A high-level microprogramming language (MPL)

by R. H. ECKHOUSE, JR.

S.U.N.Y. at Buffalo
Amherst, New York

INTRODUCTION

As late as 1967, a prominent researcher reported to his organization that he believed a successful higher-level microprogramming language seemed unlikely. At the same time, other members of the same organization were describing what they termed “A Microprogram Compiler”. Meanwhile, other hardware and software designers, equally oblivious of each other, were generating useful and powerful higher-level languages to assist them in their work. As the reader will see, the stage had been set for the development of a higher-level, machine-independent language to be used for the task of writing microprograms.

The research here reported describes a microprogramming language, called MPL, and includes several aspects of the development and use of such a language. The objectives for the language, the advantages and disadvantages, the work which has preceded the development, and the importance and relevance of developing such a language are considered. Finally, we shall consider this current research, showing some preliminary but very promising results.

HIGHER-LEVEL LANGUAGES FOR MICROPROGRAMMING

The area of microprogramming has opened new possibilities for both software and hardware designers because microprogramming has, to a certain extent, blurred the once clear separation of level between the two. In microprogrammable machines we find hardware circuits that incorporate read-only or read/write memory which can determine both the computer’s actions and its language. It is therefore beneficial to consider the needs of both the hardware designer and the software designer in the development of a microprogramming language.

Objectives

The hardware designer (the traditional microprogrammer) needs the ability to express the relevant behavior and structural properties of the system. The software designer needs the flexibility of a programming system which allows him to describe the procedures by which a machine can execute a desired function. In combining both of these needs, as microprogramming does, we find that a suitable microprogramming language must be one that is high-level, procedural, descriptive, flexible, and possibly machine-independent.

Advantages

A high-level microprogramming language will free the users from such non-essential considerations as table layouts, register assignments, and trivial bookkeeping details. The language will have the obvious benefits of improved programmer productivity, greater program flexibility, better documentation, and more transferability. By providing the necessary tools for the hardware designer, the software designer, and the machine user, this language can be part of a larger system which is viable for all phases of system design: description, simulation, interpretation, and code generation.

Disadvantages

The seemingly obvious (and traditional) disadvantages of utilizing a higher-level language for microprogramming are loss of efficiency, inflexibility, and high cost. The critics cite the need for a high degree of machine usage, tight code and maximum utilization of every bit in a microinstruction as the major factors for ruling out the use of such a language. “Basically, a compiler would generally be forced to compete with a microprogrammer who can justifiably spend many
hours trying to squeeze a cycle out of his code and who may make changes in the data path to do so".1
In light of the larger aspects of microprogramming, the above criticisms seem much less tenable. First, the current users of machines which can be microprogrammed are not only their hardware designers. These users do not wish to exercise the microprocessor to its fullest extent if this leads to "tricky" code, or code that is difficult to write, debug, and test. Instead, these users wish to be able to write their own emulation or application software and be able to use higher-level languages with all of their benefits.

The second point is that there exist scant measurement standards for determining efficiency, flexibility, and cost of the presently used methods. Manufacturers, when asked questions concerning hardware utilization, concurrency, and efficiency, tend to state that "the total core size of the microprogram is only X", or "our machine has achieved the desired speed Y", leaving us puzzled and unenlightened.

The real costs and savings inherent in a microprogrammable machine should not be measured in terms of raw speed or core size alone but must be concerned with the unique flexibility that such a machine offers. When we discover bugs in the virtual system, we know that it is clearly less costly to write and implement new microroutines in a microprogrammable machine than to rewire a non-microprogrammable machine. And when we desire to add new features such as virtual instructions or hardware I/O options, it is again less costly to do so to a microprogrammable machine. Thus it would seem that if a higher-level language can aid this process by further reducing the cost of writing microroutines, then clearly such an approach is viable and well worthwhile. The reader is referred to the work of Corbatò3 for additional discussion of the approach.

A suitable language

In developing a suitable language for writing microprograms, the language developer should ask himself if his language would be new, better, more enlighted or useful than some existing, well known language. Instead of adding to the proliferation of languages, it would be well worthwhile to utilize some existing language, with extentions if necessary, as the basis for the development. Fortunately, an appropriate, higher-level language does exist and can be used to not only write microprograms, but also be used to describe, simulate, and design hardware.

A small dialect of PL/I, akin to XPL, represents a suitably modified language amenable to microprogramming. This paper will report on the on-going effort of the design and use of this dialect, the author's higher-level language for writing microprograms called MPL. Another dialect of PL/I described in CASD4 has already been used to aid in the design of computers (both microprogrammable and not). Thus, the use of a higher-level language has already been demonstrated to be a viable technique for the design, description, and simulation of computer systems, and need not be treated further in this paper.

Importance and relevance

In the process of developing a microprogramming language such as MPL we must be concerned with the relevance of the language. We must find out how effectively the language may be used, and how capable it is in meeting the needs of the user. Thus, performance is a criterion of acceptance and we must be able to demonstrate the ease of producing a meaningful high-level program which can be suitably translated into efficient microcode.

BACKGROUND

Previous work in developing higher-level languages for software and hardware designers is rather extensive. Unfortunately neither side has been concerned with the other, and we find few attempts to reconcile the two. APL is one exception, and its proponents have categorized it as a universal, systematic language which is satisfactory for all areas of application.5 Papers have been written to show how APL can be used to describe hardware, to formulate problems, and to design systems. Another exception, previously discussed, is CASD.4 However, the CASD project has since been disbanded, and no attempt has been made to write the microcode translator discussed in the report of that project.

Systems programming languages

For all of its contributions and contributors, APL does not adequately describe systems programming or microprogramming problems (e.g., timing, asynchronicity, and multiprogramming) without additional explanations in English. In addition, only a subset of APL has been implemented and the whole language remains significant but unimplemented.

Other contributions to higher-level, systems programming languages have included EPL, ESPOL, SYMPL, and IMP. These languages possess block structure, compound statements, and logical connec-
tives which make the job of system design much easier. The MULTICS project\(^3\) and the development of the B5500, with its unconventional "machine language," have demonstrated the successful utilization of higher-level languages to operating systems design.

**Hardware design languages**

Recent papers by hardware designers seem to indicate a strong trend toward the use of higher-level, machine-independent languages for hardware design. The objectives of these papers appear to be:

1. To describe digital machines
   a. Their logic
   b. Their timing and sequencing
2. To simulate digital machines
   a. Verify new designs
   b. Verify new features
3. To have machine translatable, formal, hardware description languages
   a. Supporting (1) and (2) above
   b. To simplify machine design

The objectives have been met to various degrees as evidenced by the work of Metze and Seshu,\(^6\) Chu,\(^7,8\) Darringer,\(^9\) Schlaeppi,\(^10\) Schorr,\(^11\) Proctor,\(^12\) Gorman and Anderson,\(^13\) and Zucker.\(^14\) Much of their work seems amenable in its application to microprogramming, and all of it represents the application of an existing higher-level language structure (FORTRAN or ALGOL) to the hardware specification and design problem.

**Microprogramming languages**

The first evidence of a language structure for writing microprograms appears to be in the work of Husson et al.\(^2\) The authors present their views on the more general concepts for designing a microprogram compiler but they do not have the experience of a working compiler. They discuss a compiler-language which is high-level, procedural, descriptive, and machine-independent. They suggest that such a language will require an intermediary language (some form of an UNCOL) which will allow for the successful generation of a simulator and a machine-dependent interpretation of the microcode.

The authors go on to suggest that there should be compatibility between adjacent, architecturally similar processors or classes of machines. Thus, the compiler-language must permit hierarchical descriptions of the particular machine class.

**Universal languages**

Many of the problems encountered by the hardware and software designers which concern machine-independence are discussed in papers on SLANG\(^15\) and UNCOL.\(^16,17\) The SLANG system is concerned with the basic question, "Is it possible to describe in a machine-independent language processes which in themselves are machine-dependent?" In the papers addressing the UNCOL concept, we find the discussion on whether or not there exists some intermediate language(s) between any problem-oriented language and any machine language, and whether or not the separation of machine-independent aspects of a procedure oriented language from the machine-dependent aspect is feasible.

**THE LANGUAGE AND ITS TRANSLATOR**

The choice of a higher-level, machine-independent language for this research required consideration of several aspects in its development. Some of these aspects and the conclusions to which their consideration lead included:

1. A survey of a representative sample of microprogrammable machines, i.e., how machine-independent or widely applicable is the proposed language?
2. What is the syntax of the language? What are its syntactic elements and how do they relate to microprogramming?
3. What is the objective of the language? Is it ease of translation into efficient microcode, or is it ease of describing application problems which can be converted into microcode?
4. How is translation into microcode performed? If the language is machine-independent, at what stage in its translation is machine-dependence introduced? At what stage do we tailor the code toward the particular microprogrammable machine?
5. How do we evaluate the code produced? How do we know it is correct or good? Is it "concise"?

What follows are answers to these questions, and an analysis of the effects these answers had in dictating the ultimate results.

**Microprogrammable hardware**

The objective in surveying current hardware was to attempt to classify the similarities and differences in the various microprogrammable machines. As expected,
the architectural differences are not overwhelming, and in many cases are manifest more in terms of the "state of the art" technology, than in differences in type of instruction set, testable conditions, types of addressing, etc. Indeed, all of the machines can be classified as classical, Von Neumann in nature with only minor perturbations.

Syntax

The literature abounds with various languages for writing systems programs (MOL-360, BCPL, PL/360, etc.) and for describing and simulating hardware (LOTIS, CASD, Computer Compiler, etc.). In all cases, the syntax is simple and easily translatable into hardware implementable semantics. Such an approach was taken in specifying the PL/I-like syntax of MPL.

Procedures and declarations

As in PL/I, the basic building block of MPL is the procedure. The concepts of local and global, scope of names, etc., have been preserved and represent the block structure of the language.

Declarations of the various data items (including registers, central memory, and events) give the attributes of the items. By use of the PL/I "DEFINE" syntax, register data items are subdivided into their principal parts (i.e., we may declare a virtual 2n-bit register and then define its true constituent n-bit parts).

Data items

There are basically six types of data items. First, there are the machine registers, both true and virtual. Second, there is central and micro memory. Third, there is both local and auxiliary storage which can be similar to the register data type or the central memory data type, depending on its implementation in the actual microprogrammable machine. Fourth, there are "events," unlike events in PL/I, which correspond to testable machine conditions (carry, overflow, etc.). Fifth, there are constants of the type decimal (e.g., 2), binary (e.g., 1011B), and hexadecimal (e.g., OFX). The traditional enclosing quotes around binary and hexadecimal constants may be dropped as long as the constant begins with a digit and ends with a B or X. There are also label constants and symbol constants (or literal constants ala XPL). Finally, there are variables which will take on constant values.

Statements

Assignment statements have been modified in MPL to allow concatenated items to appear on either side of the equal sign. Thus, the concatenation of two registers R1 and R2 becomes R1//R2. This newly defined, double length register can be used logically as if it actually existed such as:

\[ R1//R2 = R1//R2 + 2; \]

Additional binary and logical operators have been added or modified in MPL and include:

- \( a \text{ .RSH. } b \) Shift a right b places
- \( a \text{ .LSH. } b \) Shift a left b places
- \( a \wedge b \) a and b
- \( a \lor b \) a or b
- \( a \oplus b \) a exclusive-or b

Finally, the IF statement is able to test an EVENT previously declared. Thus, a convenient means exists for a transfer on carry, overflow, etc.

Objectives of the language

With microprogrammable machines, two emulation objectives are commonly identified. First, the hardware may be used to emulate a particular system (S/360 on the IBM 2025, 1130 on the Meta IV). Second, the microprogrammable hardware may be used to emulate a particular environment (SNOBOL4, a banking system, etc.). Traditionally, the former objective requires tight microcoding with efficiency of the produced microcode the end goal. The latter objective requires a good run-time environment which can support, through emulated primitives, those features peculiar to the application environment.

The first objective generally requires an efficient translator, with various techniques of optimization, including the use of an intermediate language. The second generally does not require an intermediate language, since in most cases the primary language can be directly translated into the emulated primitives implemented on the host machine.

Translation procedure

In this research the use of an intermediate language called SML has greatly facilitated the translation process from a higher-level machine-independent language into microcode. The basis for this intermediate language can be found in an early paper by Melvin Conway on
the use of an UNCOL. SML-to-microcode translators have been written for the INTERDATA 3, and are capable of producing "compact" code (see Appendix A). In addition, the translation algorithm for converting the MPL code into microcode allows for multiple precision data manipulations, a feature very common to emulator programs. The result is that the process of emulating a 2n-bit word machine on an n-bit microprogrammable machine is easily done at the highest level (MPL level) in a most natural fashion.

The general organization of the translator is shown in Figure 1. Source code is initially translated into SML in phase 1. At the same time, a dictionary is constructed for later use in phase 3. Items entered into the dictionary include real and virtual registers, testable conditions, literals, and other items DECLAREd. Although the SML produced is machine-independent, the dictionary is not in that it relates virtual data items to their real equivalents.

Phase 2 of the translator produces virtual object code from the SML input. The code is virtual in the sense that the operands of machine instructions may be virtual data items (concatenated registers, multiple precision data items, literals, etc.) and need not be of equal widths.

The conversion from virtual object code to object code is resolved in phase 3. Operands of unequal or virtual nature must be converted to true machine instructions. Literals, immediate operands, virtual and concatenated registers must be looked up in the dictionary in order that their virtual representations may be replaced by their object representations.

In general, phase 3 will cause additional lines of object code to be generated. This code results from the conversion of virtual operands into true operands.

Preliminary evaluation and future work

The current translators from SML to microcode are written in SNOBOL4. They are capable of producing microcoded routines from SML which closely resembles the same code supplied by the manufacturer (see Appendix A). The whole process of coding the emulation routines has been made considerably easier by using MPL, and removes much of the busy work required in writing microprograms from the programmer's shoulders.

There are certain drawbacks, however, in using any systems languages such as MPL. In particular, when one allows the use of every facility at the highest level, conflicts may arise (such as that which can occur in any high-level language where assembler code may be generated in-line), and indiscriminate use of those facilities may lead to reasonable but unexpected results. This seems a small price to pay in terms of the original objectives set forth in the section on microprogramming languages.

The original objectives of a high-level, procedural, descriptive, flexible, and possibly machine-independent language for writing microprograms have been met so far. The entire process from the higher-level language to the microcode has been considered, and the feasibility seems clear. As Husson points out in his book, the value of this project is in its use for:

1. Designing
2. Debugging
3. Translating

In each case, the programs must be organized, written, tested, and debugged. At each level, the organization and flexibility provided are enough to justify the existence of MPL.

Future work will concern itself with further refinements to the language, and with its application to a multiple data-path machine such as the IBM 2050. Techniques for both local and global optimization of the code produced will also be considered.

APPENDIX A

Figure 2 is a portion of the INTERDATA 3 emulator written in MPL. The emulated environment is that of a simplified 360, and Figure 2 shows part of the initializing routine (to fetch the PSW) and the instruction fetch and decode routines.

In the outer procedure "INITIAL", we find the ex-
**Figure 2—An example using the PL/I-like syntax**

**Figure 3—PL/I-like code translated into SML**

**Figure 4—Second phase of the translation**
The inner procedure "PHASE 1" represents the instruction fetch, location counter update, and op-code format recognizer routines. In it can be found two features not normally found in PL/I. First, the binary operators .RSH. and .LSH. have been added to represent right-shift and left-shift respectively. Additional operators such as exclusive-or have also been included in the syntax since they occur frequently in the instruction sets of microprogrammable machines.

Second, the occurrence of the "events" CARRY, SNGL, CATN, TRUE, etc., in the IF statements are taken to imply that special conditions within the machine can be tested directly. The actual relationship of the event to the physical hardware is specified in the DECLARE statement.

Figure 3 shows the same code as Figure 2, but the translation into SML can be found interspersed on the right-hand side of the figure. Operations are denoted by an "X" followed by parentheses enclosing the name of the operation. Arguments needed for operations must first be loaded into argument or A-registers. Results of operations are placed in result or R-registers. Temporary or T-registers are available for intermediate results. Finally, literals are indicated by preceding their names (values) by an asterisk.

Figure 4 represents the output of phase two of the translator. This is the traditional and more difficult code emission phase of the translation process. The results are not true INTERDATA code, however, and must go through another phase to relate the actual facilities and data-path widths to the virtual facilities and data paths which the programmer assumes.

Figure 5 is the output of the third phase of the translator. The code produced here is actual INTERDATA code in assembler format. The output has required a dictionary to relate the virtual and physical registers, data-paths, etc., to each other. Construction of such a dictionary is accomplished in the MPL to SML phase of the translation, and is facilitated by the declarations in the MPL code.

APPENDIX B

The INTERDATA 3 is a very fast, simple and uncomplicated machine. Control instructions reside in a Read-Only-Memory (ROM) that is 16-bits wide. Thus, the microinstructions of the machine are 16-bits long. However, the data paths are only 8-bits wide.

Microinstructions for the INTERDATA are somewhat similar to the instructions for a two address (register-to-register) machine with an accumulator (AR). The instruction types include:

<table>
<thead>
<tr>
<th>L</th>
<th>Load</th>
<th>X</th>
<th>Exclusive Or</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Add</td>
<td>B</td>
<td>Branch On Condition</td>
</tr>
</tbody>
</table>

From the collection of the Computer History Museum (www.computerhistory.org)
The four formats for the ten instructions of the machine are:

<table>
<thead>
<tr>
<th>op</th>
<th>destination</th>
<th>source</th>
<th>modifiers</th>
</tr>
</thead>
</table>

for op codes A,S,N,O,X,L

<table>
<thead>
<tr>
<th>op</th>
<th>destination</th>
<th>data</th>
</tr>
</thead>
</table>

for immediate instruction forms as above

<table>
<thead>
<tr>
<th>op</th>
<th>test or command</th>
</tr>
</thead>
</table>

for test or command instructions

<table>
<thead>
<tr>
<th>op</th>
<th>C V G L</th>
<th>address</th>
</tr>
</thead>
</table>

Thus an add instruction might look like:

```
A MAH,R1
```

where the contents of the source register R1 are added to the contents of the accumulator AR and the result is stored in the destination register MAH.

Modifiers for the various instruction types include:

- **NA**: AR is not added to the source register.
- **SR**: Shift the contents of the source register right one bit and then perform the operation.
- **SL**: Shift left as above.
- **CI**: If the carry flip-flop is on, add a one to this instruction.
- **CO**: Set the carry flip-flop if a carry is generated out of the most significant bit.
- **NC**: No carry.

The assembler allows literals to be specified as hexadecimal constants and labels. Since labels may be 16-bit values, their high and low parts are specified by prefixing the literal by an H or L respectively.

REFERENCES

AFIPS-171, (18), p. 402

1 S G TUCKER
   Microprogram control
   IBM Systems Journal Volume 6 pp 222-241 1967

2 S S HUSSON
   Microprogramming: Principles and practices
   Prentice-Hall Engelwood Cliffs New Jersey pp 125-143 1970

3 F J CORBATO
   PL/I as a tool for systems programming
   MIT Project MAC Memorandum M-378 1968

4 E D CROCKETT et al
   Computer-aided system design
   AFIPS Conference Proceedings Fall Joint Computer Conference 1970

5 K E IVERSON
   Programming notation in systems design
   IBM Systems Journal Volume 2 pp 117-128 1963

6 G METZE S SESHU
   A proposal for a computer compiler

7 X CHU
   An ALGOL-like computer design language
   Communications of the ACM Volume 8 pp 607-615 1965

8 Y CHU
   A higher-order language for describing microprogrammed computers
   University of Maryland Computer Science Center
   Technical Report 68-78 College Park Maryland 1968

9 J A DARRINGER
   The description, simulation, and automatic implementation of digital computer processors

10 H P SCHLAEPPI
    A formal language for describing machine logic, timing, and sequencing (LOTIS)

11 H SCHORR
    Computer-aided digital system design and analysis using a register transfer language

12 R M PROCTOR
    A logic design translator experiment demonstrating relationships of languages to systems and logic design

13 D F GORMAN J P ANDERSON
    A logic design translator
    AFIPS Conference Proceedings Fall Joint Computer Conference pp 251-261 1962

14 M S ZUCKER
    LOCS: An EDP machine logic and control simulator
15 R A SIBLEY
The SLANG system
Communications of the ACM Volume 4 pp 75-84 1961

16 P R BAGLEY
Principles and problems of a universal computer-oriented language
Computer Journal Volume 4 pp 305-312 1962

17 T B STEEL
A first version of UNCOL

18 D J FRAILEY
Expression optimization using unary complement operators
SIGPLAN Notices Volume 5 pp 67-83 1970

19 M E CONWAY
Proposal for an UNCOL
Communications of the ACM Volume 1 pp 5-8 1958