The compiled macro assembler

by WARD DOUGLAS MAURER

University of California
Berkeley, California

INTRODUCTION

This paper describes an advance in the art of writing assemblers. It embodies an idea which has been suggested at least twice, but never actually implemented. In a compiled macro assembler, ordinary source language statements are processed in the usual way, but macros are processed in a novel way. The advantage of the compiled macro assembler is the speed with which it processes macros. An actual compiled macro assembler has been written by the author and his students, and the speed with which it processes macros, as distinguished from ordinary statements, has been rigorously tested.

The basic concept of the compiled macro assembler

We review, first of all, the operation of an ordinary assembler, which we will refer to, in what follows, as an interpreted macro assembler. (The words "compiled" and "interpreted" are presumed to modify the noun "macro," not the noun "assembler." Each pseudo-operation code in the assembly language recognized by a given assembler corresponds to a subroutine of that assembler. This subroutine is called whenever the given pseudo-operation is encountered within the source text. The collection of all of these subroutines, for a given assembler, is a fixed collection, and on a large computer this collection of subroutines is normally contained in core at all times. On a small computer, the subroutine which corresponds to a given pseudo-operation may have to be brought in from disk when the pseudo-operation is encountered; however, the total collection of subroutines corresponding to pseudo-operations remains fixed.

A macro is, in one sense, very much like a pseudo-operation. However, in an interpreted macro assembler, the occurrence of a macro does not set aside a special subroutine of the assembler for the use of that macro alone. Instead, all macro definitions are treated in the same way. The text of a macro definition is copied into memory, after various minor transformations such as the removal of blanks. In some assemblers, the information contained in a macro may be further compressed, but in an interpreted macro assembler the compression is done in an essentially recoverable way if it is done at all. When the macro is used, this text is read from memory in what may be called an interpretive fashion—although there is no separate interpreter, the entire assembler itself serving as the macro interpreter.

In a compiled macro assembler, all pseudo-operations—macros as well as others—have their corresponding subroutines of the assembler. At the start of each assembly there exists a fixed collection of such subroutines. However, when a macro is defined, a new subroutine is formed. This subroutine is compiled (hence the name, compiled macro assembler) from "source text" consisting of the original macro definition. The writing of a compiled macro assembler consists in the mechanization of the process of deducing, from the form of a given macro definition, how a use of this macro would be treated within the assembler if it were a pseudo-operation rather than a macro.

As an illustration of the concept of macro compilation, an actual compiled macro assembler was constructed by the author and his students.* This assembler is written to run on the CDC 6400. The input language is a modified form of IBM 360 assembly language; the output from the assembler is a listing of the IBM 360 code generated, and a deck of binary cards which will execute on the 360 when appropriate control cards are added.

* The students included Donald Alpert, Steven Anderson, Robert Ankerlin, Thomas Baumbach, David Brown, Dennis Griswold, Bing Joe, Richard Kayfes, David Ladd, Kenneth Lew, William Nielsen, Ralph Olstad, Paul Samson, and Edmond Van Doren.
Feasibility of macro compilation

The following paragraphs are devoted to certain feasibility considerations which the author and his students discovered in the course of writing this assembler. These points should be thoroughly understood by anyone intending to write such an assembler in the future.

Substitution of parameters

There are two common methods of handling macro parameters in an assembler. These are known as string substitution and value substitution. Either may be used in a compiled macro assembler. In addition, if value substitution is used, compilation may be carried out completely; whereas if string substitution is used, it is necessary to include both compiled and interpreted macro facilities, and it may be necessary for a compiled subroutine to call the interpretive facility.

For the sake of completeness, we now describe these two methods in general terms. In value substitution, each actual parameter in a macro usage is evaluated. This value is substituted within the macro text whenever the corresponding formal parameter is encountered. In string substitution, the character string which comprises a given actual parameter in a macro usage is copied into memory when the macro usage is encountered. If the assembler is an interpreted macro assembler, the source of input characters to it is now diverted to the location of the macro text in memory. When a parameter is encountered, the source of input characters is re-diverted to the location of the character string giving the corresponding actual parameter.

String substitution is more general than value substitution because the sequence of input characters passes freely between the characters of the macro and the characters of actual parameters. Thus syntactic units may exist partially within the macro text and partially within the parameter. One important use of this facility is the appending of prefixes or suffixes to an actual parameter to form symbols. If a macro is called with actual parameter DM, for example, the macro may then create symbols DMA, DMB, DM1, TEMPDM, and the like, and use them in an arbitrary fashion. Such symbols, of course, become global, and may be referenced throughout the text. In a value substitution assembler, this facility is not possible; but in many value substitution assemblers a symbol defined in a macro cannot be used outside the macro unless it is specially declared to be global. Thus the same symbol may be used over and over again, so long as it is always used inside a macro and only once inside each distinct usage of that macro.

String substitution has been used in most assemblers which have appeared in published work, such as Halpern's XPOP, Strachey's general purpose macro generator, and Mooers' TRAC. Value substitution, however, because it is simpler, has been used in many actual, working assemblers. Among these are the FAP assembler for the IBM 7094, the SLEUTH II assembler for the UNIVAC 1107, and an assembler for the UNIVAC III, all of which were written by Ferguson, who, so far as we know, has published only one account of his work.

Let us now consider the substitution of parameters in a compiled macro assembler. If value substitution is used, there is no problem. Suppose that a parameter usage is found within a macro definition. Corresponding to this usage in the compiled subroutine, there is a call to a subroutine which retrieves the value of the corresponding actual parameter. (That is, the compiled subroutine, which is produced by the macro compilation process, calls a fixed, special assembler subroutine, whose function it is to retrieve parameter values.)

If string substitution is used, we make a distinction between a parameter which occurs in an expression in the variable field, and one which occurs by itself in the variable field. (Most actual parameters are of the latter kind, because most people write relatively simple macros.) If a parameter occurs by itself, there is no difference, for this parameter, between string and value substitution, and it may be handled as described above. If a parameter occurs in an expression, however, it is generally impossible to handle it in a compiled manner. The text of the expression must be included with the compiled subroutine, and, at the appropriate point, this subroutine calls a fixed, special assembler subroutine whose function it is to interpretively evaluate such strings. As in the case of an interpreted macro assembler, this "subroutine" consists, from the logical point of view, of the entire assembler itself.

First and second pass compilation

The compilation process, as applied to a macro, must take place twice—once in the first pass and once in the second pass.

There are many reasons for this; the following is perhaps the simplest. Suppose that the definition of a macro involves a symbol which is not defined until after the macro is defined. Then, when the macro is first encountered, complete compilation cannot take place, since the value of the symbol is not known at that time. Therefore the macro must be compiled in the second pass. But it must also be compiled in
the first pass, since the length of the generated code is not known, and different uses of the same macro may result in different lengths of generated code. The main function of the subroutine which is compiled in the first pass, in fact, is to determine this length; at the same time, any global symbols defined within the macro are placed in the symbol table along with their addresses.

It might appear at first sight that this problem could be avoided by defining all symbols used in a macro before the macro is defined. However, this is not feasible in general. A macro may contain a call to an error routine which is at the end of the program, or, in general, which follows another usage of the macro. It is, in general, true that all symbols occurring within a macro definition which affect the length of the generated code must be defined before the macro is defined. By somewhat devious methods this may be improved slightly to read “before the macro is first used.”

**Saving a compiled subroutine**

One of the theoretical advantages in compiling macros is that the resulting compiled code can, in theory, be output to cards, in the same way that output from a FORTRAN compiler can be output to cards. These binary cards may then take the place of the original macro definition.

We have found that compiled subroutines can, in fact, be saved in most cases. There is one case, however, that creates several difficulties. Suppose that a macro definition contains a symbol which is used but not defined. Presumably such a symbol would be defined in the body of the assembly language text. (In our experience, most macros do not have this characteristic; but some do, and in any event it would be unwise to exclude it.) The definition of the given symbol in the program in which it is defined is not, however, necessarily the same as its definition in the program in which the binary cards are used. It is this latter definition, in fact, which should apply. Therefore, a distinction must be made when compiling a macro between symbols defined in the macro and symbols defined outside it. There are further difficulties concerned with optimization of the compiled code. If the value of a symbol is known at compilation time, it may be combined with others in an expression, and the value of the result used within the compiled code. If code is being compiled for later use, however, such combination cannot be made. This means that either the resulting compiled code must calculate values of expressions which would not be necessary if the macro were being compiled in that assembly, or the process of loading the binary cards must effectively incorporate some of the compilation process.

Only the second pass compilation need be saved on cards. When this is loaded during the first pass of another assembly, it is loaded in a special way which causes it to act like a first pass compilation.

**Compiled macros and conditional and iterative assembly**

Conditional statements in assembly language may be compiled; so may iteration statements. In fact, compilation of these statements is the primary justification for compiled macro assembly. A conditional statement in the definition of a macro may be replaced by a conditional transfer in the compiled subroutine; it is no longer necessary to read a number of characters without processing them if the condition is not fulfilled. An iterative (duplication) statement may be replaced by a loop in the compiled code; it is no longer necessary to interpret the iterated statements repeatedly.

A macro which is to be used only once, and which contains no conditional or iterative statements, should not, in fact, be compiled. This is a special case of a general statement which may be made about interpretation/compilation situations: compilation is faster than interpretation only if no recycling takes place. If every statement in a program is to be executed at most once, it is cheaper to interpret each statement once than to compile it (which itself involves interpreting each statement once) and then to execute it. The time saving that results from compiling is due to the fact that if a statement is to be executed several times, it will be interpreted several times if the program containing it is interpreted, but only once if that program is compiled.

A macro without conditional or iterative statements *may* be speeded up on compilation if it is to be used several times, but an intelligent judgment should be made in each such case.

**Timing tests of the compiled macro assembler**

In order to verify the premise that compiling macros improves the efficiency of macro usage processing, a controlled experiment was performed on the compiled macro assembler written by the author and his students, with the standard IBM 360 F level assembler serving as the control.

---

* The SLEUTH II assembler embodies an interesting exception to this. If a given macro always generates the same amount of code, this amount may be specified when the macro is defined. Presumably this feature could be implemented in a compiled macro assembler, removing the necessity for compiling such macros on the first pass. However, as we shall see later, such a macro probably should not be compiled anyway.
Timing comparisons of systems designed in different ways to do the same job has proved to be one of the most frustrating tasks in the computing world today. For almost every comparison which has been performed, a perfectly valid argument may be advanced which nullifies its conclusion. Usually this argument takes the form that the observed differences in timing were caused by something other than the differences in the initial conditions. The use of a controlled experiment, a technique borrowed from classical scientific method, is precisely the way in which the effects of such irrelevant factors may be eliminated. In the present situation, the following were the factors which introduced differences in timing comparable to, and sometimes exceeding, the claimed improvements in efficiency:

1. The time taken to process a macro was smaller than the time taken to read a card.
2. The time taken to process a macro was smaller than the time taken to print a line.
3. The total time taken to process a job differed depending on when the job was submitted; in fact, it sometimes happened that when the computer was asked to perform the same job twice in a row (by submitting an input deck consisting of two identical copies of a job deck) the job times differed by a factor exceeding 1.5.
4. The IBM 360 F level assembler as used at the computer center at which the test was made is slower than the Compiled Macro Assembler, by a factor which may exceed 10.
5. The IBM 360 F level assembler is not used at its own greatest efficiency by the computer center at which the test was made.

The controlled experiment was set up in the following way. A macro, RPD3, which generates code to calculate the value of a real polynomial of degree less than or equal to 3, was written for both the Compiled Macro Assembler and the IBM 360 F level assembler. The macro was called, in either assembler, by the line

RPD3 X,A,B,C,D

where X, A, B, C, and D represent addresses in memory and A + BX + CX² + DX³ is the polynomial to be evaluated. The algorithm always uses the fastest computational method; if all of the coefficients are non-zero, then A + X*(B + X*(C + X*D)) is calculated, but if any of the coefficients are zero, a smaller amount of calculation is performed. If all the coefficients are zero, the result register is loaded with zero. Otherwise, the total number of instructions generated is equal to the total number of non-zero coefficients plus the degree of the largest such coefficient.

A deck was now made up, containing 200 calls to this macro with various parameters. This deck was assembled to obtain a printout of the code it generated. A second deck was now made up which consisted precisely of this generated code. Assembly of these two decks, then, should produce identical results in different ways—with and without macro usage processing. To counteract the effect of factor (1) above, a second macro, called NIL, was written, which does nothing. The text of NIL was added to the first deck, and exactly enough usages of NIL were added to the first deck to equalize the number of cards in the two decks. To be absolutely precise, there were now four decks, because all of the above was done twice, once for each assembler.

To counteract the effect of factor (2) above, all assemblies, on both assemblers, were run with a “no list” option during the timing test, after it had been ascertained that they generated correct code. The use of this option ensures that no printing will occur during the second pass of assembly. To counteract the effect of factor (3) above, each of these four decks was reproduced several times, and the resulting copies of each deck were run as a connected series of jobs.

The results of the timing test were as follows. For the IBM 360 assembler, the runs without macro calling took 3 min. 21.94 sec., 3 min. 39.92 sec., 4 min. 25.87 sec., and 3 min. 29.00 sec. The runs with macro calling took 9 min. 33.90 sec., 7 min. 52.06 sec., and 7 min. 56.28 sec. Even with the large experimental error, it is clear that this assembler is taking over twice as long to process an assembly with macros as without macros. For the Compiled Macro Assembler, the runs without macro calling took 16.433 seconds and 16.428 seconds; the runs with macro calling took 16.458 seconds and 16.533 seconds. Thus there is no appreciable difference, in the compiled macro assembler, between assembly of macros and assembly of the identical code without macros.

The presentation of the results in this form counteracts factors (4) and (5) above. In particular, any avoidable inefficiencies which affected the timing of one of the IBM 360 runs would also have affected the timing of the other. We also note that factors (1) and (2) do not, as has been claimed, remove entirely the timing advantage of compiling macros, since on a time-shared computer the time taken to process a macro will usually be smaller than the time taken to read a card image from a file. It is also true that time-sharing systems increase the viability of assembly language coding as opposed to coding in a higher-level language, since debugging languages (such as DDT and FAPDBG) are much more amenable to machine language than they are to higher level language coding.
ACKNOWLEDGMENTS

The author is grateful for the programming help of the students mentioned in the first footnote to this paper. This research was partially supported by National Science Foundation Grant G-443 and Joint Services Electronics Program Grant AFOSR-88-1488.

REFERENCES

1 S ANDERSON
   Master's report University of California Berkeley January
   1968

2 D GRISWOLD
   Object deck output from a compiled macro assembler
   Master's report Univ of California Berkeley September 1967

3 R KAYFES
   Decimal arithmetic in a compiled macro assembler
   Master's report Univ of California Berkeley June 1967

4 K M LEW
   Non-decimal arithmetic in a compiled macro assembler
   Master's report University of California Berkeley June 1967

5 W C NIELSEN
   Subsystem implementation of a compiled macro assembler
   Master's report University of California Berkeley June 1967

6 E D VAN DOREN
   The literal facility and end card implementation of a
   compiled macro assembler
   Master's report Univ of California Berkeley September 1967

7 M HALPERN
   XPOP: a meta-language without meta-physics
   Proc F J C C 1964

8 C STRACHEY
   A general purpose macro generator
   Computer Journal October 1965

9 C MOOERS
   TRAC, a procedure-describing language for the
   reactive typewriter
   Communications of the Assoc for Computing Machinery
   March 1966

10 C MOOERS
    TRAC, a text-handling language
    Proc 20th National ACM Conference 1965

11 D FERGUSON
   Evolution of the meta-assembly program
   Communications of the Assoc for Computing Machinery
   March 1966