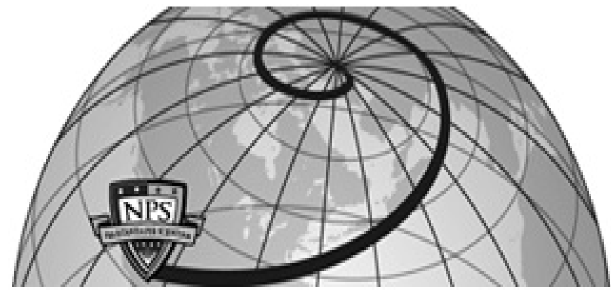
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

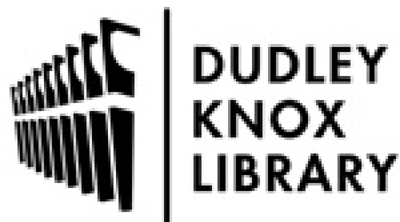
Faculty and Researcher Publications Faculty and Researcher Publications

1974

High-level language simplifies microcomputer programing

Kildall, Gary A.

Electronics, June 27, 1974.
http://hdl.handle.net/10945/45140



Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

High-level language simplifies microcomputer programing

Just as Fortran and Basic sharply reduce the time and effort required to program large computers, so Intel's PL/M eases the programing of systems based on LSI microprocessors; here are step-by-step directions

by Gary A. Kildall,
Naval Postgraduate School, Monterey, Calif.

☐ The microcomputer is being applied to more and more tasks that are not economically feasible for a minicomputer, with its larger instruction set and higher speed and cost. Although the microprocessor is slower than the central processor of a minicomputer, it can easily perform many tasks that are complex enough to require extensive digital processing.

What's more, microprocessors, which serve as central processors of microcomputers and are generally made with MOS large-scale integration, are constantly attaining higher speeds and higher circuit density per chip. As the capabilities of microcomputers are being ever extended, programing aids are being developed to simplify their use, while minimizing design and development time. These aids sometimes require use of a larger computer; when this is the case, they can be used either on commercial time-sharing networks or on a user's own large in-house computer.

The microcomputer may be viewed as a ROM-driven LSI logic chip because the microcomputer can execute complicated sequences of instructions stored in an external memory. Thus, the microcomputer chip connected to a read-only memory containing the proper data can appear to be a single custom chip. In this way, the system designer can substitute microcomputer programing for traditional hard-wired logic design or custom chip fabrication, gaining advantages in reduced development time, ease of design change, and reduced production costs.

The application of microcomputers points up the common ground between software and hardware designers. While software-system designers can use microcomputers most effectively when they are aware of the hardware environment, the hardware designer is well advised to learn the basic techniques of the programer.

These techniques include how to use assemblers, compilers, and processor simulators, which are effective tools in developing and debugging large and small microcomputer programs. This article introduces these programing tools to the hardware designer and specifically examines the advantages of the PL/M language, which make possible rapid design of systems around the MCS-8 microcomputer, made by Intel Corp.

The MCS-8 is based on the 8008 microprocessor, one of a new class of devices being offered by several manu-

1. Symbolic. This simple program for choosing the larger of two numbers takes nine lines of code in symbolic or assembly language, but typically only one line in a higher-level language, such as PL/M.

LABEL	INSTRUCTION	COMMENT
TEST	SHL B	LOAD ADDRESS OF B
	LAM	LOAD B INTO ACCUM
	SHL A	LOAD ADDRESS OF A
	CPM	COMPARE B WITH A
	JFC L1	JUMP TO L1 IF B ≤ A
	LAM	LOAD A INTO ACCUM
L1	SHL C	LOAD ADDRESS OF C
	LMA	STORE ACCUM INTO C
	END	END OF PROGRAM

Closing the loop

Readers who would like to discuss the PL/M language with Mr. Kildall may call him at (408) 646-2240 during the week of July 15, between the hours of 9 a.m. and 12 noon, Pacific Daylight Time.

facturers as a result of recent advances in semiconductor electronics. The PL/M programing aid is a good example of the service that these manufacturers can offer to simplify the use of their products.

Minimizing software costs

Like other programing tools, the PL/M approach automates the production of programs to counteract the rapidly increasing cost of software production at a time when hardware costs are decreasing. And, in addition to rapid production turnaround, the programs can be fully checked out early in the design process. What's more, the self-documentation of PL/M programs enables one programer to readily understand the work of another, which dramatically reduces program-maintenance costs and provides transportability of software between programers and to other Intel processors as they are introduced.

Additional cost reductions will also result from standardization of parts and modules, and alterability of the final program often outweighs benefits of random-logic designs or custom-chip fabrication.

The PL/M compiler, which is another program, translates the PL/M program into machine language. This compiler, which can be run on a medium- or large-scale computer, is available from several nationwide time-sharing services.

Last but not least, PL/M programs can be recompiled as improved optimizing versions of the compiler are released, as Intel has recently done. A recent revision of the PL/M compiler, for example, makes possible reduction of generated code by about 15%.

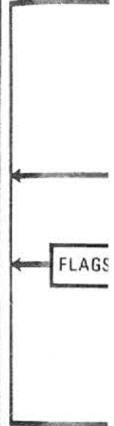
Although PL/M requires a cross-compiler—one that runs only on a larger machine—a resident compiler that uses the microcomputer itself to produce its programs is technically feasible with the advanced state of microcomputer development and today's inexpensive peripherals. Such a compiler would require several passes to reduce a PL/M source program to machine language, using the developmental system itself, and eliminating the need for large-system support.

A program for the Intel 8008 microprocessor is a sequence of instructions from its normal instruction set (see "Hardware for PL/M," p. 105) that performs a particular task. Given no programing aids, the designer must determine the machine codes that represent each of the instructions in his program and store these codes into program memory. This approach to programing quickly becomes unwieldy in all but the most trivial projects.

Nearly all manufacturers of microprocessors (and mini- and maxicomputers as well) provide symbolic assemblers—programs that ease the programing task by eliminating the need to translate instructions manually into machine-readable form. The designer can express his program in terms of mnemonics, which are abbreviations that suggest individual instructions. Then the assembler translates each mnemonic instruction into its binary representation.

Symbolic addresses

In addition, the programer can refer to memory locations by symbolic name, rather than actual numeric address; the assembler translates these, as well as the instructions. The assembler usually runs on a larger computer, although both Intel Corp. and National Semiconductor Corp. have assemblers that run directly on their microcomputer-based development systems, and symbolic programs for Rockwell microcomputers can be assembled on a machine built around that unit by Applied Computer Technology Inc. The assembler requires significantly less development and check-out time than manual translation, and there are fewer coding errors.



```
LINE   STATEMENT

 1     DECLARE MESSAGE DATA ('WALLA WALLA WASH'),
 2         (CHAR, I, J, SENDBIT) BYTE;
 3
 4     /* SEND EACH CHARACTER FROM MESSAGE VECTOR TO TELEPRINTER */
 5         DO I = 0 TO LAST(MESSAGE);
 6         CHAR = MESSAGE(I);
 7         SENDBIT = 0;
 8
 9         /* SEND EACH BIT FROM CHAR TO TELEPRINTER */
10             DO J = 1 TO 11;
11             OUTPUT(0) = SENDBIT:
12             CALL TIME (91); /* WATTS 9.1 MS */
13             SENDBIT = CHAR AND 1;
14             /* ROTATE CHAR FOR NEXT ITERATION */
15             CHAR = ROR (CHAR OR 1, 1);
16             END;
17         END;
```

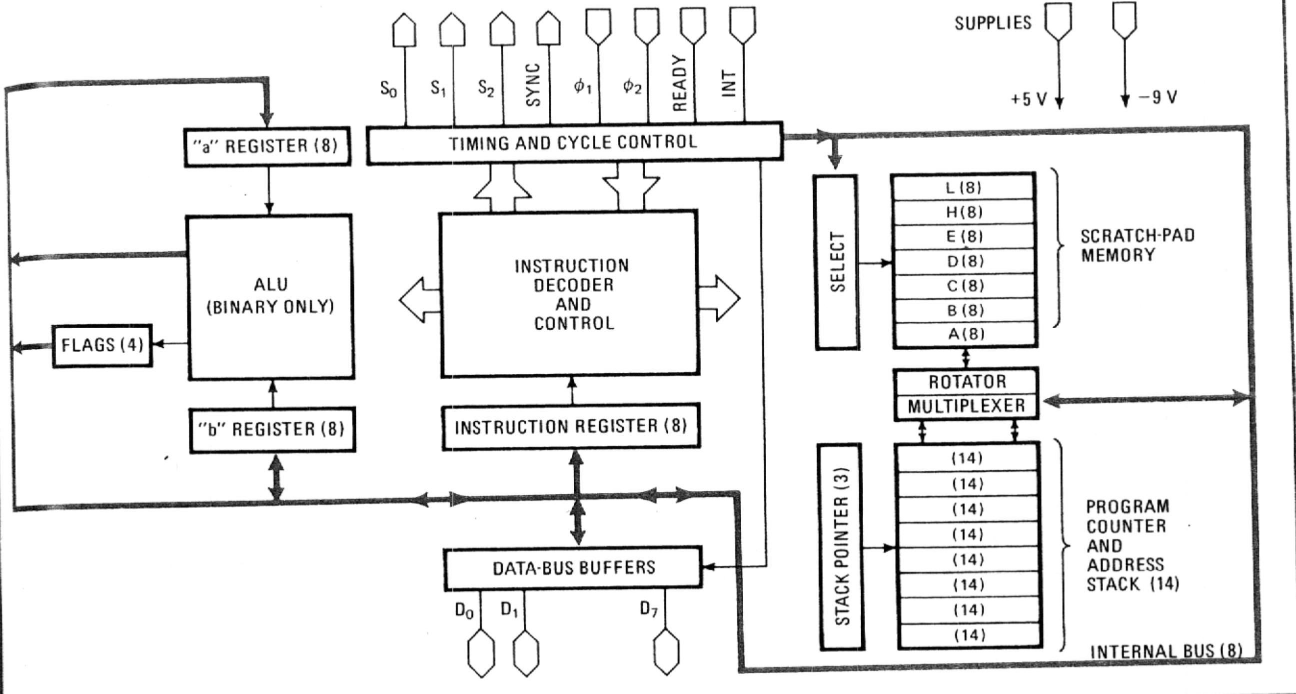
2. Serial sender. To print a short message on a Teletype, this routine in PL/M transmits 11 pulses at 9.1-millisecond intervals for each character in the message, stopping after the last one. The pulse train consists of one start pulse, eight data pulses, and two stop pulses.

Hardware for PL/M

The Intel MCS-8 microcomputer consists of the 8008 microprocessor plus a collection of standard read/write and read-only memories and shift registers. The 8008 is a single-chip MOS device with

- 8-bit parallel word size
- Seven 8-bit general-purpose registers
- 16,384-word address capability, in either read-only or read/write memory
- Up to 32 8-bit latched input and output ports

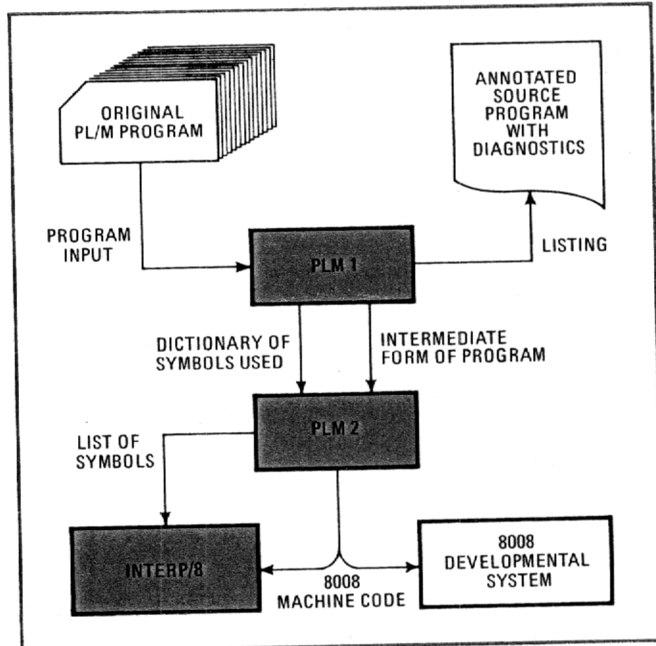
The MCS-8 instruction set includes register-to-register, register-to-memory, and memory-to-register transfers, along with arithmetic, logic, and comparison instructions. Conditional and unconditional transfers and subroutine calls are also provided. Input and output instructions read data from input ports and set data into output-port latches. Each of these instructions is represented in program memory by a sequence of one, two, or three 8-bit words.



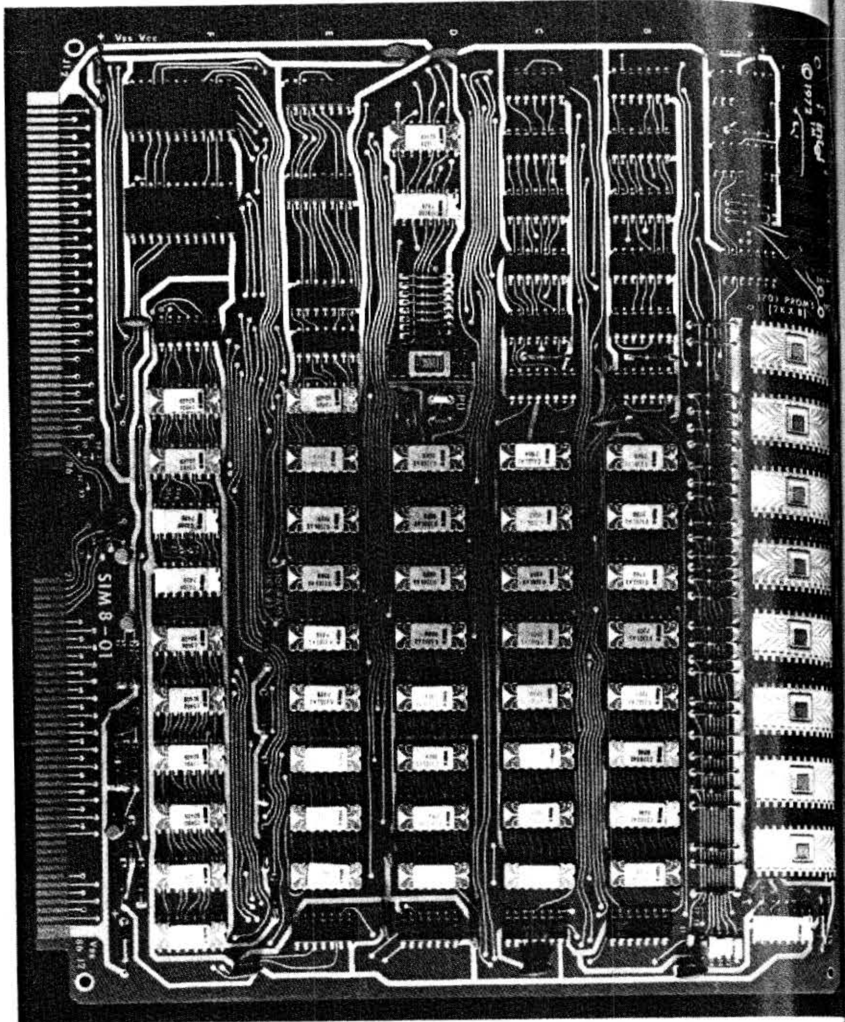
Assembly-language programing, however, is necessarily closely related to the machine architecture because instructions in symbolic code have a one-to-one correspondence with those in machine code. As a result, the programer must spend much more time keeping track of the location of data elements and proper register usage than actually conceptualizing the solution to his problem.

On large-scale computers, high-level languages have been developed to provide important facilities independently of particular machine architectures, while eliminating the trivialities of assembly languages. These facilities include program-control structures, data types, and primitive operations suitable for concise expressions of programs in particular problem environments. For example, a problem environment may be one of numerical computation, in which application-oriented programing languages like Basic and Fortran are appropriate. Or the environment may be the control of a particular class of computer and all its functions, for which system languages, which are necessarily closely related to the machine architecture, are useful.

In a system language, program statements generally correspond directly with machine-level instructions, and conversely, every machine operation is reflected in a high-level language statement. Because of this corre-

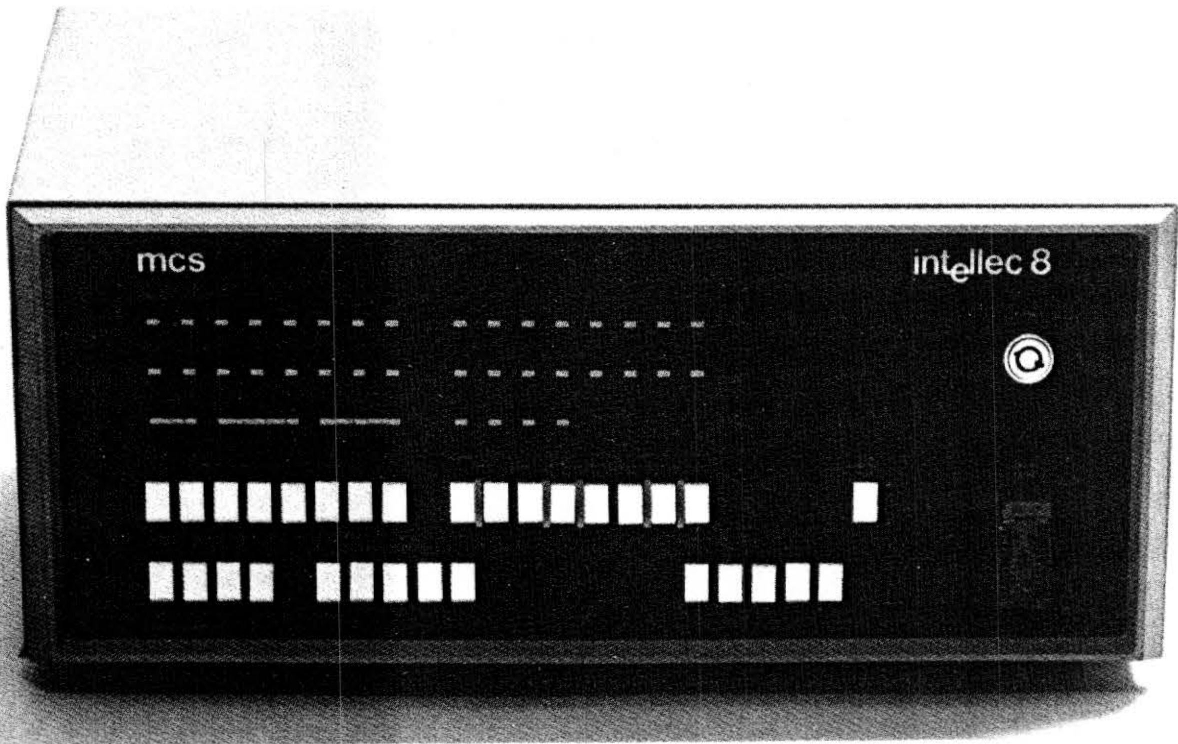


3. Compiler. Translating PL/M programs into machine language takes two passes with programs called PLM1 and PLM2, run by a larger machine. A third pass, with Interp/8, simulates the microprocessor on the big machine to check out the program.



4. SIM8-01. This Intel product checks out a program written in MCS-8 machine code or compiled into machine code from the PL/M language. Erasable ROMs store the program, and a Teletype gives input/output.

5. Intellec 8. This developmental system can check out programs written in PL/M. It also serves as a prototype for production systems based on the MCS-8.



spondence, system-language programs usually translate efficiently to the machine-language level, and the programer finds all the machine's facilities directly available to him. PL/M, an example of such a language, was designed for use with the 8008 microprocessor, and is also usable with Intel's newer 8080 microprocessor [*Electronics*, April 18, p. 95], which has more useful machine-level instructions and a considerably faster instruction cycle than its predecessor.

Nevertheless, some hardware designers, particularly those newly introduced to software systems, may prefer to work at a comfortable level, which may mean programing in absolute machine code initially and then moving to assembly language as more capability is required. Similarly, they can easily make the transition to a high-level language when programing in assembly-language becomes tedious.

In any case, the designer soon becomes familiar with various programing levels. One of these levels can then be intelligently selected as most appropriate for a given application. Each level has its own advantages. For example, a program in PL/M that compiles into about 500 bytes of memory space when using the 8008's instruction set might require perhaps as much as 30% less space if it were coded directly in assembly language. But larger programs running 1,000 bytes or more usually turn out to be more compact when written in PL/M than in assembly language because the compiler can keep track more easily of memory-reference areas, registers, and other resources. The amount of machine code generated in assembly language or PL/M varies, of course, with program complexity and style. Thus, an absolute comparison between the two is not possible.

Simple coding

The PL/M language consists of a number of basic statement types in which complicated arithmetic, logical, and character operations on 8-bit and 16-bit quantities can be expressed in a form resembling usual algebraic notation. Relational tests can be expressed in a natural way to control conditional branching throughout the PL/M program.

For example, to move the larger of two numbers in locations A and B into the location called C, either the PL/M statement,

```
IF A > B, THEN C=A; ELSE C=B
```

or the nine-instruction assembly-language program shown in Fig. 1 can be used. The statement reads, "If the value of A is greater than the value of B, then set C to equal A; otherwise set C to equal B."

Additional language structures provide iteration control to permit program segments to be "looped," or executed repeatedly a prescribed number of times. Subroutine facilities include mechanisms that are useful for modular programing and construction of subroutine libraries.

The over-all structure of the PL/M language is most easily demonstrated by a simple example. Suppose a teleprinter is connected to the least-significant bit of an output port of the Intel 8008. A PL/M program that sends a short message to the teleprinter is shown in Fig. 2; it individually times the transmission of the bits through the output port. This program can be translated into machine code loaded into the memory of the MCS-8, and then it is executed.

The program begins with a data declaration that defines a string of Ascii characters—the words "Walla Walla Wash" as shown in line 1. The 16 individual characters of this string are labeled from 0 to 15 so that they can be addressed by the program (spaces are characters, too). Four variables, or 8-bit memory locations, CHAR, I, J, and SENDBIT, are defined on line 2.

Any names

These designations are wholly arbitrary; the programer may use any names he wants, so long as he defines them before he uses them. CHAR holds each character of the message in succession for transmission, I identifies the position of the character in the message, and J controls the position of the bit in the character. The right-most bit of location SENDBIT is the next bit to be transmitted.

Since the instructions between lines 5 and 17 are executed repetitively, they are collectively called a loop. Before each repetition, the variable I is incremented until its value indicates the position of the last character in MESSAGE—in this case, 15.

First, the value of all bits in SENDBIT is set to 0 on line 7 to send a start pulse as the first bit (line 11). Then the individual bits of the selected character are sent in the inner loop between lines 10 and 16. This loop is executed 11 times, corresponding to the start pulse, 8 data bits, and 2 stop pulses, during each passage through the outer loop, beginning on line 5.

Each successive bit is sent on line 11, followed by a 9.1-millisecond time-out. This time delay is a standard feature in PL/M; the compiler implements it by inserting a wait loop in the program. The wait loop stores an appropriate number in a counter, decrements it once each processor cycle, and allows the program to continue when the counter reaches zero.

On each inner-loop iteration, the right-most bit of CHAR is selected on line 13 by the AND function, and it is stored in SENDBIT. The operation on line 15 places a 1 in the right-most position of CHAR and then rotates the result one step to the right. This step gradually fills CHAR with 1s, working from left to right in each iteration, so that two stop pulses, which are 1s, are sent properly on the 10th and 11th iterations.

The operation of the PL/M compiler and its PLM1 and PLM2 subdivisions is shown graphically in Fig. 3. PLM1 accepts a PL/M source program from a card reader, time-sharing console, or other input device. This first pass produces a listing of the source program, along with any error diagnostics, and analyzes the program structure. An intermediate file that contains a linearized version of the original program is written, and the symbols used in it are listed.

Although the linearized version does not resemble either an assembly language or PL/M, it has been reduced to a highly simplified form of the original program. PLM2 uses this intermediate file as input and generates machine code for the 8008 microcomputer.

A PL/M program can often be checked out by simulating the 8008 microcomputer's actions on a larger ma-

```
STRING COMPARISON PROGRAM
TYPE SOURCE STRING:  A  B  C  D
TYPE TEST STRING:
  A  B  C  D
*   *   *   *
TYPE SOURCE STRING:  666 666 666
TYPE TEST STRING:  6
666 666 666
*** *** ***
TYPE SOURCE STRING:  AAAAAAAABABABA
TYPE TEST STRING: AB
AAAAAAAABABABA
       * * *
TYPE SOURCE STRING:  XXXXXXX$
TYPE TEST STRING:  XXXX
XXXXXXX
****
TYPE SOURCE STRING:  WALLA WALLA WASH
TYPE TEST STRING:  WALLA$
WALLA WALLA WASH
*     *
```

6. Test run. Sample PL/M program produced this printout. Manually entered data is in color, and machine output is in black. Technique is valuable debugging tool.

chine. A third program, called Interp/8, is available for this purpose. The three programs PLM1, PLM2, and Interp/8 are written in ANSI standard Fortran IV, and will run on most larger computer systems.

A new version of the PL/M compiler is available for use with the extended instruction set of the 8080. Consisting of sections PLM81 and PLM82, it is accompanied by a new simulator called Interp/80. New coding is not required for the 8080. Working with old PL/M programs written for the 8008, the compiler can produce binary code requiring 10% to 20% less storage than the 8008 requires, and having the advantages of new interrupt and decimal-arithmetic capabilities.

Experience with PL/M will enable designers of future Intel microprocessors to incorporate new machine-level instructions that will make more efficient use of the PL/M language. Furthermore, if Intel so chooses, it can alter its processor architecture in future designs, as it did between the 8008 and 8080, without affecting the user of PL/M at all, except possibly to improve the performance of this application.

A number of microcomputer manufacturers are considering the use of high-level languages to augment their assembly-language products, although none have been announced yet. Several minicomputer producers, however, offer high-level applications languages, and at least one minicomputer company, Microdata Corp., provides a systems language. In fact, Microdata's MPL language [*Electronics*, Feb. 15, 1973, p. 95] closely resembles PL/M; both of them, in fact, were essentially derived from the same basic system language.

Once the PL/M program is written and checked out, the machine code is punched on paper tape (Fig. 3) and loaded into memory of a microcomputer developmental system. Again, the program is verified, and all real-time and environmental considerations are checked out. Final production systems can then be developed from this prototype. The production system, for example, may use read-only memory for the program when the developmental system's memory is read/write.

How to go on the air

Given a PL/M program and an MCS-8 microcomputer, how does a programer actually go through the compilation and execution process? As mentioned previously, the PL/M compiler is available from several nationwide time sharing services. These are the General Electric, Tymshare, National CSS, Applied Logic Corp., and United Computing Services facilities. Documentation for general programing is available from Intel Corp., and the time-sharing services provide system-dependent operating instructions.

Once the programer has a contract with the commercial service, he is assigned a work area in the host system in which he can store PL/M programs. These programs are created on line by using the time-sharing service's editor, which allows the programer to enter and alter program files. When a particular program is created, it is saved in a permanent file for subsequent compilation.

In the compilation process, PLM1 is executed first, using the saved PL/M program as input. Any diagnostic messages are printed at the time-sharing console. If no program errors are detected during the PLM1 pass, then the programer can call for PLM2. This second pass leaves code in MCS-8 machine language, which corresponds to the original program in the user's work area.

With this code, the programer may execute the Interp/8 program, which reads the machine code and simulates the actions of the MCS-8, as previously discussed. If execution errors appear during simulation, the programer can alter the original PL/M program and repeat the compilation and simulation process. When the programer is convinced the program is correct, he can punch the machine language on paper tape or other medium at his local console.

Programing at home

When a large amount of development work is to be done, the user may find it feasible to purchase the PL/M compiler and CPU simulator directly from Intel and run them on an in-house computer system. The user, at his option, can program either in batch or time-sharing mode.

The machine code produced by the compiler can be executed in several different ways. The easiest method is with a developmental system, such as the Intel SIM8-01 or Intellec 8 (Figs. 4 and 5) or equivalent prototyping hardware. These systems include hardware and software for Teletype, as well as facilities for loading and checking out programs.

The machine code is loaded into the SIM8-01 from the Teletype into erasable read-only memories. These chips are then inserted into sockets on the prototype board, and the program is executed. With the Intellec 8 developmental system, the machine code is entered from the Teletype into read/write memory, where the program can be subsequently executed and tested. Both approaches bypass the simulation stage.

After testing the program on a developmental system, a production model making use of MCS-8 and a mixture of read-only and read/write memory can be tailored closely to the final application. Although the hardware is minimized in the production system to reduce costs, the programs remain the same as in the prototype.

Developing systems

Intel Corp. has completed a number of projects using PL/M, including an assembler that runs on the Intellec 8 developmental system. This assembler's characteristics show the effectiveness of the PL/M approach to system development. For example, it has full macro capabilities, which means that a programer can define special pseudo-instructions that cause the assembler to insert sequences of instructions in the main program during the assembly process. Macros are like subroutines, except that the main program executes them as it comes to them, instead of branching out of the main stream and then returning, as it does with subroutines.

The assembler is also capable of conditional assembly, which means that it can react to such external signals as the positions of console switches at the time of assembly. Such signals indicate conditions that are not necessarily known to the programer at the time he writes the code—such as the availability of particular output equipment to which the assembler's results are to be sent.

Another useful characteristic of the assembler is evaluation of expressions at assembly time, which permits the programer to specify certain parameters algebraically instead of numerically or symbolically. Then when a program is assembled, the assembler evaluates the algebraic expressions and inserts the correct values in the machine-language program. The process requires the variables to be specified ahead of time, but it permits the programer to alter these variables by changing their specification only once, rather than every time they are used in the program. It's a great time-saver and bug-killer.

While these characteristics are not uncommon in advanced assembly languages, high-level languages that can handle them are quite rare. Yet by using PL/M, the assembler was coded in approximately 100 man-hours, and it requires 6,000 bytes of program storage—equivalent to 3,000 words on a minicomputer with a 16-bit word size. Intel estimates that the project would have taken five times as long to code and debug directly in assembly language, with little or no reduction in program-memory space. The resulting assembler is easy to maintain and alter, and, equally important, it can be recompiled for Intel's new 8080 microprocessor without alteration.

A practical example

PL/M permits many programing shortcuts, such as dividing a complex task into individual subtasks, or procedures, that are called upon when needed to simplify the job of writing the program itself. These procedures are conceptually simple and therefore easy to formulate and express in PL/M, as well as easy to check out before being incorporated in a larger program.

For example, consider a simple program for character manipulation—one that might be part of the work of a more comprehensive word-processing system. The function is relatively simple: the program asks the keyboard for two input-character strings, scans the first string for all occurrences of the second, echoes the first string, and types an asterisk under the starting position in the first string of each occurrence of the second string. A sample interaction with this program is shown in Fig. 6; all lines typed by the operator are in color.

Stated in this way, this example may seem to have little or no practical value. But it is almost identical to a program needed to fetch the strings from two different data-entry devices and do something more sophisticated than printing an asterisk when it finds a match.

This suggests a practical application—a teleprinter to check out a routine before it is embedded in a larger program. When all the bugs are out of the routine, the procedures that transfer data to and from the teleprinter can be replaced with other procedures that, for example, check sensors and turn indicators on and off. The new procedures, of course, have to be checked out in a real environment, but that's much easier when the main routine is known to be bug-free. □