent is always larger than another floating-point number with smaller exponent. For numbers with the same exponent, it is also obvious that a number with larger mantissa is larger than another number with smaller mantissa. Therefore, the comparison of two positive floating-point numbers can be done using an integer compare operation. In other words, the floating-point representation of positive numbers increases monotonously. However, for negative numbers, the magnitude of floating-point numbers is reversed, meaning a monotonous decrease of representation. Therefore, it is necessary to use a different compare algorithm for negative numbers, causing additional processing cycles for compare operations. If the negative representation increases monotonously like positive numbers, it is possible to use the same compare operation for both positive and negative numbers. Taking the 2's complement of the exponent and mantissa bits for negative numbers is one of the implementation for the monotonous increase. The monotonous value of the proposed representation format is illustrated in Figure 4b. Since the new format uses a single zero representation as described previously, the floating-point representation value continuously increases from negative to positive numbers. This means that the compare operation for the new floating-point format can be executed using a single integer compare instruction.

Experimental Methodology and Results

To examine and experiment with the proposed floating-point format, the IMPACT [3] research compiler was altered to encorporate the properties of the new representation. The alteration affects the code generation and high-level to low-level phases of the compiler. Otherwise, the traditional C-language preprocessing, parsing, and internal representation of the compiler were unchanged. The IMPACT compiler has support for traditional opti-

mization, speculation, predication, ILP transformations, scheduling, and register allocation. In addition, other compiler optimizations are implemented specifically for the new floating-point format and floating-point operations. First, since floating-point compare operations for new format can be performed by single integer compare instruction, the function call for a floating-point comparison can be inlined at the call site. This reduces the overhead of function calls and register allocation pressure. Another optimization involves the composition of floating-point operations. For each floating-point operation, the exponent and mantissa are extracted from each source operand, and after the operation the exponent and mantissa of the result must be combined into a operand destination. This overhead can be reduced by combining consecutive operation calls into a single call to a complex computation operation function, which consists of multiple traditional operations. After the first operations is performed, the exponent and mantissa of the result are used for the second operation directly.

In order to show the effectiveness of new floating-point format, preliminary experiments have been performed by examining the speedup of primitive operations. It is desirable to measure the actual execution time of each primitive operation while real control program is running on an actual electronic control unit. However, it is not possible to measure the overall real system performance since device interruptions happen during floating-point operations. Therefore, in the experiment, run-time operands of floating-point numbers have been gathered by an in-circuit-emulator when primitive operations are called in an actual control program running on an real control unit. After converting the numbers into the new format, the test program, which includes new library functions for the new floating-point format operations, was ran for the numbers as the operands on the emulator, and the execution time of each primitive operation was measured. The target architecture studied in these experiments are based on the Hitachi SuperH RISC Engine [4], a RISC style processor with a multiplication unit. It is important to note that the design of the floating-point format meets all of the precision requirements of the automotive programs tested.

Figure 5 shows the speedup of the execution time of operations using the proposed format compared to the IEEE standard format. The compare operation using the new format achieves the highest speedup, approximately 20 times faster than using the IEEE format. The large percentage of compare observed in control programs and the significant speedup for the compare primitive illustrate the benefits of the proposed floating-point format. Also noteworthy is a 3.8 times speedup for addition and subtraction operations that is mainly due to the reduction in the normalization steps due to multi-bit normalization.

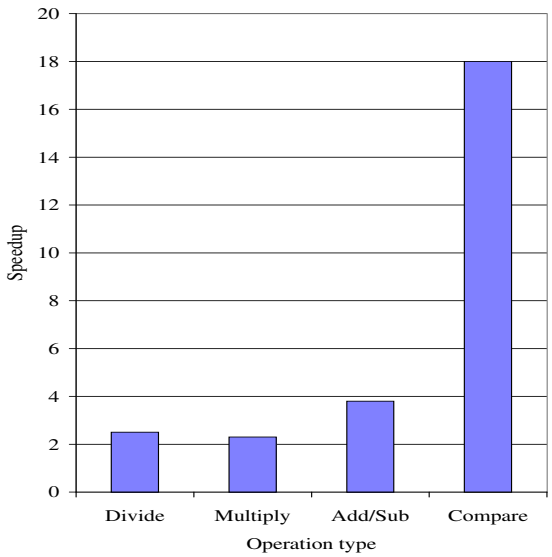

Figure 5: Performance speedup of floating-point primitive.

In addition to floating-point primitive speedup evaluation, the code size reduction between the proposed format and the IEEE format was measured. Figure 6 shows the code size (in bytes) for the software floating-point emulation code. The new comparison, addition/subtraction, multiplication, and division emulation routines experience code reductions of 97%, 52%, 29%, and 42% respectively. Again, the large reduction in the the comparison routine is due to the ability of the new format to use an integer comparison operation in place of an emulation routine.

The overall profile-estimated speedup for an engine control program was 2.1 times the base performance model using the IEEE format. This speedup did not include savings from the elimination of spill code and subroutine call convention operations. Future experiments

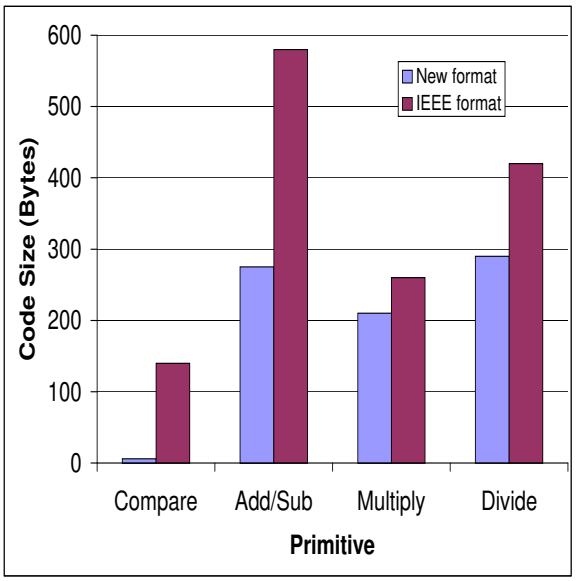

Figure 6: Code size of floating-point emulation code.

will include overall real-time program performance measurements by examining the execution time for a variety of engine control programs.

Summary

A new software-oriented floating-point format for embedded control systems that requires no special floating-point hardware for operations has been proposed to improve the execution of floating-point operations. The IEEE floating-point standard has been simplified for the needs of automotive control systems. Simplifications such as using an explicit leading bit of mantissa rather than the hidden bit of the IEEE standard improve the performance of primitive operations in control programs. Additionally the monotonous increase of new format makes it possible to execute a compare operation for floating-point numbers using single integer compare instruction. Normalization by multi-bits improves the execution speed of operations, especially for addition and subtract operation. This proposed technique enables the reliable and efficient development of programs in C-language for enhancing the performance of cost-sensitive control systems.

References

[1] IEEE, "Ieee standard for binary floating-point arithmetic," *SIGPLAN Notices*, no. 2, pp. 9–25, 1985.

[2] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufman, 1996.

[3] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[4] Hitachi Company, Tokyo, Japan, *SuperH Architecture and Instruction Set Reference Manual*, 1995.