1

FINAL REPORT(Part I)

A MICROCODE COMPILER THAT RUNS ON THE IBM AT AND SUPPORTS CASCADABLE MICROCOMPUTERS

by

Thomas H. Weight,Ph.D.

Period Covered: 23 Sept 87 to 22 Sept 89

Contract DAAD07-87-C-0119
for
White Sands Missile Range, New Mexico 88002

11 Nov 89

PENGUIN SOFTWARE,Inc.
7005 E. Spring St.
Long Beach, Calif. 90808

Thomas H. Weight
Thomas H. Weight,Ph.D.
Principal Investigator
PENGUIN SOFTWARE,Inc.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS --
2a. SECURITY CLASSIFICATION AUTHORITY ------------------------ 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE ------------------------	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution is unlimited. PART I
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ------------------------	5. MONITORING ORGANIZATION REPORT NUMBER(S) --
6a. NAME OF PERFORMING ORGANIZATION PENGUIN SOFTWARE, Inc. 6b. OFFICE SYMBOL (If applicable) ---------	7a. NAME OF MONITORING ORGANIZATION U.S. ARMY White Sands Missile Range
6c. ADDRESS (City, State, and ZIP Code) 7005 E. Spring St. Long Beach, CA 90808	7b. ADDRESS (City, State, and ZIP Code) COMMANDING OFFICER, STEWS-ID-T U.S.ARMY White Sands Missile Range New Mexico 88002-5143
8a. NAME OF FUNDING/SPONSORING ORGANIZATION ----------------- 8b. OFFICE SYMBOL (If applicable) ---------	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAD07-87-C-0119
8c. ADDRESS (City, State, and ZIP Code) ------------------------	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 665502 PROJECT NO. 1P665502M40 TASK NO. -------- WORK UNIT ACCESSION NO. -------

11. TITLE (Include Security Classification)
A Microcde Compiler that runs on the IBM AT

12. PERSONAL AUTHOR(S)
Thomas H. Weight, PH.D.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final Technical	FROM 23Sept87 TO 22Sept89	1989,Nov,11	

16. SUPPLEMENTARY NOTATION
----

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
------	------	---------
------	------	---------

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
High-level language, microprogramming, automated microcode generation, micrcompiler

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

PENGUIN SOFTWARE,Inc. has developed a retargetable microcode compiler. Our approach does not have a fixed machine independent language, but allows the user to develop a language specific to each particular target machine. PENGUIN SOFTWARE"s Microcode Compiler starts out with an underlying meta-assembler and builds up a higher level language capability around it. This capability allows the user to incorporate knowledge of target machine design into the language definition, and thus avoid the necessity for resource allocation and code compaction in the application program. This approach results in a microcode development tool which is a very low risk, very fast, and is capable of supporting virtually any digital hardware architecture. [illegible]

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL FOO LAM	22b. TELEPHONE (Include Area Code) (505)678-3010 22c. OFFICE SYMBOL STEWS-ID-T

intentionally left blank

SUMMARY

The microcode compiler being developed at PENGUIN SOFTWARE, Inc. is based on the concept of a retargetable compiler. This approach does not have a fixed machine-independent language, but allows the user to develop a language specific to each particular target machine. This provides a means for the user to incorporate knowledge of target machine design into the language definition, and avoid the necessity for resource allocation and code compaction in the application program.

Previous microcode compilers have evolved from traditional compiler concepts and design methodologies. They have adopted an approach involving machine-dependent languages and have attempted to capitalize on the associated advantages. This approach invariably gets bogged down in the problems associated with microcode compaction and resource allocation. Until these problems are solved, this approach has to be considered very high risk.

The Microcode Compiler being developed at PENGUIN SOFTWARE, Inc. has evolved from the classic microprogramming tool i.e. the meta-assembler. Our microcompiler starts out with an underlying meta-assembler and builds up a higher level language capability around it. This approach results in a microcode development tool which is a very low risk, and is capable of supporting virtually any digital hardware architecture.

This microcode compiler is a new type of program. According to one source, this approach has never been tried before; at least no other retargetable microcode compiler exists. This study has established the advantages of this approach to be:

1. Low Risk : There are no high risk algorithms remaining to be developed.

2. Robustness : This compiler can support any processor architecture supported by the current state-of-the-art meta-assemblers.

3. Speed : This compiler is competitive with meta-assemblers in terms of speed. This is in sharp contrast to other microcompiler designs.

4. Timeliness : This approach is working now.

Currently, PENGUIN SOFTWARE, Inc. is involved in a continueing development effort to produce a commercially marketable microcompiler which is capable of supporting the development of microcode for cascadable, parallel, and systolic arrays of microprocessors.

PREFACE

This report describes the microcode compiler feasibility study and the compiler development undertaken at PENGUIN SOFTWARE,Inc. under the phase two SBIR contract DAAD07-87-C-0119 sponsored by White Sands Missile Range.

There is currently a critcal need for a fully retargetable microcode compiler and for an effective development tool to support programming the new non-von Neuman architecture microprocessors. As more parallel processors, systolic arrays, and cascadable processors become available, these needs can only become more acute. Most of the chips resulting from the VHSIC program and all near-term gallium arsenide devices will require microcoding. For the benefits of these advances to be fully realized, microcode development will have to be automated in the relatively near future.

Our feasibilty study required the development of two compilers: the Syntax Compiler and the Microcode Compiler. The Syntax Compiler generates a file which defines the syntax and semantics of a machine dependent language. The Microcode Compiler along with this "Syntax File" then becomes tailored to a particular target machine and a particular language. An application program is then written in this new language and compiled into an intermediate language file.

This intermediate language is typically a lower level language than the application programing language but higher than the binary object file. We choose to use HILEVEL Technology's HALE meta-assembler language as our intermediate level language. Since HALE is more or less campatible with MICROTEK Research, using HALE is not as restrictive as it'may at first appear.

The advantages of using a state-of-the-art meta-assembly language as an intermediate level language were quite significant. The intermediate file is easily modified by the programmer in a familiar language; the meta-assembler handles the actual assembly, linking, and PROM formatting.

While there are very few research journal articles related to our work, we have not been forced to work in a vacuum. We would like to thank the marketing staff at HILEVEI Technology in Irvine, and we would especially like to thank Mr. Warren Long, Product Marketing Manager at HILEVEL technology for his many suggestions in the areas of requirements and engineering design. We would also like to thank the engineers and managers of Rockwell International, Hughes Aircraft, Northrop, and TRW that have contributed their time and allowed themselves to be interviewed.

Part I - FINAL REPORT

TABLE OF CONTENTS

LIST OF TABLES

1.0 INTRODUCTION

The microcode compiler being developed at PENGUIN SOFTWARE, Inc. is a retargetable compiler. This compiler does not have a fixed machine-independent language, but allows the user to develop a language specific to each particular target machine and application.

The microcode compiler is a two stage translation scheme. The first stage provides a fixed high-level design language to permit the easy definition of the machine-dependent high-level language. The language definition compiler, called the Syntax Compiler, produces syntax files which define both the syntax and semantics of the target machine language.

The application program is compiled by the Microcompiler to produce the object file microcode. The application programs are written in a language specifically designed for the particular target processor. This language is almost completely free from constraints with just enough underlying language structure to facilitate the application language compilation process.

The approach to automated microcode generation being developed at PENGUIN SOFTWARE, Inc. is based on the concept of machine-dependent languages. This provides a means for the user to incorporate knowledge of the target machine into the language definition. There are some obvious advantages and disadvantages to this approach.

Some of the advantages of this approach are expressiveness, robustness, and low risk.

a. This approach allows the programmer to design a machine-language tailored to the needs of a particular application. This can result in a highly expressive and efficient language.

b. Because of this microcode compiler's relationship to the underlying HALE meta-assembler, this microcode compiler is extremely unlikely to fail due to a particular target machine design.

c. Finally, this approach avoids the tough problems of microcode compaction and resource allocation.

Some of the disadvantages of this approach are lack of a stable application language, and the difficulties associated with trying to imbed low-level hardware details in a high-level language.

a. A stable application language is desireable because it makes application programs more transportable and facilitates the development of microcode simulators.

b. Because of the hardware design, it may be difficult to design a suitable high-level language for a particular application.

The basic requirement of our Phase II effort was to demonstrate the feasibility of PENGUIN SOFTWARE's approach to microcode development. Section 2.0 describes, in some detail, the research and development efforts expended at PENGUIN SOFTWARE, Inc. to meet the Phase II requirement and summarizes the important results from Phase I. These results were essentially the design specification of the class of languages supported by the Microcode Compiler.

Section 3.0 describes the objectives of the Phase II contract. Under the current contract, the following objectives were completed at PENGUIN SOFTWARE, Inc.:

a. Developed the operational concept based on using HALE as an intermediate level language. We had to varify that we could use our Microcode Compiler essentially as an HALE preprocessor.

b. Upgraded existing software to reflect our new philosophy. The Microcode Compiler now functions more generally as a semantic action processor, and not just a high-level language processor designed to produce binary object files.

c. Modified source code to output HALE as an intermediate-level language. In addition to modifying the Syntax and Microcode Compilers to output ascii code, the capability to support HALE pseudo-operations was also required.

d. Included the capability to support cascadable microprocessors. The Microcode Compiler now supports one or more cascaded processors as specified by parameteric command line input.

e. Developed a dynamic memory allocation scheme. Now key arrays are stored either in lower RAM or in extended memory above one million bytes. This feature is designed to keep the Microcode Compiler from becoming obsoleted by a large program.

f. Optimized the Microcode Compiler for Speed. The Microcode Compiler was restructed in order to speed up processing during the time critical compilation of the application program.

Section 4.0 gives a summary of conclusions and recommendations. We concluded that all technical objectives were attained. This project shows that this is the low risk approach to developing a Microcode Compiler. This follows from the evolutionary (rather than revolutionary) nature of our relationship to the proven meta-assembler technology. After restructuring the Microcode Compiler runs quite fast, and should be completely satisfactory commercially. Our approach to high-level language microcoding has proven to be quite robust. We have found that for difficult problems, the level of the application language may degrade. But there appears to be no significant tendency for the efficiency of the microcode to be impaired.

2.0 MICROCODE COMPILER STUDY

PENGUIN SOFTWARE's retargetable microcode generator system emulates a wide range of computer architectures. By allowing the user to define a machine-dependent high-level language, and then write an application program in this new language, the microcode generator is able to translate the application program into efficient horizonal microcode.

Our minimum goal was to produce a working microcode compiler which would be a useful tool in industry. In order to achieve this goal, we determined to start with the most powerful meta-assembler that we could find and then enhance its capabilities to achieve a higher-level language capability. For a project overview, see Appendix A. Many of the project tasks were divided into two parts. The first part, the syntax analysis, determines the readability of the language. The second part, the semantic action, determines to a large extent the level of the language.

2.1 Define Machine Dependant Languages

2.1.1 Statement Notation

Two types of statement notation were considered. The first type, "action verb", is similiar to the standard assembly language statement. Unfortunately, "action verb" does not seem compatible with the requirements of horizonal microprogramming. The second type of statement notation, "list notation", is ideal for horizonal microcoding. This statement has the basic form shown in Fig. 1. We have adopted the convention that all labels are immediately followed by a colon, the entities are separated by ampersands, and the last entity is followed by a semicolon. Everything to the right of the semicolon is a comment.

2.1.2 Entity Notation

The lowest level of entity notation identified is "positional notation." In positional notation each entity is related to one particular field. This notation does not appear to have any role in our high-level language except possibly to force specific values into a specific microword, and even then there seems to be better ways to accomplish that.

Several nonpositional entity notations were identified. These include "function reference notation", "value mnemonic notation", and "keyword notation". All of these notations are desireable in different applications. Rather than decide on anyone of them, a different notation has been selected which includes all of the above: this is the "free notation". In the free notation there are almost no constraints on the

```
<label>:   <entity>&<entity>&...&<entity>;   <comment>
```

FIGURE 1. List Notation Format.

```
<label>: <clause>&<clause>& ... &<clause>;<comment>
```

FIGURE 2. Compiler Sentence Format.

```
CLAUSE  {<syntax definition>}{<semantic actions>}
```

FIGURE 3. CLAUSE Definition Format.

notation used. The only constraints imposed on the entity notation are those derived from the design requirements and the implementation of the program.

2.2 Define Syntax Definition Language

We have adopted the terms "sentence" for the entire line from the label to the end of the comment, and the term "clause" for the entity as shown in Fig.2. Since only the structure of the sentence is hard-wired into the microcode compiler, it remains to the syntax compiler to define the syntax and semantics of the clauses. A syntax compiler source file is constructed using the language outlined below. This file will define all of the legal application language clauses and define what these clauses mean. The basic clause definition appears as shown in Fig. 3.

2.2.1 Syntax Definition

The syntax definition process for each clause is accomplished by using three related contructs: literal strings, arrays, and literals.

Literal strings are the simplest and in some cases the most efficient means for defining the syntax of a clause. A literal string is a string of alphabetic or special characters enclosed in double quotes. Numerical digits can also be included, but they must be preceeded by an alphabetic character. Ignoring the semantic definitions for the moment, the examples as shown in Fig. 4 show clauses defined using the literal string.

Literal strings are most useful for clauses, such as "NOP", which are, in a sense, one-of-a-kind. Literal strings are not very efficicient in situations like "REG1 = REG12" where there are several registers that may appear on either side of the equation such as shown in Fig. 5. This situation is handled by the two types of arrays described below.

First, the ARRAY is identified by an array name, then the ARRAY is defined by an array of tokens and their associated semantic actions. Fig. 6 is a simple example of an ARRAY structure. An ARRAY is used in a CLAUSE definition simply by specifying the ARRAY name. Using an ARRAY in a CLAUSE definition can considerably reduce the amount of effort required as can be seen by the example in Fig. 7. In this example, we have a very simple situation where the contents of any source register can be moved into any destination register. Experience has shown that quite often these ARRAY's will be used repeatedly to define other CLAUSE's. This results in a tremendous savings of programming effort.

```
CLAUSE  { "NOP" }

CLAUSE{ "REG1 = REG12" }

CLAUSE { "REPEAT(FOREVER)" }
```

FIGURE 4. Example: Literal String Definitions.

```
CLAUSE { "REG1 = REG1" }
CLAUSE { "REG2 = REG1" }
CLAUSE { "REG3 = REG1" }
           .
           .
           .
CLAUSE { "REG1F = REG1" }
CLAUSE { "REG1 = REG2" }
CLAUSE { "REG2 = REG2" }
           .
           .
           .
CLAUSE { "REG1F = REG2" }
CLAUSE { "REG1 = REG3" }
           .
           .
           .
CLAUSE { "REG1F = REG1F" }
```

FIGURE 5. Example: When NOT to use Literal Strings.

```
ARRAY    <array name> = {
                   <token #1>{ semantic action };
                   <token #2>{ semantic action };
                             .
                             .
                             .
                   <token # n>{ semantic action };
}
```

FIGURE 6. ARRAY Definition Format.

```
CLAUSE { REGD "=" REGS }

ARRAY REGD = {
         REG1{ semantic action };
         REG2{ semantic action };
         REG3{ semantic action };
                .
                .
                .
        REG1f{ semantic action };
}
ARRAY REGS = {
        REG1{ semantic action };
        REG2{ semantic action };
        REG3{ semantic action };
               .
               .
               .
       REG1f{ semantic action };
}
```

FIGURE 7. Example: Using ARRAY Definitions.

Occasionally, we have a situation, unlike the preceeding example, where the source and destination registers are not orthogonal. That means, in this case, only certain source registers can be used with certain corresponding destination registers. The result is that simple ARRAY's would no longer work efficiently. Now we could fall back on the use of literal strings as discussed before and define several CLAUSE's as shown in Fig. 8. Although this approach would work, it is still possibly quite inefficient. For this reason, we have adopted an approach using dimensioned ARRAY's. Dimensioned ARRAY's have the particular property that if more than one column of the ARRAY is used in a CLAUSE definition, then the tokens must all come from the same row. The definition of a dimensioned ARRAY is shown in Fig. 9. A dimensioned ARRAY is used to solve the CLAUSE definition problem discussed above as shown in Fig. 10.

The final construct is the LITERAL. LITERAL's are used to define the processing of numbers and statement labels. The LITERAL is defined as shown in Fig. 11 and is used in a clause as shown in Fig. 12. The LITERAL allows the application programmer to include values in the program at compile time.

2.2.2 Semantics Definition

The semantics of a particular statement is defined in terms of the semantic actions to be performed when that statement is encountered in an application program. For each application program statement, the microcode compiler must reconstruct the statement definition to determine the particular CLAUSE, ARRAY row, and LITERAL as appropriate. These semantic actions are associated with the CLAUSE's, rows of the ARRAY's, and with the LITERAL's.

The semantic actions which have been implemented are desribed below:

a. DEF'ed statement: A DEF'ed statement with field values specified or left blank. This statement is to be inserted into the HALE source file, along with any other statements associated with the current line of microcode.

b. LBL: LBL appearing in a DEF'ed statement will cause the contents of the Active Label Register to be inserted into the corresponding field.

c. PSH: PSH will cause the contents of the Active Label Register to be pushed onto the Microcode Compiler Internal Stack.

d. PPS: PPS will cause the Microcode Compiler Internal

```
CLAUSE { "REG1 = REG5" }
CLAUSE { "REG1 = REG7" }
CLAUSE { "REG2 = REG1" }
             .
             .
             .
CLAUSE { "REG1F = REG12" }
```

FIGURE 8. Example: When NOT to use simple ARRAYs.

```
        ARRAY    <array name>(dim #1,dim #2,...,dim #m) = {
<token(1,1),token(1,2),...,token(1,m)>{ sematic action }
<token(2,1),token(2,2),...,token(2,m)>{ sematic action }
                                     .
                                     .
                                     .
<token(n,1),token(n,2),...,token(n,m)>{ sematic action }
        }
```

FIGURE 9. Dimensioned ARRAY Definition Format.

```
CLAUSE { REGDS(1) "=" REGS(2) }

ARRAY REGDS(1,2) = {
REG1,REG5{ semantic action };
REG1,REG7{ semantic action };
REG2,REG1{ semantic action };
REG3,REG5{ semantic action };
         .
         .
         .
REG1F,REG12{ semantic action };
}
```

FIGURE 10. Example: Using Dimensioned ARRAYs.

```
LITERAL( <literal number> ) = { semantic action }
```

FIGURE 11. LITERAL Definition Format.

```
CLAUSE { REGD "=" LITERAL(1) }
LITERAL(1) = { semantic action }
ARRAY REGD = defined as above in figure 7.
```

FIGURE 12. Example: Using LITERAL.

Stack to be popped, and the value popped will be inserted into the Active Label Register.

e. MAK: MAK will cause a label to be created and stored in the Active Label Register.

f. PLB: PLB will cause the label stored in the Active Label Register to be inserted into the HALE source file at the current microcode address.

g. LBLn: LBLn (where n = 1,2,3,4,5) will cause the contents of the Active Label Register to be stored in Microcode Compiler Internal Register n.

h. SWP: SWP will cause the top two labels stored in the Microcode Compiler Internal Stack to be swapped.

i. LIT: LIT used in a LITERAL pseudo operation will cause the corresponding literal to be stored in the Active Label Register.

j. LITn: LITn (where n = 1,2,3,4,5) will cause Microcode Compiler Internal Register n to be copied into the Active Label Register.

2.3 Compiler Capabilities

One of the key design decisions was to use the HILEVEL Technology HALE Meta-assembler language as an intermediate language for our compiler. As a consequence, the compiler was modified to output HALE compatible code, to handle HALE pseudo-operations, and to process HALE Definition Phase source files. Our compiler gained flexibility and many of HALE's most important capabilities at very little cost.

2.3.1 MACRO's

One of the advantages of using a high-level langauge is the ability to generate more than one line of object code from one line of source code. This capability has been included in the microcode compiler.

2.3.2 Structured Control

Structured control statements allow the development of block structured code and an orderly approach to program control flow. Structured control is included as an important part of the microcode compiler. Since developing structured control statements requires the use of Microcode Compiler internal resources, a brief example may be appropriate.

Suppose you intend to provide in your language the DO...WHILE instruction as shown in figure 13. Reasonably, the DO statement must provide for:

1) The creation of a statement label (MAK).

2) The assignment of the newly created label to the current location in the microcode (PLB).

3) The saving of the label on the Microcode Compiler Internal Stack (PSH).

The WHILE statement must then:

4) Retrieve the label from the stack (PPS).

5) Insert the label into the microcode as the destination of a conditional jump statement using the LBL pseudo opcode.

For further examples, see the PSI Microcompiler User's Manual

2.3.3 Compiler Evaluation Of Expressions

Arithmetic expressions are not evaluated during the compile phase. There are inserted into the HALE source file output from the compiler. The final evaluation of the expressions is performed during the HALE assembly phase.

2.3.4 Pseudo Operations

PENGUIN SOFTWARE's microcode compiler was modified to use all of the HALE meta-assembler pseudo-operations. Most of the psuedo-ops are just inserted into the HALE source file at the appropriate places e.g MAP. Other pseudo-ops are processed in the microcode compiler or the HALE meta-assembler as required e.g. LIST.

2.3.5 Conditional Compilation

There were essentially two ways to handle conditional compilation. One was to evaluate the condition and just insert the appropriate microcode into the HALE source file. The other approach, the one adopted, was to insert the conditional statements and both parts of the microcode into the source file. This final approach is much closer to the way things are commonly done, and was much easily to implement.

2.3.6 Relocatable Object Code

Relocatable microcode is produced by including the appropriate pseudo-ops in the Microcode Compiler source file. These pseudo-ops are then inserted into the HALE source file

```
DO
   .
   .
   .
WHILE( <some condition> )
```

13.a EXAMPLE: Application Program use of DO...WHILE

```
CLAUSE { "DO" }{ MAK & PLB & PSH }
CLAUSE { "WHILE(" <some condition> ")" }
          { PPS & AM2910   ,LBL,,CJP }
```

13.b EXAMPLE: Possible Definition of DO...While

FIGURE 13. Using Internal Compiler Resources.

at the appropriate places.

2.3.7 Linkable Object Modules

The HALE meta-assembler produces relocatable microcode modules which are then linked using the HILEVEL Technology linker program.

2.4 Support Cascadable Microprocessors

The decision to use HALE as an intermediate language has had the severist consequences in the requirement to support cascadable microprocessors. This requirement involves essentially multiplying the microword produced for one processor by some number n representing the number of cascaded processors. While this was not difficult to achieve in the Syntax and Microcode Compilers, the maximum HALE word width represents a significant limitation.

2.5 Dynamic Memory Allocation

This concept was originally based on a misconception about the architecture of the IBM/AT and the operation of the MSDOS operating system. The original concept was based on the misconception that the MICROSOFT "C" compiler runnig on MSDOS allowed easy access to memory above the 640K limit. This, of course, proved to be wrong.

We were forced to develop an approach similiar to the way RAM Disks are implemented. This involves paging data in and out of Extended Memory. Using MSDOS interrupts called through "C" functions data is moved in or out of Extended Memeory as required. This requires maintaining page counters and page pointers down in local memory.

Also we were required to develop an appropriate page replacement strategy. We finally settled on leaving the first and last pages resident in local memory, while paging intermediate pages in and out as required. This has several advantages which are based on the construction of our symbol tables and hash tables. After hashing a symbol, we are left with a pointer into a list of linked lists. All of these linked lists begin on the first page. Any symbol not found in the symbol table is inserted at the end of the list, which is always on the last page.

WARNING: Since this approach is similiar to RAM Disks, it is not suprising that all data stored in RAM Disks is subject to be destroyed unless care is taken addressing this problem.

a. One solution might be to not use RAM Disks.

b. Another solution might be to locate the RAM Disk above the two megabyte boundary and reserve the second megabyte for the microcode compiler.

2.6 Optimize Compilers For Speed

The original concept was to identify certain code segments which were consuming large percentages of the processing time. Once these segments where identified, we were going immediately into an assembly language recode of those areas. This approach was modified considerably.

Asembly language routines where coded and inserted into the two compilers. An interrupt was used to stop the program periodically, and a table of location counters was incremented based on where each interrupt occured. After a run through each compiler, the location counter tables where dumped to a file for further analysis.

The analysis of this data showed that optimizing the syntax compiler was not worth while short of recoding the whole program. As a consequence, the Syntax Compiler was not modified. We decided to focus all of other recoding effort on the Microcode Compiler. Recoding in assembly language carries a heavy penalty in terms of reliability and maintainability. As a consequence, it was decided initially to try restructuring the "C" program in order to obtain the required increase in speed. This restructuring was performed with the result that the Microcode Compiler is now running in times comparable to most meta-assemblers.

2.7 Generate Design Modifications

One of the most substantial design modifications involved converting the compilers to "semantic action machines". We took this to mean that the compilers would be designed to handle a wide range of semantic actions and would be designed to facillitate incorporating new semantic actions in the future. This change reflects our intensions to push the level of the languages supported to higher levels, and to support a much wider spectrum of application than just microcoding. Several suggestions have been recieved for enhancements along these lines e.g.

a. Allow creating a label from any arbitrary symbol. This would be extremely useful when the user is creating subroutines or functions.

b. Allow creating variables and assigning storage classes at compile time. Currently these variables are assigned a fixed storage class at syntax definition time.

c. Process (for example) SIMSCRIPT pseudo operations in programs. This would allow the use of this compiler for the rapid prototyping of ultra high level simulation languages.

2.8 Make Code Changes

Code modifications where intended to adhere to standard software engineering practices. We tried to avoid the temptation to patch in changes and cutting other corners. As a consequence, there were few problems with our level zero testing (software debug at the lowest level).

2.9 Generate Test Cases

Test cases were generated to satisfy three requirements: to debug the program; to perform the compiler timing studies; and to demonstrate the capabilities of the microcode compiler.

The debugging effort was automated as much as possible to encourage and facilitate frequent testing. Both a test file (or files) and an answer file were created. DOS commands were then issued from a ".BAT" file to automatically run a test case and verify the results.

WARNING: Inspite of these efforts, no effort was made to test these programs to any standard appropriate for a commercial product or to a standard adequate for a military deliverable item.

Test files were constructed for use in the timing studies. The Syntax compiler was timed with one large and one small test case, while the Microcode Compiler was timed with a large, a medium, and a small test case for each of the Syntax Compiler output files.

Our original goal was to be as fast as the state-of-the-art meta-assemblers. With our current approach, this was impossible. Since the compiler produces a HALE source file which must in turn be processed by the HALE meta-assembler, we were forced to include the assembly times in our analysis.

The third type of test case was designed to demonstrate the expressive power and flexibility of the Microcode Compiler. Various languages were developed. These languages were realistic in the sense that the were designed to solve certain problems; they were complete only to the extent that they solved the required problems. Some languages were generated to support the customers in-house microprogrammable computers and were intended to prove the Microcode Compiler's ability to support a wide range of machines.

2.10 Integrate Software Modifications

Perhaps the biggest problem we encountered was involved with the interface between HALE and the PENGUIN SOFTWARE Microcode Compiler. HILEVEL Technology provided us with a set of documentation for a new version of HALE. Unfortunately, this version does not seem to have ever been developed. This required back tracking to an earlier version of HALE which was in fact released. Our Microcode Compiler now works with the latest versions of the HALE meta-assembler.

WARNING: Our Microcode Compiler is known not to work with the out dated versions of HALE. In particular, the HALE version must be able to handle labels on empty sentences i.e. sentences which do not contain DEF'ed statements.

3.0 STATUS OF ACCOMPLISHMENTS

During this period, a full scale retargetable microcode compiler was specified, designed and constructed. The purpose of this development effort was to demonstrate the feasibility of our approach to automated microcode development. At completion of this feasibility study, we had produced a fully operational microcode compiler and had applied it to several different test cases for evaluation.

3.1 Develop Operational Concept

An operational concept was developed which integrated our microcode compiler into the already existing HILEVEL Technology HALE meta-assembler microcode development system. This approach has several advantages. By outputing HALE as our intermediate level language, we are able to fully exploit the features and capabilities of that state-of-the-art meta-assembler. Some of these features include expression evaluation, relocatable microcode, and linkable microcode.

3.2 Upgrade Existing Software

The design of our final product was based on the design of the Phase I prototype. The development of this prototype unfortunately involved generating a lot of kludges which reduced reliability and maintainability. The entire program was reviewed and recoded as necessary in line with modern software engineering practices.

3.3 Output HALE Source Code

The microcode compiler now outputs ASCII strings of HALE source code instead of a binary object file. At first glance the may appear to be a step backward, and it is. This approach however does have several advantages. In addition to adding all of the power of the HALE development system to our microcode compiler, it also generates output in a form which can be easily modified by a microprogrammer.

3.4 Support Cascadable Micrprocessors

The microcode compiler now contains the option of expand the microword by some operator input factor in order to accomodate cascadable microprocessors. Unfortunately this capability is somewhat limited by the need to process our output through the HALE system which has a limited microword wide.

3.5 Dynamic Memory Allocation

As the need for larger and larger microprograms continues, any microcode compiler is in danger of running out of main memory and becoming obsolete. In an attempt to extend the life of our microcode compiler and investigate the last remaining new algorithm, we included the capability to use extended memory where available and specifically permitted. Various predefined large arrays can be located in an open ended manner in otherwise useless extended memory.

3.6 Optimize Compilers For Speed

It was ultimately determined that our microcode compiler was too slow. An analysis of CPU time useage showed that only the microcode compiler (and not the syntax compiler) needed to be accelerated. While it was originally intended to recode certain critical functions in assembly language, this approach was finally rejected as too unreliable, and difficult to maintain. What we did was use our timing analysis to guide a restructuring of the "C" code.

4.0 CONCLUSIONS AND RECOMMENDATAIONS

4.1 Conclusions

A new and powerful approach to microcode development has been investigated at PENGUIN SOFTWARE, Inc. under the contract DAAD07-87-C-0119 sponsored by White Sands Missile Range. This approach has required the development of two prototype compilers. The first compiler is used to design and define a machine-dependent language for a particular target machine. The definition of this language is contained in a syntax and semantic definition file. This syntax file is inverted and input to the microcode compiler. This microcode compiler then functions somewhat as a HIGH LEVEL Technology HALE preprocessor which converts the application program into an HALE source program.

4.1.1 Tasks Completed and Objectives Attained

All proposed tasks have been completed and all technical objectives have been attained resulting in a demonstratiion of the feasibility of our proposed approach.

4.1.2 Low Risk Approach

After running several test cases during the software debug, software validation, timing study, and the demonstrations, we have gained confidence not only in the correctness of the software but also in the validity of our approach. This being a new approach, it had several conjectured disadvantages. We have found that there were indeed no undeveloped algorithms waiting to trip us up. We were able to proceed to a full up working production-quality microcode development system.

4.1.3 High Speed

In the real world, program speed is certainly an important issue. We were concerned that the microcode compiler would be non-competitive with state-of-the-art microcode development systems because of the added algorithm complexity. Our approach involved using proprietary algorithms and data structures developed here at PENGUIN SOFTWARE, Inc. over the past five years. The result is that the existing microcode compiler runs in times more than competitive with existing meta-assemblers.

4.1.4 Graceful Degradation

All commercially successful meta-assemblers have at least one capability in common: This is the capability to support any machine that may be designed. Using the traditional approach with microcode compilers. It is easy to envision target machine designs that would render a traditional microcode compiler uselessly inefficient. The complexity and sensitivity of the resource allocation schemes and the microcode compaction algorithms leave these compilers extremely vulnerable to total failure.

With our approach, a situation involving total failure due to target machine complexity is avoided. In the worst cases, the high-level language supported will fail to meet user expectations with respect to the level of the language. In this case the greatest concern has to do with how successfully low level hardware design features can be incorporated into a high level language. This is a small issue compared to a case were the microcode compiler simply no longer works or produces unacceptably inefficient microcode.

4.1.5 Language Level

PENGUIN SOFTWARE's microcode compiler supports some popular and important language features. These features include the common arithmetic expressions appearing in languages like FORTRAN, "C", and ADA. In addition, our compiler supports semantic actions which allow the definition of various "structured" programming constructs e.g. DO...WHILE, REPEAT...UNTIL, IF...THEN...ELSE...ENDIF, CASE, etc.

PENGUIN SOFTWARE's microcode compiler does not achieve the language level of "C", and ADA. We fail to achieve this capability in two important areas.

Firstly, there is resource allocation. Typically in high-level languages, the programmer has the ability to either specify resource allocation at compile time or have the compiler allocate resources automatically. With our compiler, resources must be allocated during language definition or at compile time: there is no automatic allocation of resources.

Secondly, there is code compaction. An ideal microcode compiler would automatically produce compact and efficient microcode. While our compiler is capable of producing good microcode, it does not do it automatically. Essentially, our compiler produces efficient microcode by a combination of designing efficiency into the high-level language and by requiring the application programmer to apply knowledge about the target hardware.

4.1.6 Comparing The Two Approaches

There are several advantages to having a microcode compiler support a machine independent language e.g "C", or ADA. These languages free the programmer from needing any hardware knowledge and drastically reduces the skill level needed in the programming staff. In addition, there is a large supply of programmers already skilled in using the common machine independent languages. Finally, a language written in "C" (for instance) automatically provives a microcode simulator. This follows from the fact that "C" compilers are common and a "C" programm can be compiled and ran on practically any microcomputer.

The main advantage of our approach is that it works. Our approach to automated microcode generation was initially considered to be low risk in two important respects.

a. During our development effort, there were no high risk algorithms to be developed. We were quickly able to generate working compilers which were then refined through several stages of improvement.

b. For the end user, our approach is much more robust than the competition. This follows from the current state of code compaction algorithms.

The result is that we have developed a microcode development system which we can use and evaluate in the short term. In the long term, when the code compaction and resource allocation algorithms are perfected, the machine independent language approach based on ADA or "C" will be far superior to the machine dependent approach.

4.2 Recommendations

We have two recommendations that are contrary to conventional wisdom.

4.2.1 Rethink Expectations

The advantages of automated microcode generation result mostly from the readability of high-level languages. Code that is easier to read is easier to write, debug, test, and maintain. High-level language microcode can more than pay its way with these advantages. Unfortunately, many people have unrealistic expectations about productivity improvements resulting from code expansion.

QUESTION: Many high-level language statements expand into ten assembly language statements. Likewise, many assembly language statements expand into ten microcode statements. Can we reasonably expect high-level language statements to expand into one hundred microcode statements?

Our answer, with a few obvious exceptions, is no. The fact of the matter is that the code expansion mentioned above results mostly from mismatches between the hardware structure and the languages being supported. Our examples of "C" statements and their handcoded microcode equivalents show that a well matched machine and high-level language will result in a one-to-one translation of high-level language to microcode very similar to a meta-assembler with MACRO capability.

4.2.2 Design Hardware To Match Compilers

There is one fact about high-level languages which seems to be consistently overlooked: modern computers are designed to support these compilers. The microcoding community must accept the obvious consequences of this fact: hardware design requirements will have to include provisions for supporting microcode compilers.

APPENDIX A

PROJECT OVERVIEW

A1.0 MAINSTREAM APPROACH TO AUTOMATED MICROCODE GENERATION

Most efforts at automated microcode generation have turned toward high-level languages and hence microcode compilers. In addition, all research efforts appearing in technical journals have adopted a similiar approach to developing a micocode compiler. We call this approach the "mainstream" approach.

The mainstream approach to automated microcode generation starts with a high-level language capability and attempts to work toward efficient microcode. This approach is based essentially on the existing compiler technology(see Fig. A-1). The main idea is to start with some appropriate machine-independent language as a target language, design a compiler to expand source code (written in the target language) into some intermediate language, expand the intermediate language into microcode, allocate resources in the microcode, and finally process the microcode in an attempt to improve the efficiency of the microcode to a satisfactory level.

A2.0 PENGUIN SOFTWARE'S APPROACH

PENGUIN SOFTWARE's approach to automated microcode generation starts with efficient microcode and works towards a high-level language capability. Rather than start with existing compiler design technologies, we start with existing meta-assembler capabilities. To the typical meta-assembler capabilities, we have added more semantic actions upgrading the level of the languages supported.

Our approach requires two compilers: the first compiler for the language definition, and the second compiler for the microcode generation (see Fig. A-2). A language Descriptiion File is prepared which defines the target language. This file is compiled to produce a set of drive tables (Syntax Definition File) which define the target language for the Microcode Compiler. Finally, the application program (written in the target language) and the drive tables are processed by the Microcode Compiler which in turn produces the microcode Object Program.

A3.0 state of microcode compilers

Figure A-3 attempts to show our analysis of the current state of the Microcode Compiler effort. The upper left hand corner represents the current state-of-the-art in compiler technology: very high-level language and very low efficiency microcode. The lower right hand corner represents the current state-of-the-art in meta-assembler technology: very high efficiency microcode and very low-level languages.

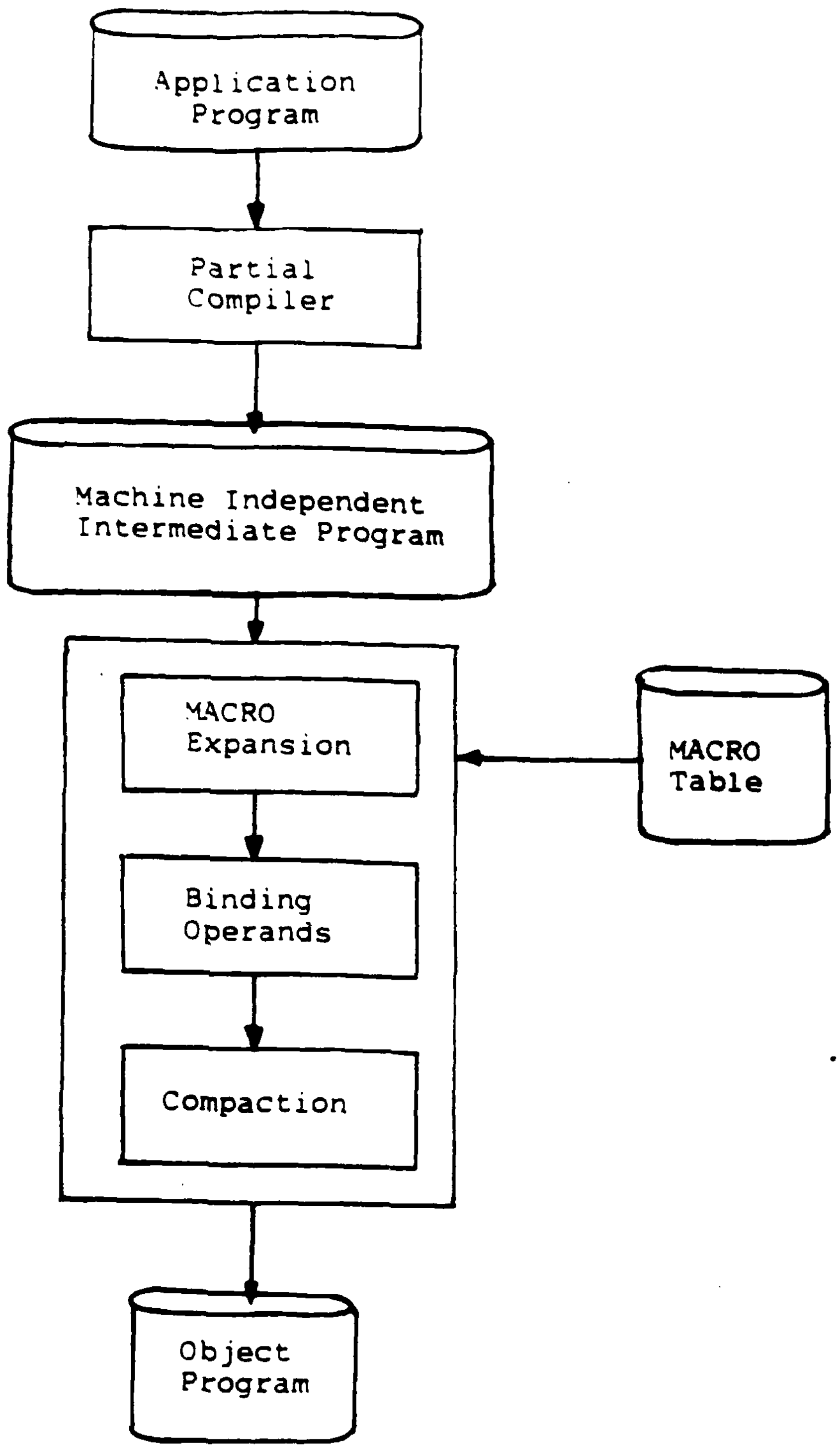


Figure A-1. MAINSTREAM APPROACH

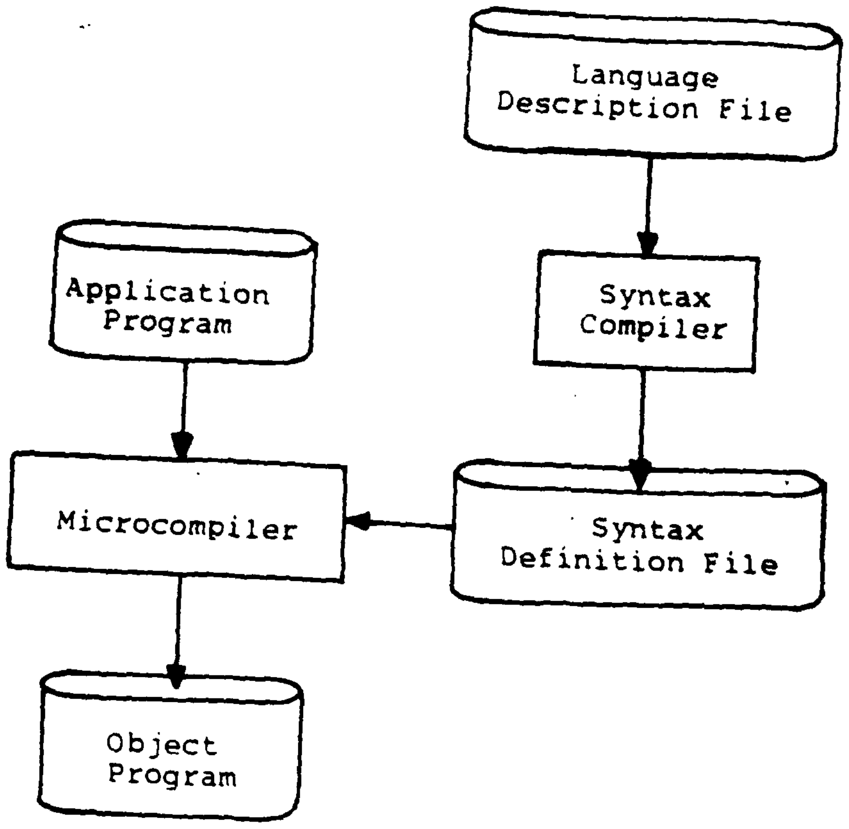
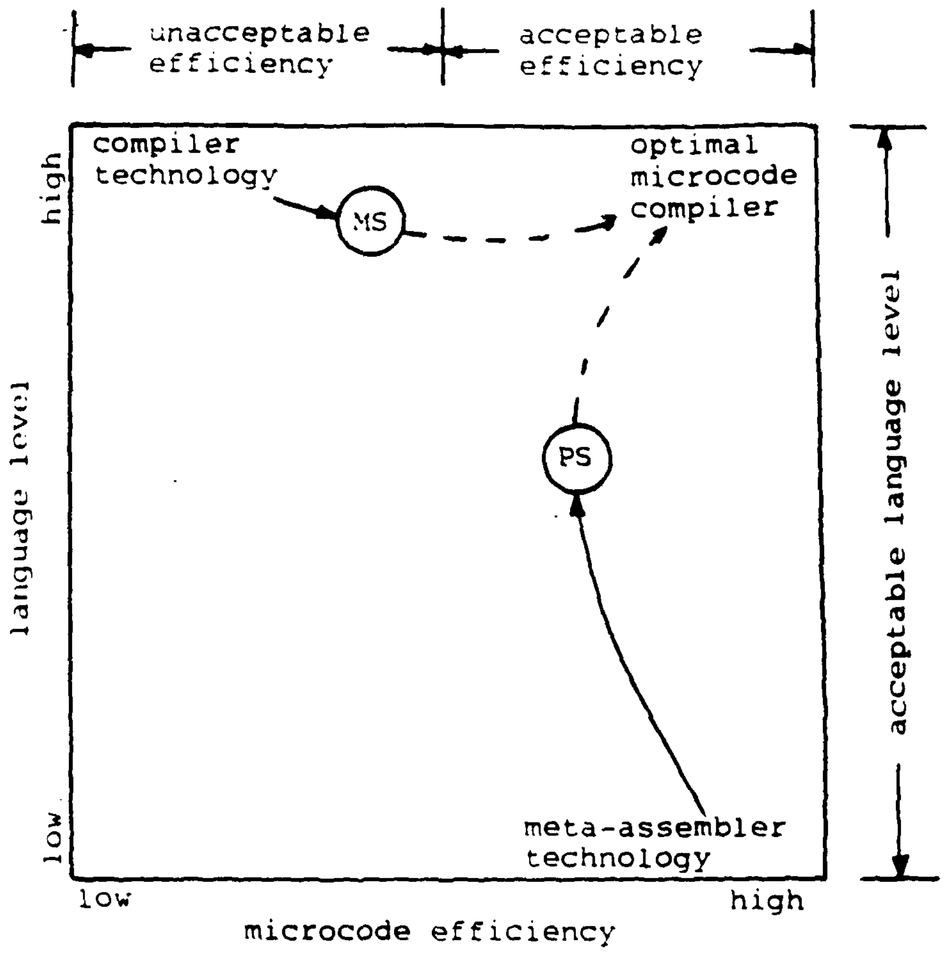


Figure A-2. PENGUIN SOFTWARE'S APPROACH



MS = state of Mainstream
PS = state of PENGUIN SOFTWARE

Figure A-3. STATE OF AUTOMATED MICROCODE

The mainstream approach (represented by "MS") is essentially a standard compiler with a microcode compaction algorithm added on at the end. This allows a substantial improvement in efficiency but still fails to achieve an acceptable level of performance. The mainstream approach attempts to capitalize on the substantial benefits of providing a machine-independent language. The success of this approach is dependent on the development of acceptable resource allocation and code compaction algorithms. Without these algorithms, this whole approach fails catastrophically and is not commercially useful as a microcode development tool. After decades of research, useful algorithms are still not available.

The PENGUIN SOFTWARE approach (represented by "PS") is essentially a standard meta-assembler with additional semantic actions. This allows a substantial improvement in the language level without necessarily sacrificing microcode efficiency. When this approach fails, it fails gracefully. The success of our approach, for a particular application, is determined by the level of the language supported and the level of customer satisfaction with that language. The minimum level of the language and the minimum level of efficiency of the microcode need never be any worse than for the underlying meta-assembler. Since this meta-assembler represents the current state-of-the-art, the Microcode Compiler should never fail to be a useful development tool.