RC 10524 (#45515) 10/28/83
Computer Science 9 pages

Research Report

Multiplication by Integer Constants

R. Bernstein
Victor S. Miller

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

IBM Research Division
San Jose · Yorktown · Zurich

RC 10524 (#45515) 10/28/83
Computer Science 9 pages

Multiplication by Integer Constants

R. Bernstein
Victor S. Miller

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract: In this paper we consider the problem an optimizing compiler, such as the PL.8 compiler, faces when generating a sequence of shifts, adds, and subtracts to multiply the contents of a register by an integer constant.

Introduction.

In this paper we consider the problem an optimizing compiler faces when generating a sequence of shifts, adds, and subtracts to multiply the contents of a register by an integer constant. Multiplication by integer constants is commonly required, for example, to compute offsets into arrays. Why not use the hardware multiply? On some microcomputers and reduced-instruction-set computers there is no full-word multiply instruction since this function is expensive in silicon. On those machines which do have a hardware multiply it is typically a slow instruction, comparable to several "shift" or "add" instructions on the same machine. Although we assume a machine capable of executing "shift left n positions," "add," and "subtract" instructions, our results can be easily modified for machines with a more restricted instruction set (such as being able to shift left only by one) and our approach has been used on a machine with the more exotic instruction "shift left one and add".

There are two facets to this problem. First, we would like to produce "good" code, which involves the not mutually exclusive considerations of using a small number of registers, using a small number of instructions, or using a fast-executing instruction sequence. Second, suspecting that this problem NP-Complete (it resembles a knapsack problem), we want an algorithm that does not require an inordinate amount of time or space. We are willing to trade search time for code quality when the gains are small, or when gains occur for only a small percent of the cases.

Logarithms will always be given in base 2; *n* will always denote the known multiplier (an integer constant). The multiplicand, assumed to be in a register, will always be denoted R001; other partial or final products will be denoted as the letter "R" followed by the three digit number of the product computed. We will usually confuse instruction sequence length with the time to execute such a sequence, tacitly assuming that the time required to shift, add, and subtract in registers is approximately the same. On a computer such as the 801 [Ra82] this is the case, and it is more or less true on many other machines as well.

The binary method.

A simple way to multiply a register by an integer is to write the integer in binary; all except one of the 1's in the binary number then turn into a shift and an add of the final code sequence. For example, to multiply R001 by 113 = 1110001_2 we might use:

```
R002 ← R001 shift 1      or:    R016 ← R001 shift 4
R003 ← R002 + R001              R032 ← R016 shift 1
R006 ← R003 shift 1             R064 ← R032 shift 1
R007 ← R006 + R001              R065 ← R064 + R001
R112 ← R007 shift 4             R097 ← R065 + R032
R113 ← R112 + R001              R113 ← R097 + R016
```

Many other sequences are possible using the binary method. The above two sequences have the property that the result of each computation is consumed immediately in the following instruction. Knuth [Kn81] calls this sequence a *star-chain sequence*. Star-chain sequences are good because operations can be done naturally on a two-address machine. (A two-address machine performs operations of the form: x ← x *op* y.) However a copy of R001 must be saved. The first sequence given above, which has alternating shifts and adds, may be desirable for another reason: only two different registers are needed to compute the product, one to hold the partial products R002, R003, R006, R007, R112, R113, and one to hold R001.

We may be able to reduce the number of instructions if we allow subtraction to enter the sequence. Long runs of consecutive 1's within the binary representation can be carried out by shifting left by the length of the run and then subtracting. In the previous example we note that the three adjacent 1 bits can be replaced by a shifting left three and subtracting, saving two instructions to compute R007. The star-sequence property can still be maintained, and two registers on a two-address machine still suffice. We can still arrange the sequence so that there is a shift every other instruction. An $O(\log n)$-time algorithm for generating this kind of code should be obvious.

Factoring.

Knuth suggests that by using the factors of the number n, one may be able to shorten the instruction sequence. For example, to multiply R001 by 119 we might write this as (R001 × 7) × 17. Since both 7 and 17 are prime we use the binary method to compute these yielding the four instructions:

```
R008 ← R001 shift 3
R007 ← R008 − R001
R112 ← R007 shift 4
R119 ← R112 + R007
```

A pure binary method would require six instructions. Note again that we can arrange the factoring so that it maintains a star-chain property. Knuth gives a method for constructing a table of small star-chain sequences for all numbers up to some desired limit. We decided not to use a table built into the compiler because of the space it would require for the range of numbers that might appear in compilations. It seemed unwise to burden a compiler with such a large table when only a few multiplications usually occur in a program. Furthermore, different tables might be needed for different target machines. Rather we decided to grow a table containing the multiplications as they are encountered for the first time in the program and as they are needed as intermediate results. The table is sensitive to the instruction set and instruction set costs of the target machine.

The basic algorithm.

One of us (Miller) empirically discovered that many of the factors which lead to short instruction sequences are the ones which can be computed in only a few instructions, say less than 3. Using the instruction repertoire stated in the introduction, these factors are a power of 2 (as is exploited in the binary method), or a power of 2 plus or minus 1. This leads to the following formula:

$$Cost(1) \quad = 0 \text{ — } \textit{no instructions are needed to multiply by 1}$$

$$\begin{aligned} Cost(even\_n) &= Cost(odd\_n) + \textit{cost of shifting by } i \\ &\quad \textit{where } even\_n = odd\_n \times 2^i \end{aligned}$$

$$\begin{aligned} Cost(odd\_n) = \; & Min\,(Cost(odd\_n - 1) && + \textit{addition cost}, \\ & \quad Cost(odd\_n + 1) && + \textit{subtraction cost}, \\ & \quad Cost(odd\_n \,/\, (2^i - 1)) && + \textit{cost of shifting by } i \textit{ and adding}, \\ & \quad Cost(odd\_n \,/\, (2^i + 1)) && + \textit{cost of shifting by } i \textit{ and subtracting}\;) \\ & \textit{where } odd\_n \,/\, (2^i \pm 1) \textit{ is integral} \end{aligned}$$

Note that if the last two lines of the above Min function are removed, we get exactly the binary method. Also note that even numbers are shifted right until they become odd, as in the binary method. One can always arrange a sequence found by the above so that on a 2-address machine no more than 3 registers will be needed simultaneously. Like the binary method, one register is needed to hold a copy of the register multiplicand R001, and one register to hold the partial or final product. But a third register may be required to hold the last factor computed. In the example given in the previous section, a third register would be needed to hold R007 if R001 were needed again. This would occur if we were computing R001 × 120 by adding R001 to R119.

A straightforward implementation of the preceding formula is slow. In fact, one can get into an infinite recursion if one is not careful about how a program carries out the equations. Two techniques are used to speed things up. First, we save all final and intermediate products that get computed, and hash further requests for the results. Only when the values are not found do we need to compute them from scratch. Second, we use search cutoff bounds.

Hashing.

We use a universal hash [CW79] on odd values, saving sequences for these only. If one needs to save table space, the following proof by induction shows that not much is lost by not storing negative values.

Theorem: The minimum number of instructions needed for multiplication by a negative number is not less than the number needed for multiplication by the corresponding positive number.

Basis: Multiplication by 1 requires no instructions and uses one fewer instruction than to multiply by -1 (using a complement, or a subtract from zero).

Inductive hypothesis: Assume true for positive numbers less than n. We show by contradiction that multiplying by n requires no more instructions than for multiplying by $-n$. Assume not, and consider the last instruction in the sequence for multiplication by $-n$. If it is a subtraction, the operands can be reversed so that $+n$ requires exactly the same number of instructions as $-n$. If it is a shift left then the product computed in the next-to-last instruction in absolute value is less than n and the inductive hypothesis asserts that no more instructions are needed to compute the absolute value of this product. The only remaining possibility is that the last instruction is an addition. But then both operands of the addition must be negative (if the signs were mixed, we then have the subtraction case handled above, and the sum of two positives is positive). Thus even here we can invoke the inductive hypothesis to achieve a contradiction. Q.E.D.

Thus we can construct an instruction sequence to multiply by a negative number which is at most one instruction longer (namely a complement instruction) than the optimum sequence, provided we have the optimum sequence for the positive value. If the last non-shift instruction of the sequence for the positive value is subtraction, the negative value can be easily obtained by reversing the order of the input operands.

The decision not store even products in the table was not so much to save table space, or to hash more values to the same table entry, but to save searching time. As will be shown later, our search-time bound is dependent on even intermediate terms being shifted right until becoming odd. We can show however that this means longer sequences are generated in certain cases. For example, under the given formula multiplication by 154 requires two fewer instructions if it is calculated as $155 - 1$ than if it is calculated as 77×2. Many of the cases occur when the shift to the odd number is small and there is a factor one greater or less than *even_n*. It is easy to augment the definition for *even_n* to search for such cases.

Search-cutoff heuristics.

We achieve another speed up by bounding the cost of the sequence of instructions needed, stopping early in our search when we exceed the bound. The upper bound for the binary method given earlier (twice the number of 1's in the binary representation minus 1) can be computed easily before searching for any sub-sequence. As indicated above, the formula insures that we find a sequences and sub-sequences at least as good as the binary method. Another bound which we use is the cost to perform a general multiplication through a hardware multiply or through a call to a multiply subroutine. As candidate sequences are found, this upper bound may decrease further and restrict the search more. We also keep a lower bound for numbers (possibly encountered as intermediate results) which have not been computed exactly because of exceeding the bound requirement. This lower bound plus the cost of the partial sequence found so far must then be less than the least upper bound found so far if the search to complete the sequence may yield better results.

An example might help to clarify the above. Suppose we want to find a sequence for $155 = 10011011_2$. The binary method estimate is 8 instructions. Dividing by the two-instruction factor $31 = 2^5 - 1$, we get 5. Then we search for a sequence which will multiply by 5 in at *most* $8-2 = 6$ instructions. But the binary estimate for 5 is 2 instructions, so we tighten the amount of further searching we are willing to do. Using the binary method for 5, we find the 4-instruction sequence:

```
R004 ← R001 shift 2
R005 ← R004 + R001
R160 ← R005 shift 5
R155 ← R160 - R005
```

We continue searching from 155 by subtracting 1, and then dividing by 2. We are looking for the optimal sequence which we can multiply by 77 using at most $4-2 = 2$ instructions. We will not find such a sequence, but we can still record the fact that 77 will require at *least* 2 instructions. If on some future search we are given the task of multiplying by 77 in not more than 2 instructions, we can signal failure immediately without further search.

Time complexity.

In the following discussion we will assume $c \log n << k$, where c is the average time to perform an add, subtract, or shift instruction and k is the time required to perform the multiply in hardware, or the time to perform a general multiplication subroutine. To obtain an upper bound for the search time of our basic algorithm, consider all possible odd intermediate products in an instruction sequence. Recall that an instruction sequence alternates between shift and add/subtract instructions. For intermediate product i with odd value, there are at most 2 ($\log n - \log i$) odd products following i to which we can shift and then add/subtract R001, and same number to which we can shift and then add/subtract factor i. Thus each i has at most 4 ($\log n - \log i$) successors. Log i time is needed generate factors of the form: $2^i \pm 1$. Thus the total time required is:

$$\sum_{i=1}^{n/2} 4\,(\log n - \log i) \log i \leq O(n \log n)$$

This is a very crude and pessimistic upper bound since it does not take into account the effects of either of the speed up heuristics described earlier.

Data structure note.

Because all star sequences we generate alternate between shifts and adds/subtracts, a tree represents the set of multiplication sequences concisely. Each node contains the value of some integer multiplier, and the parent of the node is the product computed two instructions before. Each edge has labels "shift left i", and "add" or "subtract" using either R001 or the last factor. The labeling of edges on the path from the root with value 1 to a node with value n shows the sequence of instructions needed to multiply by n.

As a final aside, we mention that transformations such as the one described here make compilers produce fast-running code, but are hard to capture in a simple macro-processing system which most table-driven code generating systems provide.

Acknowledgements

We would like to thank members of the PL.8 Compiler group, Greg Chaitin, Dick Goldberg, Marty Hopkins, Peter Markstein, and Hank Warren for their criticism and encouragement. Peter Markstein first described the proof of the theorem.

References

[CW79] J. L. Carter, M. N. Wegman, "Universal classes of hash functions," *J. Comput. Sys. Sci.* 18 (1979), 143-154.

[Kn81] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms* , Addison-Wesley, 1981, Reading, Mass, 441-462.

[Ra82] G. Radin, "The 801 minicomputer," *Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems,* March 1-3, 1982, Palo Alto, California, 39-47.