Macro-Based Cross Assemblers

KAREL R. TAVERNIER AND PAUL H. NOTREDAME

Abstract—**The problem of implementing cross assemblers by means of a macro expansion technique is addressed. Various problems caused by the implementation technique proper, as well as the target machine instruction set, are identified.**

A systematic solution to these problems is presented in the form of a set of multilevel macro definitions and expansions. The solution is general in that assemblers can be generated for a wide range of target machines.

A practical example illustrates the developed technique.

Index Terms—**Addressing modes, cross assembler, imbedded macro calls, imbedded macro definitions, macro assembler, macros, meta assembler, microprocessor.**

I. Introduction

WITH THE advent of a cheap hardware building block—the microprocessor—the need arises for economical support software. The great diversity in existing microprocessor architectures prohibits the universal use of any simple software product.

This paper considers the systematic construction of cross assemblers for microprocessors using a macro expansion technique. Macro expansion has been widely used for extending the assembly language of a computer, or even to process a language that is completely different from "native" assembly language. An extreme example of this is the use of macros to describe the syntax of a language and to implement the corresponding parser [6]. It is a known fact that a cross assembler can be implemented using a macro expansion technique [1], [2].

In most assemblers, only a small fraction of the code is devoted to translation of the assembly language into object form. The largest part of an assembler is machine language, or better, instruction set independent. Indeed, such important assembler tasks as file I/O, symbol manipulation, and processing of directives or "pseudoinstructions," and accomplishing, e.g., storage allocation, symbol and constant definition, operating mode setup, conditional assembly, and macro expansions, do not depend on the instruction set proper. A macro-based cross assembler makes use of this fact by translating target machine instructions into assembler directives, which are then normally processed by the assembler. In principle, this translation requires nothing more than a predefined macro for each instruction of the target instruction set. For the host assembler, all instructions in target machine language are macro calls and are treated accordingly. The resulting assembler directives generate the target machine object code. This is an attractive way to acquire a cross assembler if a host computer is available. In comparison with an assembler running on a development system, the cross assembler can take advantage of facilities of the host system, such as the fast peripherals and mass storage, the sophisticated file system, multiuser capabilities, etc. Besides its implementation speed, macro-based cross assembly has a number of other advantages: cross assemblers for different target microprocessors, all running on the same host assembler, are very similar. All directives and operating procedures are identical. For design engineers this can greatly reduce the effort of switching to another type of microprocessor. A macro-based cross assembler inherits many features of the host assembler. A powerful host assembler results in a powerful cross assembler without any special programming effort. Some of the features often available on minis and larger machines are the following:

macro capabilities
conditional assembly
assembly time variables and expressions
a cross reference processor
error diagnostics
an alphabetically listed symbol table
program sectioning directives
data storage and location counter directives
radix control.

It should be noted that these features can reduce programming time, errors, and cost, and they improve program reliability. Existing cross assemblers rarely provide these features simultaneously and in the same degree [3].

A disadvantage of macro-based cross assembly is its slow execution, which can be several times slower than a direct assembler. Besides this speed drawback, a variety of problems are associated with the construction of a macro-based cross assembler. These problems arise as a result of specific features both of the host and the target. The purpose of this article is twofold. In the first place, various problems specific to the implementation technique proper or caused by peculiarities of the instruction set of the target microprocessor will be identified (Section II). Our second goal is to discuss solutions and to present a systematic implementation methodology suitable to a wide range of target microprocessors (Section III). Finally, in Section IV, features of the host assembler identified as being necessary or desirable for a successful implementation are listed.

Manuscript received August 8, 1978; revised January 26, 1980.

The authors are with the Interfacultair Centrum Informatica, Laboratorium voor Electronica en Meettechniek, Rijksuniversiteit Gent, Ghent, Belgium.

0098-5589/80/0700-0334$00.75

II. The Major Implementation Problems

These problems fall into three categories:

1) method related problems
2) host related problems
3) target related problems.

A. Method Related Problems

1) The Parsing Problem: We start our discussion with the problems caused by the choice of macro expansion as an assembler implementation tool. The fact that macro expansion is used can make it very difficult to perform a detailed parsing of the source text. Such parsing can be necessary to recognize various constructs, for instance addressing modes written in a standard notation.

For example, consider the MOS TECH 6500 series instruction LDA (load accumulator) with the following addressing modes:

1) LDA #ARG (immediate)
2) LDA ARG (direct)
3) LDA ARG, X (indexed with X)
4) LDA ARG, Y (indexed with Y)
5) LDA (ARG, X) (indirect with preindex X)
6) LDA (ARG) ,Y (indirect with postindex Y).

A macro assembler usually recognizes only a limited number of characters such as blanks and commas as separators. Unless character by character examination is provided, it is extremely difficult to decompose a construct as "(ARG)" in the sequence "(", "ARG", ")". Consequently, parsing the above notation is often impossible and certainly slow with a macro facility.

2) The Listing Problem: Another problem is associated with assembler output. The generation of an aesthetic program listing and good error diagnostics is a major function of any assembler. We do not want to distract the programmer by cluttering the listing with the expansion of macros he has not written. The programmer should be unaware of the fact that the assembler for the microprocessor consists of macro calls to a host assembler.

Providing such an aesthetic listing is a major difficulty in implementing a macro-based cross assembler. To the programmer, the instructions written down are elementary, but to the host assembler these instructions are ordinary macro calls. We want these macro calls to be treated in a special way their expansion must never be listed, except perhaps for the generated machine code, which must be listed at the programmers request.

B. Host Related Problems

The fact that an existing assembler is used can introduce compatibility problems.

1) Unavailable Number Radices: For example, the host computer, and its assembler, may use octal notation, while the target computer is hexadecimally oriented.

2) Memory Organization: A more essential compatibility problem concerns the smallest addressable unit in memory. It is very difficult to cross assemble code on a host whose smallest addressable unit in memory is larger than that of the target machine. Suppose we want to write a cross assembler for the HP2100, a 16-bit processor with word addressing using the PDP-11 as a host. With word addressing we mean that the smallest addressable unit is a 16-bit word. In the host, however, the smallest addressable unit is one byte. This gives rise to an address mismatch as the PDP-11 assembler location counter runs twice as fast as it should.

Another compatibility problem is due to addressing restrictions in the host computer. On many machines byte addressing is used, but only bytes can occupy arbitrary addresses. Larger storage units such as instructions, 16-bit words, and 32-bit words are restricted to addresses that are a multiple of 2, 4, etc. This is typically true for the PDP-11, where 16-bit words must lie on even addresses, and for the IBM/360 series where similar restrictions exist.

These limitations are not present in 8-bit microcomputers with 1-byte opcodes, where a 16-bit operand may start at any address. A first consequence of this fact is that many host assemblers are unable to generate code for such a 16-bit operand, unless the operand is decomposed into bytes. The second consequence is more serious: in the presence of host addressing restrictions, it is impossible to use the linker (linkage editor) of the host to generate absolute code. Furthermore, it is hard to write a special linking program since the decomposition mentioned above results in a complex relocation problem or even in the loss of relocation information.

Conclusion: Host assemblers with built-in addressing restrictions are generally unable to generate relocatable target code. Therefore, all program modules must be "linked" at assembly time, so that the assembler can generate absolute code. As a partial solution to this problem, macros can be written to declare a program module at the source level. They enable the user to assemble and list each module separately, in order to improve turnaround time and to encourage modular program design. Another partial solution, useful in modules with a tree-like dependence, is to extract global symbol definitions from the assembler output of the root by postprocessing. These definitions can then be included in the source files of subsequent assemblies.

C. Target Related Problems

Typical microprocessor instruction sets cause problems in any assembler implementation—not only one based on macro expansion. For completeness, we will discuss them briefly.

1) The Nonhomogeneity of Addressing Modes: By this we mean that not all instructions contain all addressing modes. This is particularly true for third generation microprocessors such as the Z80, 6800, and 6500. For example, the 6500 has 13 addressing modes, only eight of which are allowed in the LDA instruction, four in the INC instruction and two in the JMP instruction.

2) Instruction Size Optimization: Consider a target machine with instructions having a shorthand and a normal version. As a first example we have span dependent instructions (see [8]), such as a short range and a long range branch. Zero page addressing is another example: an instruction can have a single byte address instead of the normal two-byte address if the

operand is in the lower 256 bytes of memory. It is desirable for the assembler to perform automatic instruction size optimization.

Consider the case of zero page addressing. If, during the first pass of assembly, the operand is not known because of a forward reference, the assembler will temporarily (that is, until the second pass is executed) assume a zero value, leading to a zero page address. Thus, in the first pass 1 byte is reserved for the operand. If, during the second pass, the operand is found to be nonzero page, 2 bytes are generated and a location counter conflict occurs; the assembler is unable to generate correct code.

III. The Implementation

Before discussing the implementation of a macro-based cross assembler the problems discussed in the previous section must be given an adequate solution.

A. Method Related Problems

1) The Parsing Problem: Parsing general expressions with macro time assembly directives is very difficult and time consuming if not impossible. It is therefore advisable to introduce a notation that can be more easily recognized by the macro assembler while being as close as possible to the standard notation. We are in fact using the cross assembler as a universal assembler. It is possible to implement an assembler for all (or most) target machines on a universal assembler, but not to mimic a particular one [4].

Consider the recognition of the addressing modes of the LDA instruction (see Section II-B1). A naive solution would be to put the addressing mode information in the macro name. We could define eight macros for the instruction LDA. Source lines corresponding to the addressing modes listed above might then look like the following:

```
1) LDAI   ARG   (immediate)
2) LDAD   ARG   (direct)
3) LDAX   ARG   (indexed with X)
4) LDAY   ARG   (indexed with Y)
5) LDAXI  ARG   (indirect with preindex X)
6) LDAIY  ARG   (indirect with postindex Y).
```

Most of the elegance of addressing modes is lost in this notation. The number of macros to be defined increases dramatically, but each macro is rather simple. It explicitly describes the code to be generated. We do not check if an addressing mode is legal. There simply is no macro for it.

Another solution is to create a mnemonic for each addressing mode, e.g., the use of "#" to designate an immediate operand. We now have only one macro for each instruction, called with the addressing mode mnemonic as one of the parameters. The source lines equivalent to those listed above become:

```
1) LDA ARG,#    (immediate)
2) LDA ARG      (direct)
3) LDA ARG,X    (indexed with X)
4) LDA ARG,Y    (indexed with Y)
5) LDA ARG,X@   (indirect with preindex X)
6) LDA ARG,@Y   (indirect with postindex Y).
```

With macro facilities where the comma is the separator (which is nearly always the case), the string "ARG,@Y" is easily decomposed in "ARG", "@Y". The only requirement left is that the macro facility is capable of recognizing, e.g., "@Y" as a predefined string. We conclude this notation is much easier to process than the standard notation.

2) The Listing Problem: In a typical macro assembler the programmer can alter the listing status (enable/disable output listing, enable/disable macro expansion listing, etc.) by issuing the appropriate listing control directives. Providing a listing with no residual information from the macro expansion process can be attempted by manipulating the listing status of the host. This creates a more subtle problem: we want to allow the programmer to define his own macros, and control their listing at will. The host assembler, however, cannot distinguish between programmer defined macros and instruction macros. The listing directives issued by the programmer can interfere with the proper listing of the instructions. This problem can be solved if we are able to "read" the listing status upon entry of the instruction macro, save it, enforce the listing status suitable for the instruction macro, and then restore the programmer defined status upon exit. Unfortunately, not many assemblers provide a function to read the listing status.

Instead we took the following approach. Upon entry of the instruction macro, some elements of the listing status are changed. Upon exit they are restored to a default value. The programmer cannot change that value, unless a macro for it is provided. The net result is that the programmer freedom to control the listing is slightly impaired, but it is still greater than in most custom cross assemblers.

Some less important listing problems remain.

1) The machine code is not necessarily listed on the same line as the source line instruction, since it is generated by another statement.

2) Source lines with an error are always listed. If the programmer makes an error, several lines in the instruction macro may be listed. To the programmer these lines may be meaningless.

3) The number base used in the listing (e.g., octal) may not be the standard number base used for the target microprocessor.

These problems can be solved by postprocessing the listing.

B. Host Related Problems

1) Unavailable Number Radices: This is more of an annoyance than a problem. A naive solution is to predefine, for all values, an assembly time variable with a name that, in the desired number radix, suggests its value. This obviously clutters the symbol table. A better solution is to provide a macro that assigns a value expressed in the desired number radix to any assembly time variable. For example, to incorporate hexadecimal notation in an assembler that lacks it we could define a macro .HEX. The statement

```
.HEX VAR, 1FE
```

would then assign the value 1FE (hex) to the assembly time variable VAR. Of course, the host assembler should support character by character examination of macro parameters to implement such a macro.

2) Memory Organization: The smallest addressable unit in memory of the host is either smaller than, equal to, or larger than the smallest addressable unit of the target.

Obviously, there is no problem in the second case.

In the first case the solution is to use two program sections, one for location counter tracking and one for storing the assembled code.

In the third case the excess bits of the host words can be ignored or set to zero.

C. Target Related Problems

1) The Nonhomogeneity of the Addressing Modes: This problem is solved by subdividing the instruction set into groups of instructions with an identical set of addressing modes, and by defining a special macro for each group. In these macros extensive use is made of assembly time variables and powerful conditional assembly directives, such as character string comparison. This is treated in more detail in Section III-D.

2) Instruction Size Optimization: There are several ways out of this problem.

1) Prohibiting forward references for those instructions that allow implicit zero page addressing.

2) Generating full addressing in case of a forward reference. This eliminates conflicts, but it is not code efficient.

3) Requiring the user to specify zero page addressing explicitly.

4) Using a multipass assembler or using a special two-pass assembler such as described in [8].

The last solution is the best one. However, nearly all existing assemblers suitable for cross assembly are of the classical two-pass type.

D. The Systematic Implementation of the Cross Assembler

We will illustrate our discussion with the implementation of AS6500, a cross assembler for MOS TECH 6500 which was implemented on a PDP-11/34 with DEC's standard macro assembler Macro-11 as the host assembler. It is an 8-bit microprocessor with one-, two-, and three-byte instructions. The first byte contains the opcode, the other bytes the operand, if any. Some instructions have several addressing modes, but these are nonhomogeneous. The addressing mode is indicated in a field in the opcode. Furthermore, the processor has zero page addressing modes for several, but not all, instructions. This mode should be used by the assembler without explicit programmer intervention.

For each target instruction we define a macro in the host assembler, called instruction macro. This instruction macro performs the following functions.

1) It extracts the operands.

2) It tests whether an addressing mode is legal and fills out the addressing mode field in the opcode.

3) It generates error messages if necessary.

4) It writes the object code in the object file and in the listing file.

5) It suppresses the listing of its expansion, except perhaps the statement where the object code is generated.

Provided the macro assembler allows nested macro calls, functions common to many instruction macros can be incorporated within auxiliary macros. These auxiliary macros are not used by the programmer, but are called by the instruction macros. Such macros increase the readability and also make debugging and modification easier. Some of them can be taken over when a cross assembler for another target instruction set is implemented.

E. Code Macros

The generation and listing of machine code is common to all instruction macros. It could be concentrated in a few macros, the code macros. The instruction macro calls the code macro and passes the opcode and the operand(s) to it as parameters. The instruction macro then writes the corresponding code to the object file and to the listing file.

Hence, there should be a code macro for each instruction length or format. The instruction macro will call the appropriate code macro for each addressing mode encountered.

The instruction length can depend upon the operand because of possible zero page addressing. Other factors, such as certain default settings, can affect the instruction length. Therefore, a level of intermediary code macros should be provided. These intermediary macros are called by an instruction macro. They determine the instruction length and call the corresponding final code macro.

For AS6500 we define the following code macros.

1) .BT1 OPCODE for one-byte instructions.

2) .BT2 OPCODE,OPERAND for two-byte instructions.

3) .BT3 OPCODE,OPERAND for three-byte instructions.

4) .PT OPCODE,OPERAND for those instructions that have both a two- and three-byte version. This is an intermediary code macro. It calls .BT2 in case of zero page addressing and .BT3 otherwise. Furthermore, it adjusts the addressing mode field.

F. Group Macros

We can precisely define an instruction by specifying the following three elements.

1) The mnemonic.
2) The opcode with addressing mode field set to zero.
3) The description of the addressing modes. For each legal addressing mode we must specify:
   a) the addressing mode mnemonic;
   b) the code macro to be invoked;
   c) the addressing mode field in the opcode.

These three elements completely and unambiguously define an instruction as far as the assembler is concerned.

Hopefully, the set of addressing modes, or more precisely, the third element of the description, will be common to several instructions. Therefore, we introduce the concept of instruction groups. An instruction group contains all the instructions with the same set of addressing modes. A group is defined by its name and the description of the addressing modes allowed in that group. An instruction is then defined by its mnemonic, its opcode, and the name of the group it belongs to.

As an example of an instruction definition for AS6500 consider the instruction

```
LDX,241,GR2.3 .
```

The instruction mnemonic is LDX (load register X), its opcode with the addressing mode field set to zero is 241 (octal), and GR2.3 is the name of the group it belongs to.

Here is an example of a group definition,

```
GR2.3,<<,.PT,4>,<#,.BT2,0>,<Y,.PT,20>>.
```

The name of the group is GR2.3, the list of the addressing mode descriptions is between angle brackets. Each individual description is also between angle brackets. This particular notation was chosen for its convenience in Macro-11, and is otherwise irrelevant. The first addressing mode is described by the string <,.PT,4>, in which we specify the following.

1) The mnemonic: null string, which stands for direct addressing.

2) The code macro: .PT, for automatic choice between zero page and full addressing.

3) The addressing mode field: contents = 4.

The second description stands for immediate addressing, with mnemonic #. It is a two-byte instruction (.BT2). The address field is zero. The third allowed mode is indexed with register Y (mnemonic Y). There is a two-byte and a three-byte instruction for it.

In this way we can give a formal description of the assembler to be implemented.

The instruction macro must check whether an addressing mode is legal, and if so, must fill out the addressing mode field in the opcode and call the corresponding code macro. Otherwise, an error message must be generated. This part of the instruction macro is the same for all instructions belonging to the same group. We can thus define a set of macros, termed group macros, that will perform that function.

G. Error Message Macros

The assembler directive that generates a particular error message will probably be repeated many times in the source text. It is then appropriate to define a macro for that function.

H. Generation of the Instruction and the Group Macros

Through the introduction of the group and code macros as described above, we can make all instruction macros look alike. In fact, the various instruction macros differ only in:

1) their name

2) the group macro to call

3) the opcode.

Thus, writing the instruction macros could be done in a naive way by writing a standard instruction macro and then adapting it manually for each instruction. As observed earlier the only parts to be modified are the mnemonic, the opcode, and the group name.

However, if the macro assembler allows nested macro definitions, there is a much more elegant and less error prone method to generate the instruction macros. We can define a macro of which the parameters are the mnemonic, the opcode, and group macro defining an instruction. Calling this macro generates the definition of the corresponding instruction macro. We refer to that macro as the instruction generating macro or the instruction generator.

We illustrate the above with the instruction generator of AS6500. First we have a few words about the syntax of Macro-11. A macro is defined with the .MACRO directive. The first argument of the directive is the macro name, the other arguments are the formal parameters. The macro body is between the .MACRO directive and the .ENDM directive with the macro name as argument. In AS6500 the instruction generating macro is .INS, and it is defined as follows:

```
.MACRO  .INS            MNEMONIC,OPCODE,GROUP
.MACRO  MNEMONIC        OPERAND,MODE
.NLIST
.E$=0
GROUP   OPCODE,OPERAND,MODE
.IIF EQ,.E$,ERRMAC
.LIST
.ENDM   MNEMONIC
.ENDM   .INS
```

Calling .INS with the description of the instruction LDX as arguments generates the definition of the instruction macro LDX. The call of the instruction generating macro then becomes

```
.INS  LDX,242,GR2.3
```

resulting in the macro definition

```
.MACRO  LDX     OPERAND,MODE
.NLIST
.E$=0
GR2.3   242,OPERAND,ADMODE
.IIF EQ,.E$,ERRMAC
.LIST
.ENDM   LDX
.ENDM   LDX
```

To define all instruction macros it suffices to supply their definitions to the instruction generator.

A similar technique can be used to generate the group macros. The group generator macro is named .GRGN in AS6500. For example, the macro call

```
.GRGN  GR2.3    <<,PT.,4>,<#,.BT2,0>,<Y,.PT,20>>
```

will define the group macro GR2.3.

Defining a group macro is thus reduced to supplying the group macro generator .GRGN with the description list. Three group macros with particular features, such as relative addressing, were hand-coded. The effort of devising a more general macro .GRGN outweighed the gain.

Imbedded macro definitions have several major advantages over manual adaptations of a standard definition, among which we have the following.

1) The source text is several times shorter.

2) It is less error prone.

3) The same function is not defined in different places. This makes debugging and modification much easier.

If nested macro definitions are not allowed, but a separate macroprocessor is available, we can use the same method by feeding the macroprocessor with the calls to the generator macros as a prepass. The result is a file containing the definitions of all instruction and group macros.

IV. On the Relationship Between Host and Target Assembler

A. Cross Assembler Features

Whether or not the host assembler must be powerful depends largely upon the instruction set of the target microprocessor. In [1], for example, a cross assembler for the 8080 is described, running on a very unsophisticated host assembler, lacking almost all of the features described hereafter, including macro processing. This last feature, clearly indispensable, is provided by a relatively simple macro processor acting as a prepass before assembly.

However, implementing a 6500 cross assembler using the same cross assembly system has proven to be almost impossible. This is mainly due to the fact that the 6500 instruction set contains explicit addressing modes, whereas the 8080 does not: its (few) addressing modes are implied in the instruction mnemonic (see also Section III-A).

Severe restrictions would also be imposed upon the 6500 source language, such as compelling the user to distinguish explicitly between absolute addressing and zero page addressing. This would be necessary because the simple assembler does not provide conditional assembly.

We concluded that the implementation of certain features requires the use of a more-than-elementary host assembler. Some features of the cross assembler which require a relatively sophisticated host assembler are the following.

1) An instruction set with explicit addressing modes.

2) The generation of relevant error messages. This requires that error conditions specific to the cross assembler, such as a nonexistent addressing mode, can be intercepted by the appropriate macro before the host assembler generates an irrelevant message (e.g., "undefined symbol" for an illegal addressing mode).

3) An aesthetic listing containing only information relevant to the user. The expansion of the cross assembler macros should never be listed.

4) Macro processing. At first sight this is trivial since a host macro assembler is used. However, the host assembler has to support nested macro calls to a sufficient nesting depth. Furthermore, for diagnostic purposes it should be possible to list the expansion of the user defined macros, while the cross assembler macro expansion should remain invisible.

The last two features require extensive listing control on the host assembler.

B. Features of the Host Assembler

As a result of our study we can give an overview of the capabilities necessary or very convenient for a host assembler to be suitable for macro-based cross assembly.

These capabilities are as follows.

1) Macro processing with capability for nested macro calls. Imbedded macro calls allow a reasonably compact description of the cross assembler using group and code macros. They are also needed for user defined macros.

2) Assembly time variables. They are very useful in conjunction with the conditional assembly directives.

3) Assembly time arithmetic and logical operators. These are mainly used for instruction assembly, using as ingredients the basic opcode, the addressing mode bits, and the operand.

4) An extensive set of conditional assembly directives, allowing testing of numeric as well as alphanumeric values.

5) An extensive set of listing directives. These directives should provide the ability to generate a listing without residual information from the expansion process.

6) Ability to generate absolute code.

Some host assembler features which are not essential, but still desirable are as follows.

1) Imbedded macro definition. This allows a very compact description of the cross assembler in terms of macro generating macros (see Section III-H).

2) Assembly time shift/rotate functions. Bit field positioning is most easily done using shift and rotate functions. These functions can often be substituted for by multiplication and division, but this is time consuming and it requires great care. No assembler known to the authors offers such features.

3) A more general listing control. Ability to *READ* the listing status of the assembler would greatly facilitate the generation of garbage free listings, allowing at the same time full listing control by the user, without side effects. Control over the number base of the source and of the listing could eliminate the need for postprocessing.

4) It is convenient if the smallest addressable unit of the host computer is not smaller than that of the target computer.

5) Character by character string recognition.

V. Conclusion

Macro-based cross assemblers can be implemented very rapidly provided that some problems are identified and understood. These problems have been discussed in detail and possible solutions have been proposed. A systematic implementation technique, requiring a minimum of code to be written, has been discussed. Indications have been given to evaluate an existing assembler as to its ability to behave as a macro-based cross assembler.

Many of the difficulties encountered during the implementation of a cross assembler are due to peculiarities of the host assembler. Indeed, this assembler is very much biased toward the architecture of the host computer. The implementation of a general-purpose cross assembler should be considered. This assembler would be stripped down in the sense that it would not recognize any instruction mnemonic, but only directives or pseudoinstructions. The length of the smallest addressable unit should be user definable. Such an assembler would be more compact than a dedicated assembler and it would provide all those facilities we considered necessary or useful. Also, it should be able to produce relocatable code.

A further step would be to generalize the conditional assembly directives which play a very important role in the macro assembly process. High level language constructs could be provided such as block structure, the if-then-else and case constructs and even repetition constructs as for, while and until. This would transform the macro-based cross assembler into a assembler compiler.

Acknowledgment

The authors wish to thank Dr. Van Canpemhout for his many helpful comments and thorough proofreading.

References

[1] P. H. Notredame, "Toepassingen van ALMAC, een algemene macro-processor," Rijksuniversiteit Gent, Laboratorium voor elektronika en meettechnick, Internal Rep., Ghent, Belgium.
[2] T. A. Seim, "Assembling microprocessor software with minicomputers," Battelle, Pacific Northwest Lab., Richland, WA.
[3] I. M. Watson, "Comparison of commercially available software tools for microprocessor programming," *Proc. IEEE*, June 1976.
[4] D. W. Barron, *Assemblers and Loaders*. Amsterdam, The Netherlands: Elsevier, 1972.
[5] M. Campbell-Kelly, *An Introduction to Macros*. Amsterdam, The Netherlands: Elsevier, 1971.
[6] Digital Equipment Corp., IAS/RSX-11 Macro-11 Ref. Manual.
[7] Digital Equipment Corp., IAS/RSX-11, I/O Operations Reference Manual, ch. 7 (The table driven parser (TPARS)).
[8] Szymansky, "Assembling code for machines with span-dependent instructions," *Commun. ACM*, vol. 21, Apr. 1978.

Karel R. Tavernier, photograph and biography not available at the time of publication.

Paul H. Notredame, photograph and biography not available at the time of publication.

Overhead Storage Considerations and a Multilinear Method for Data File Compression

TZAY Y. YOUNG, MEMBER, IEEE, AND PHILIP S. LIU, MEMBER, IEEE

Abstract—**The paper is concerned with the reduction of overhead storage, i.e., the stored compression/decompression (C/D) table, in field-level data file compression. A large C/D table can occupy a large fraction of main memory space during compression and decompression, and may cause excessive page swapping in virtual memory systems. A two-stage approach is studied, including the required additional C/D table decompression time. It appears that the approach has limitations and is not completely satisfactory.**

A multilinear compression method is proposed which is capable of reducing the overhead storage by a significant factor. Multilinear compression groups data items into several clusters and then compresses each cluster by a binary-field linear transformation. Algorithms for clustering and transformation are developed, and data compression examples are presented.

Index Terms—**Cluster analysis, data file compression, data transformation, multilinear approach, overhead storage, performance analysis, piecewise linear transformation, storage reduction techniques.**

I. Data File Compression and Overhead Storage

CONSIDER a data file consisting of fixed-length records. The contents of a record are divided into several fields, with each field representing an attribute or a key. To reduce the size of a file, data compression may be used at record level or field level. Record-level compression treats each record as a data vector during compression and decompression. In many files different types of data items are stored in the data fields, and some fields may be compressed more effectively than others. It may be more successful to compress the data fields separately, using different compression techniques. This is called field-level compression, and data items in a data field, or possibly items formed by merging two or three fields together, are regarded as data vectors to be compressed, apart from other data fields of the records. It is noted that data items within the same data field are not stored contiguously in the secondary memory.

A very simple and commonly used method for field-level compression is the fixed-length minimum-bit (FLMB) encoding scheme. Consider the set of data items in a data field. With N_r records in the file, there are N_r data items in a field, one for each record. But some of the data items may be identical, and there may be only N distinct data items, $N < N_r$. Let

$$n = \lceil \log_2 N \rceil \tag{1}$$

where $\lceil \cdot \rceil$ denotes the smallest integer greater than or equal to its argument. Clearly each data item can be represented uniquely by an n-bit binary number. The compression ratio is l/n, where l is the bit length of the original data item. FLMB compression can be very effective when N is substantially smaller than N_r. For example, in a file of student records kept by an academic department, the field of course numbers (e.g., EEN 201) is of major concern, and it can be compressed effectively by the FLMB scheme since a course is taken by many students.

The FLMB scheme requires the construction and storage of

Manuscript received March 9, 1979; revised December 6, 1979. This work was supported in part by the National Science Foundation under Grant MCS 77-01483. Preliminary results were presented at COMPSAC 79.

The authors are with the Department of Electrical Engineering, University of Miami, Coral Gables, FL 33124.

0098-5589/80/0700-0340$00.75