The near-optimal instruction set

Editor:

Dennis Fairclough's article on instruction set design ("A Unique Microprocessor Instruction Set," May 1982) addresses an important issue during the current explosion of microprocessor usage. I find several problems, however, with his recommendations.

The article does not adequately address the issue of discarding instructions with low static frequency. Such instructions are essential to the processor. I suspect that the MC6800 RTI (return from interrupt) falls in the author's low static usage set, yet it is essential for interrupts. When deciding whether a particular instruction should be implemented or not, the designer considers not only the potential static and dynamic usage of the instruction, but also the cost of implementing it vs. the cost of not implementing it. Some instructions, though not frequently used, would be expensive to simulate in macrocode—the DAA (decimal adjust), for example.

More critical to users are the useful instructions that are omitted, not because of lack of opcode space, but because of oversight or misjudged impact. PUSH X is a good example—it was omitted on the MC6800, but added to the MC6801 and MC6809.

Although several minicomputers and some microprocessors initially had ad hoc instruction set designs, the latest offerings show careful consideration of software impact. Instruction sets are becoming more orthogonal and more supportive of high-level languages. The Intel IAPX 432 and Motorola 68000 are examples.

Multitasking, synchronization primitives, and high-level subroutine linkages which handle parameter passage and automatic local storage allocation are examples of operations which can and should be supported by hardware. No mention of such support was made in the article.

Fairclough states, "The fewest number of bits should be used to huffman-encode every field in the instruction format. . . ." I was disappointed that he did not give a bit layout for his design on a particular architecture. I wanted to see how he used a variable-length encoding of opcodes (a single bit for MOVE!) while keeping the opcode, address mode, and address modifier fields separate. Would the program counter address to the bit level?

Commercial processors do use variable-length opcodes to obtain memory efficiency, but the length of the opcode is determined by the number of modifiers required and, in some cases, by the frequency of usage (multibyte rather than single-
Thad Smith III was very observant in his reading of my article—he makes some valid points and some not so valid points.

First let me point out that my article is planned to be one of a series of four and does not explain all of the details of the complete near-optimal instruction set design. This was not explicitly stated in the article.

Smith states, “The article does not adequately address the issue of discarding instructions with low static frequency. Such instructions are essential to the processor. I suspect that the MC6800 RTI instruction (return from interrupt) falls into the author’s low static usage set ...” The purpose of the static instruction frequency usage is to guide us in the design of an instruction set. The RTI instruction does not fall into the low static frequency group; since it modifies the program counter it is in Group II (program modification instruction group, with a 25 percent instruction usage). The RTI instruction is provided by the MOVE source 1, destination 1, source 2, destination 2, CC instruction, in which the instruction takes the form of MOVE top of stack to processor status register, and next top of stack to program counter, on condition.

Smith also makes the point, and rightly so, that “When deciding whether a particular instruction should be implemented or not, the designer considers not only the potential static and dynamic usage of the instruction, but also the cost of implementing it vs. the cost of not implementing it.” This is a very valid point, but I am not sure that most computer engineers apply this criterion. It is hard to believe that much is known about how microprocessors are used (or computers in general, for that matter), when 8.7 percent to 30.3 percent of the instructions provided are never used. This was pointed out in my article in Table 1. I believe the methods mentioned in my article, when used to include or exclude instructions, are valid and in time will prove to be a worthwhile design methodology.

Smith also points out that the DAA would be expensive to implement in macrocode. There is no evidence to support the requirement that the DAA should be implemented at all. Shustek (Reference 17 in my article) states that “it would probably be better to eliminate directly computing decimal arithmetic and do the conversions when necessary.” The research for my article shows that DAA instructions or their equivalent were used 1.82 percent of the time in the MOS6502, that BCD instructions were used less than 0.1 percent of the time in the MC68000, and that the TMS9900 doesn’t even have DAA or BCD arithmetic instructions.

“More critical to users are the useful instructions that are omitted, not because of lack of opcode space, but because of oversight or misjudged impact,” Smith notes. I think this point is exactly the problem that my article tried to address. We cannot solve a problem if we do not know what the problem is! My article, I hope, suggested ways to ensure that such “oversights” and “misjudged impacts” will not become part of the design. My design did not omit the PUSH X instruction, as was the case in the MC6800. The PUSH X instruction may be implemented with a MOVE top of stack instruction.

Quoting Smith again, “Although several minicomputers and some microprocessors initially had ad hoc instruction set designs, the latest offerings show careful consideration of

Z-80® and 8086 FORTH

FORTH Application Development Systems including interpreter-compiler with virtual memory management, assembler, full screen editor, line editor, decomplier, demonstration programs, and utilities. Standard random access disk files used for screen storage. Extensions provided for access to all operating system functions.

120 page manual.

Z-80 FORTH for CP/M 2.2 or MP/M ........ $ 50.00
8086 FORTH for CP/M-86 .......... $100.00
PC/FORTH for IBM® Personal Computer .......... $100.00

Floating point extensions for above systems. Specify software floating point, AMD 9511, AMD 9512, or Intel 8087 support ................ additional $100.00

Nautilus Cross Compiler systems allow you to modify the FORTH nucleus, recompile for a different target computer, generate headerless and/or ROMable code. Supports forward referencing. Produces load map, list of unresolved symbols, and executable image in RAM or disk file. 100 page manual. (Prerequisite: one of the application development packages above for your host system)

Z-80 host: 8080 or Z-80 target .......... $200.00
8086 host: 8080, Z-80 or 8086 target .......... $300.00

PC/FORTH distributed on 5¼ inch soft sectored double density diskettes. All other software distributed on 8 inch soft sectored single density diskettes. Prices include shipping by UPS or first class mail within USA and Canada. California residents add sales tax. Z-80 is a trademark of Zilog, Inc. IBM is a trademark of International Business Machines Corp. CP/M is a trademark of Digital Research, Inc.

Laboratory Microsystems
4147 Beethoven Street
Los Angeles, CA 90068
(213) 306-7412

Reader Service Number 3

IEEE MICRO
software impact. Instruction sets are becoming more orthogonal and supportive of high-level languages. The Intel iAPX 432 and Motorola 68000 are examples. It is not apparent to me that the newer minicomputers or microprocessors show careful consideration of software impacts. I think what has happened is that we have packed more active devices on a chip and then covered our tracks by spraying the terrain with more instructions. If these were not so, the MC68000 would not have 55.3 percent of its instruction set used less than 0.1 percent of the time. The MC68000 instructions are far from being orthogonal—just try to use the address registers and data registers with the same instructions.

Smith wants the instruction set to be able to handle high-level-language requirements. This is a reasonable requirement and I think my design does just that! Without specific arguments to the contrary, it is not possible for me to address these areas.

The bit layout of the instruction format were not included in the article, as Smith noted, but will be included in one of my follow-up articles.

Smith makes the point that with my design some type of modifier field is required for the different types of addressing modes. He is right. The modifiers are not instruction modifiers; however, they are addressing modifiers and will be discussed in a follow-up article on addressing modes and modifiers.

He also points out that my instruction set is not orthogonal for the MOVEM instruction, since it does not have a condition-code field. I’m afraid he has exposed a weakness in the design here. I could not see any possible use for a move multiple instruction on condition. There may exist such an animal but I could not justify creating it.

My instruction set will handle all of the fixed-point arithmetic that a user may want to implement (and it will handle it very efficiently). It does not provide floating-point instructions, as these would be best provided by a floating-point or arithmetic-processing chip rather than by specialized instructions. The latter would be an unnecessary burden on the instruction set—an external chip is a better solution to the problem.

“The area to which I take the strongest objection is Fairclough’s ‘scientific’ approach to designing an instruction set . . . .” Smith states. He goes on to say that my approach **ignores application requirements and language considerations.** I strongly contest this criticism, for I used a large number of high-level-language programs, including applications programs, in generating the instruction groups presented in the article. I did not ignore these areas at all! I feel that my instruction set adequately addresses the applications and high-level-language programs.

Smith concludes by recommending that we should “first design machines to solve problems, then optimize.” I agree with this statement for the most part, but have one slight criticism of it. It has been my experience that we never get around to optimizing. I feel that if we have the data that tells us where and what to optimize (and we do), we should optimize first. My article attempted to provide data showing where to optimize an original design. There is a saying many readers may be familiar with—“We don’t have time to do right, just time to do it over again.” In the case of computer architecture, we don’t even have the time to do it over again. As a result, it is programmers who end up compensating for our design misjudgments—over and over again.

Dennis A. Fairclough

August 1982