FORGET MNEMONICS?

Editor:
The special feature by Wayne P. Fischer, "Microprocessor Assembly Language Draft Standard," in Computer December 1979, has just been brought to my attention. I am horrified. How can the answer to too many mnemonics be more mnemonics? I regret that my views on microprocessor assembly languages have taken so long to reach your side of the Atlantic; my paper, "Level-Independent Notation for Microcomputer Programs" in the IUCC Bulletin, Vol. 1, No. 1, Spring 1979, has probably not circulated outside the United Kingdom, and my book Microprocessor Programming and Software Development, although published by Prentice-Hall International in September 1979, appeared too late to be considered by the authors of the report. In both of these publications, there is an account of another approach to the problem of expressing programs for microprocessors at machine-code level, rejecting the conventional use of literal mnemonics.

Although it is probably true that "assembly languages" based on literal mnemonics were among the first languages used in programming computers, it is no less true that many computers never had languages of this type. Indeed, although I was aware of such languages, I had no need to be involved with one at all during the first 20 years of my programming experience. But then, during 1975-6, when I first had to come to terms with a number of different microprocessors, I was faced with four or five all at once. In order to save time, money, and mental distress, I wrote my first microprocessor assemblers, disassemblers, monitors, and arithmetic packages (about 12K-14K bytes) in absolute hexadecimal; my working notes throughout this stage gradually, but quite quickly, turned into programs in the developing notation.

The 6800 and 8080 have, I think, 37 instructions in common, but their official assembly languages share only three mnemonics. For example, both processors have eight-bit registers called A and B. Each has an instruction for replacing the contents of A by a copy of the contents of B, leaving B undisturbed. For the 8080, we are supposed to write MOV A,B ("move B to A"), and for the 6800, B,A ("transfer B to A"); and now your contributors propose MOV B,A ("move B to A"). The first point I have to make is that "move" and "transfer" both give very misleading impressions as to the nature and utility of the operation. Nothing is moved or transferred. The contents of B remain. What happens is that a copy of what is, and remains, in B takes the place of what was in A. It is a copying operation. The second point is that a clear and unambiguous notation for this has been in widespread use for the last 20 years. It is A: = B, usually read as "A becomes B." In none of the mnemonic forms is the direction of the operation clear, but there is no possible doubt as to the meaning of A: = B. Then there is addition. We have ADD, DAD, ABA, and now ADD. The last is plain enough, but what is wrong with " + "?

These ideas are introduced and developed in Chapter 2 of my book, and in Chapter 3 the complete instruction sets of the 6800 and 8080/8085/Z80 are expressed in the resultant notation. Chapter 8 contains the complete text, in this notation, of the first assembler and disassembler for it on the 8080. (The current version of the assembler has better facilities and is more compactly written.)

Since early 1977, this notation has been used in many undergraduate and further education courses attended by around 500 students. At Bristol it is the basis of microprocessor work in the computer science syllabus; this year it is being used by about 100 undergraduates. Many of them show their preference for my notation rather than mnemonics, and regret that it is not more widely used. Some evening students have introduced it into the systems operated by their companies. It has been wholeheartedly adopted by my colleague Professor J. R. Mühlbacher at the University of Linz, in Austria.

I would be the last to suggest that any work of mine could be adopted as a standard without very critical revision, and I am not at all sure that standardization is a proper objective in the current situation. We do not know what other new microprocessors are coming along, and none of us has the experience and wisdom to provide ideal solutions for all possible developments. A far more realistic objective would be to aim for a notational system which is amenable to improvement and development and is applicable across a significant part of the range of available and anticipated machines. . . .

...there are, and always will be, programs, or passages in programs, in which operations at the lowest level have to be expressed, whatever the height to which the programming language may aspire. The mnemonic-based assembly languages, including that proposed, are quite unlike and unrelated to the higher-level languages. Basic with PEEK and POKE is at best an unsatisfactory, makeshift union. The approach I have taken, as suggested in my I.U.C.C. paper and in Chapter 7 of my book, opens up at least the possibility of a unified language able to comprehend both high-level expression and low-level details such as the recognition and generation of signal levels and pulses.

In my view the IEEE would be most wise to proceed with the proposed standard. Were it to do so, and produce a standard, that standard would be necessarily ill-defined and restrictive. Many of us would ignore it, and those who observed it would find their programming stultified in a style which is already archaic and which precludes any development in the direction of good practice at higher levels. To take one specific point, there is no accommodation for structural elements. Structure is at least as important in low-level programs as in high, and never more so than in programs which interact with an external physical environment.

There is, however, an aspect of standardization which the IEEE would do well to pursue, for reasons which are evident in the pages of Mr. Fischer's report. Standard definitions of arithmetic and logical operations, and specifically of conditions, condition flags, and of the circumstances in which such flags ought and ought not to be set, are needed, and designers need to observe such definitions, or say how they have improved on them.

F. G. Duncan University of Bristol

Editor:
The attempt by Wayne Fischer ["Microprocessor Assembly Language Draft Standard," Computer, Dec. 1979] to define a standard for assembly language conventions for present and future microprocessors, although valiant, is in our view misguided. . . . We completely agree with Mr. Fischer about the existing mishmash of assembly languages for microprocessors currently in existence, but suggest that before
attempting to define standards it is well worth considering why we need assembly languages.

Assembly languages, in which each instruction is translated directly into a machine code instruction, are used in cases where high-level languages are incapable of expressing the required machine operations. Such cases arise almost without exception when a particular aspect of the machine architecture needs to be exploited for a particular task. It follows that the assembly language programmer must have a close knowledge of the hardware when designing and coding such algorithms. . . .

The proposed standard defines a very large set of mnemonics, which almost without exception have the effect of disguising the basic architectural features of the machine from the user. Furthermore, the technique of defining a standard by enumeration immediately brings problems when new features are introduced in future machines. (What will happen, for example, if 32-bit micros include a set of increment/decrement instructions and not just the single count used at present? Or, how would this set cope with a search list instruction which already exists on some mainframe machines?)

We suggest that a better way to approach this problem is the notation due to F.G. Duncan [Microprocessor Programming and Software Development, Prentice-Hall, 1979] which provides the equivalent of a syntactic framework which can easily be extended to describe a range of machines. Furthermore, the language convention used closely resembles those of high-level languages, and this greatly adds to the user convenience. . . . We have both had experience teaching this to classes of undergraduates at the Universities of Linz, Austria, and at Bristol, and the advantages of this notation (which we should emphasize is not a standard, but the framework for describing a range of machines) has been demonstrated, especially when larger and more highly structured programs are implemented. There is one more advantage to this approach: the notation has not been copyrighted by any microprocessor manufacturer in order to preserve proprietary interests. A complete listing of the assembler is freely available for implementation on any system. We hope that this type of approach will be carefully considered before a stampede into standards occurs.

Chairman’s reply:

In reference to Mr. Duncan’s and Mr. Rogers’ letters, I basically concur that Mr. Duncan’s approach to machine language programming may be a better way. Although I have not read his book, or programs written in his proposed language, it seems to me that his approach is a readaptation of Pascal on the machine level. Using a subset of Pascal oriented directly to the machine level is surely a worthwhile approach, and is another way of moving the programmer as far away as possible from the machine—all he needs to know is the language’s grammar. In the future, there may very well be a macro-type Pascal assembly language that will become a practical tool for programming large programs that still need to be tightly coupled to the machine.

I nevertheless maintain that for current processor architectures assembly language programs will always be with us, for the following reasons:

- Manuals written for existing processors describe instructions in title, mnemonic, and operand formats. This is still the best way to describe instructions, since each must be described in a precise manner.
- Engineers and programmers writing tightly coupled programs for particular processors will prefer to stay with the conventional assembly languages, since they describe exactly what is happening to the processors internally.
- Because a tremendous amount of code has been written in assembly language, assemblers will be needed for many more years just to support that code.

Even though Pascal is becoming one of the high-level languages, Basic (or a similar derivative) will always remain a low-level language used by programmer novices and people who have a basic (no pun intended) understanding of computers. Pascal is a highly structured and complex language and, once learned, is powerful, versatile, and self-error-checking. Only the person highly skilled in mathematics and programming can effectively use the full power of Pascal. On the other hand, a person not familiar with computers and having a limited math background can quickly learn and use a Basic-type language.

The same philosophy applies to assembly language. Hardware and systems designers who know very little about programming and yet are applying microprocessors to their designs can quickly write small programs for their applications. Moreover, designers can implement assembly-language-type programs in microprocessor-based systems that have no, or very small, operating systems; with higher-level-language programs, no such implementations are possible.

I wholeheartedly support a low-level, Pascal-type, structured, macroassembler language which allows the programmer to write more instructions with fewer errors. In the meantime, however, let’s finish a task that still needs to be finished.

Wayne Fischer
Chairman, IEEE Task P694
(Microprocessor Assembly Language)

796 Bus Standard to allow optional connector numbering

Editor:

One major comment about the edge connectors J1, J2, and J3 in the IEEE 796 Bus Standard—they’re backwards. [See “Proposed Microcomputer System 796 Bus Standard,” Computer, Oct. 1980.] Every major flat cable connector manufacturer puts wire #1 on the multibus’ pin #2. I realize this preserves the “handedness” symmetry for PCB designers, but many more users have to struggle with “is pin 34 wire 35 or 36 . . .” How about having pin 1 = wire 1 = RS232 pin 17? It may be late to change, but that’s the purpose of a standard—to take the bugs out of proprietary designs.

Bruce Greif
Sundsvall, Sweden

Chairman’s reply:

When this same comment was received by the P796 Working Group on October 9, 1980, the group passed a resolution: “Allow optional connector numbering, compatible with the industry standard. Section 4.2.1 changed.” Section 4.2.1 now refers to only main connectors P1 and P2, not to J1, J2, J3, etc., at the top edge. Additionally Figure 32 was changed to remove all pin-numbering references for the example J1, J2, J3 connector configuration shown. Also note that J1, J2, J3 in this figure are labeled “a possible connector configuration” with no limitation as to actual configuration or numbering scheme.

Richard W. Boberg
Palo Alto, California

*To be considered for publication, a letter to the editor must be accompanied by a statement giving Computer permission to publish that letter.