1982-05-01

A Unique Microprocessor Instruction Set

D. A. Fairclough
fairclde@uvsc.edu

Original Publication Citation

Fairclough, D. A. "A Unique Microprcessor Instruction Set." Micro, IEEE 2.2 (1982): 8-18

New instruction sets have been based on tradit

Here, the latter has yielded a simple, optima

A Unique Micropr

Dennis A. Fairclough, Brigham Young University

The fundamental building block of any computer is its instruction set. In the architecture of a new computer, the selection of the instruction set is the most critical decision. The decision is even more critical in microprocessor architecture, due to the restrictions on power dissipation, number of input/output pins, die size, speed, and chip complexity.

The instruction set format is composed of the instruction, the word size to be accessed, the address mode, the address modifiers, and the condition codes, in a binary format. The *instruction set* is the group of instructions that the system may execute, while the *instruction set format* is the arrangement of the instruction field, address field, and other fields in memory.

Instruction sets appear to vary widely from computer to computer, but a closer analysis shows that they have many similarities. The instruction set of one microprocessor can be remarkably similar to that of a different microprocessor. Different microprocessors can even share identical instructions which are supersets of some earlier microprocessor.

Computer instruction sets are evolutionary rather than revolutionary in design; most are simply extensions of earlier sets. One reason for this is the need to be program-compatible with older computers. Another is the complexity of the design task itself. This complexity also forces a heuristic approach—designers must make heuristic decisions because they lack data on how instructions will be used.

An example of how heuristic architectural decisions are made is detailed in a paper by Foster.[1] Foster relates,

0272-1732/82/0500-0008$00.75 © 1982 IEEE

ther than on the application of scientific method.

n-member instruction set.

essor Instruction Set

"Someone once asked why Stretch was designed with eight index registers. The answer that is reported to have been given was 'clearly four are not enough and 16 would be too expensive.' In the same spirit we choose 32 bits as the word length of our microcomputer." Design decisions made in this manner are typical of those made in the design of computer instruction sets.

A very good programmer once told me, "the instruction set on the Nova is the best that will ever be." Heaven help us if this is true of any instruction set. Instruction sets are like high-level languages—no one is the best; there are only languages that are less restrictive than others.

In providing for program portability, programmers have created their own software instruction sets. The p-code used in the UCSD Pascal system[2] is a good example of a software metainstruction set. By forcing a standard software interface, software portability is more easily achieved. This is not all bad, as evidenced by the large number of microprocessors using UCSD Pascal. The UCSD approach forces transportability at the expense of efficiency in the execution of the target-machine program.

The above are just a few examples of architectural complexity and how this complexity forces heuristic design decisions. The complexity that exists in computer architecture in general, and instruction set design in particular, encourage heuristic designs and evolutionary microprocessor architectures. Because computer architects generally do not know how their instructions will be used, they find it safer to follow an existing design than to create a new set from scratch.

The first goal of this article is to present data on how instruction sets have been used. The second is to define a scientific approach to instruction set design and then to use that approach to construct a new microprocessor instruction set. If we do not understand the past, we are condemned to relive it.

Previous work

A study of the Maniac computer by Herbst, Metropolis, and Wells[3] analyzed by Foster[4] showed that in the Maniac 16 of the 36 possible instructions accounted for 90 percent of all instructions written, and 24 instructions accounted for 99 percent of all instructions written. Foster[4] observed that the CDC-3600 instruction set could be reduced to one-half or one-quarter the size of the present instructions without loss of flexibility. He demonstrated that if the CDC-3600 instruction set was reduced from 142 instructions to 64 instructions, only two percent of the instructions executed would not be in the smaller instruction set. Alexander and Wortman[5] reported similar results in a study of the IBM System/360.

In research by the author, studies were made of programs used on four microprocessors: the Texas Instruments TMS9900, the MOS Technology MOS6502, the Motorola MC6800, and the Motorola MC68000. A significant number and variety of programs were analyzed.

Table 1.
The use of microprocessor instructions.

INSTRUCTIONS EXECUTED	TMS9900		MOS6502		MC6800		MC68000	
	INSTR.	CUM.	INSTR.	CUM.	INSTR.	CUM.	INSTR.	CUM.
≤ 0%	8.7%	8.7%	7.1%	7.1%	27.1%	27.1%	30.3%	30.3%
≤0.1	14.5	23.2	12.5	19.6	27.1	54.2	3.9	34.2
≤0.5	29.0	52.2	10.7	30.3	27.1	81.3	21.1	55.3
≤1.0	20.3	72.5	14.3	44.6	6.5	87.8	11.8	67.1
≤2.0	7.2	79.7	26.8	71.4	0.9	88.7	10.5	77.6
≤3.0	5.8	85.5	16.1	87.5	1.9	90.6	13.2	90.8
≤4.0	7.2	92.7	3.6	91.1	4.6	95.2	0.0	90.8
≤5.0	2.9	95.6	3.6	94.7	0.9	96.1	6.6	97.4
>5.0	4.3	99.9	5.4	100.1	3.7	99.8	2.6	100.0
TOTAL	99.9%		100.1%		99.8%		100.0%	

A summary of the data is given in Table 1. The table shows that 8.7 percent to 30.3 percent of all the microprocessor instructions were never used; also 44.6 percent to 87.8 percent of the instructions were used 1 percent or less.

The data could be misleading if you simply look at the percentage of use. Ideally the frequency of use would have a uniform distribution. For the microprocessors analyzed, the uniform instruction-usage distribution should be:

- TMS9900 1.45 percent
- MOS6502 1.79 percent
- MC6800 0.93 percent
- MC68000 1.32 percent

These percentages place a figure-of-merit value on the instruction usage. If the instruction usage U is

$$U/10 \leq UD \qquad \text{(Equation 1.)}$$

Where U = instruction usage and UD = uniform instruction distribution usage, then the inclusion of the instruction in the instruction set should be eliminated. Using this criteria, we find that for the

- TMS9900, 18 of 69 total instructions (26%),
- MOS6502, 13 of 56 total instructions (23%),
- MC6800, 29 of 107 total instructions (27%),
- MC68000, 31 of 76 total instructions (41%),

satisfy equation 1 and should be eliminated from the instruction set.

The instruction set usage presented above prompts one to ask, Is this a science or an art? Based on the data presented, we must answer that instruction set architecture is an art masquerading as a science. Knuth,[6] in an article entitled "Computer Programming as an Art," said, ". . . a transition of programming from an art to a disciplined science must be effected. . . . we have actually succeeded in making our discipline a science, and in a remarkably simple way: merely by deciding to call it 'computer science.' " Computer architecture and instruction set design are a combination of parody, art, and science.

The studies by Herbst, Metropolis, Wells, Alexander, Wortman, and Foster, and the research reported here, clearly indicate that a scientific approach to the design of instruction sets is required.

The procedure is first to determine how existing instruction sets are used and what particular groups of instructions predominate (and are significant). The methodology is based on instruction group usage. Only when the findings are in hand can a new instruction set be designed.

Before we begin this procedure, however, we must first investigate the history of instruction mixes, instruction frequencies, and instruction groups.

Table 2.
The Gibson Mix.

(1)	LOADS AND STORES	31.2%
(2)	FIXED-POINT ADD AND SUBTRACT	6.1
(3)	COMPARES	3.8
(4)	BRANCHES	16.6
(5)	FLOATING-POINT ADD AND SUBTRACT	6.9
(6)	FLOATING MULTIPLY	3.8
(7)	FLOATING DIVIDE	1.5
(8)	FIXED-POINT MULTIPLY	0.6
(9)	FIXED-POINT DIVIDE	0.2
(10)	SHIFTING	4.4
(11)	LOGICAL AND, OR, ETC.	1.6
(12)	INSTRUCTIONS NOT USING REGISTERS	5.3
(13)	INDEXING	18.0
	TOTAL	100.0%

Instruction mixes. The most widely quoted study on the usage of instructions is that of Gibson.[7] Gibson attempted to quantify a tool—the "Gibson Mix"—in order to ". . . plan and design new computers, to estimate the worth of a computer to a user, and to plan data processing systems." Unfortunately, the Gibson Mix combined both instruction usage, instructions not using registers, and indexing. The Gibson Mix also combined instructions and addressing modes, two entirely different processes; mixing them only obscured the relationships between instructions and addressing modes.

The Gibson Mix grouped instructions by common characteristics, according to the function they performed. The Gibson Mix is shown in Table 2.

We must make a point in defense of Gibson's approach—most computer architects mix instructions and addressing modes. A good example of this mixing is the LDA (LOAD ACCUMULATOR IMMEDIATE) instruction format. The LDA instruction LOADs the IMMEDIATE data (the source address) into the A register (the implied destination address).

The mixing of address modes with instructions is a common occurrence. In early computers, instructions and address fields were mixed due to the small memory space available. Memory space is not a major problem today and there is no reason to mix address modes, instructions, and other information; in fact, there are many reasons not to mix them. One might argue that the mixing of instructions and address types saves bits in the instruction format and thus reduces the fetch time of the address. The discussion on address types will demonstrate that this is not the case.

After Gibson, and following his lead, other researchers developed other mixes—Arbuckle[8] one for the IBM 650, Agarwal[9] and Lunde[10] one for the PDP-10, Schreiber[11] one for the IBM 360, Foster[12] one for the CDC-3600, and others[13-15] ones for other machines. Such instruction mixes provided the basis on which the author generated instruction groups.

Instruction frequencies. Instruction frequencies have been investigated by Alexander[16] for the IBM 360, Shustek[17] for the Amdahl 470 and Intel 8080, Shima[18] for the Z8000, and Stritter[19] for the MC68000. Instruction frequencies record the frequency of use for individual instructions. They suffer from many problems—one is that in most computers instructions and addressing modes are mixed; another is that individual instructions do not always appear in differing computer architectures. The lack of uniformity from one computer to another makes individual instruction frequencies measurements of limited usefulness.

Instruction groups. Instruction groups were originally created by the author in May, 1976, but have not been published until now. The groups are variations on the Gibson Mix but with one very important difference: instructions in these groups are not mixed with address modes. All instructions are pure instructions.

Instructions are divided into eight categories:

- *Data movement instruction group.* Instructions in this group are LOAD, STORE, and MOVE.
- *Program modification instruction group.* Instructions in this group are BRANCH, JUMP, CALL (subroutine), and RETURN (subroutine).
- *Arithmetic instruction group.* Instructions in this group include ADD, SUBTRACT, MULTIPLY, and DIVIDE.
- *Compare instruction group.* This group includes both arithmetic and logical compare instructions.
- *Logical instruction group.* Instructions such as AND, OR, XOR, and NOT are included in this group.
- *Shift instruction group.* Instructions in this group are SHIFT and ROTATE.
- *Bit instruction group.* Bit set, clear, and test are in this group.
- *Input/output and miscellaneous instruction group.* In this group are input and output instructions and those instructions that do not logically fit into any of the other seven groups.

The collecting of instructions into groups masks out the idiosyncrasies of individual instruction sets. Using these groups allows the instruction sets of many different computers to be easily and accurately analyzed.

Definitions

To provide an unambiguous working vocabulary, the following definitions are provided:

- *Instruction set.* A computer's instruction set is the set of all instructions that can be executed.
- *Instruction format* (*instruction set format*). The instruction format is the binary configuration that the instruction-bit field, the address-mode-bit field, the address-modifier-bit field, and all other instruction-bit fields are formatted into. The instruction format is in a form that the control unit may easily decode for instruction execution and address resolution.
- *Address field.* The address field is that portion of the instruction format that defines the unique bit configurations for the specific address types.
- *Instruction field.* The instruction field is that portion of the instruction format that defines the unique bit configurations for the instruction (operation code).
- *Instruction* (*operation code*). The instruction (op-code) is an object that instructs the control unit which operation to perform.
- *Address type.* The address type refers to a general addressing method used to obtain the final effective address for a memory unit. The address type is made up of the address mode and the address modifier.
- *Address mode.* The address mode is the algorithm by which the address of a memory unit is calculated. These addressed units include flip-flops (bits), registers, and main memory.
- *Address modifier.* The address modifier provides the modification to a memory address (effective address) just prior to actually addressing memory. Indexing is an example of an address modifier.
- *Source address.* The source address is the modified effective address from which address data is read.
- *Destination address.* The destination address is the modified effective address to which data is written.
- *Pure instruction.* A pure instruction is an object (field) that contains only operation-code information.
- *Pure address mode and modifier.* A pure address mode is an object (field) that contains only addressing information.

Instruction groups

The instruction group method allows the classification of instructions by function. It provides a convenient way to compare, on a common basis, the instruction sets of many different computers. However, it can only be used to compare instruction sets on machines having similar architectures. This work concentrated on von Neumann-style register-to-register and memory-to-memory architectures.

The data on instruction groups were obtained by analyzing programs written for the TMS9900, the MOS6502, the MC6800, and the MC68000. Programs were randomly selected and were of many differing types. The programs analyzed included applications programs, assemblers, interpreters, compilers, monitors, kernels, op-

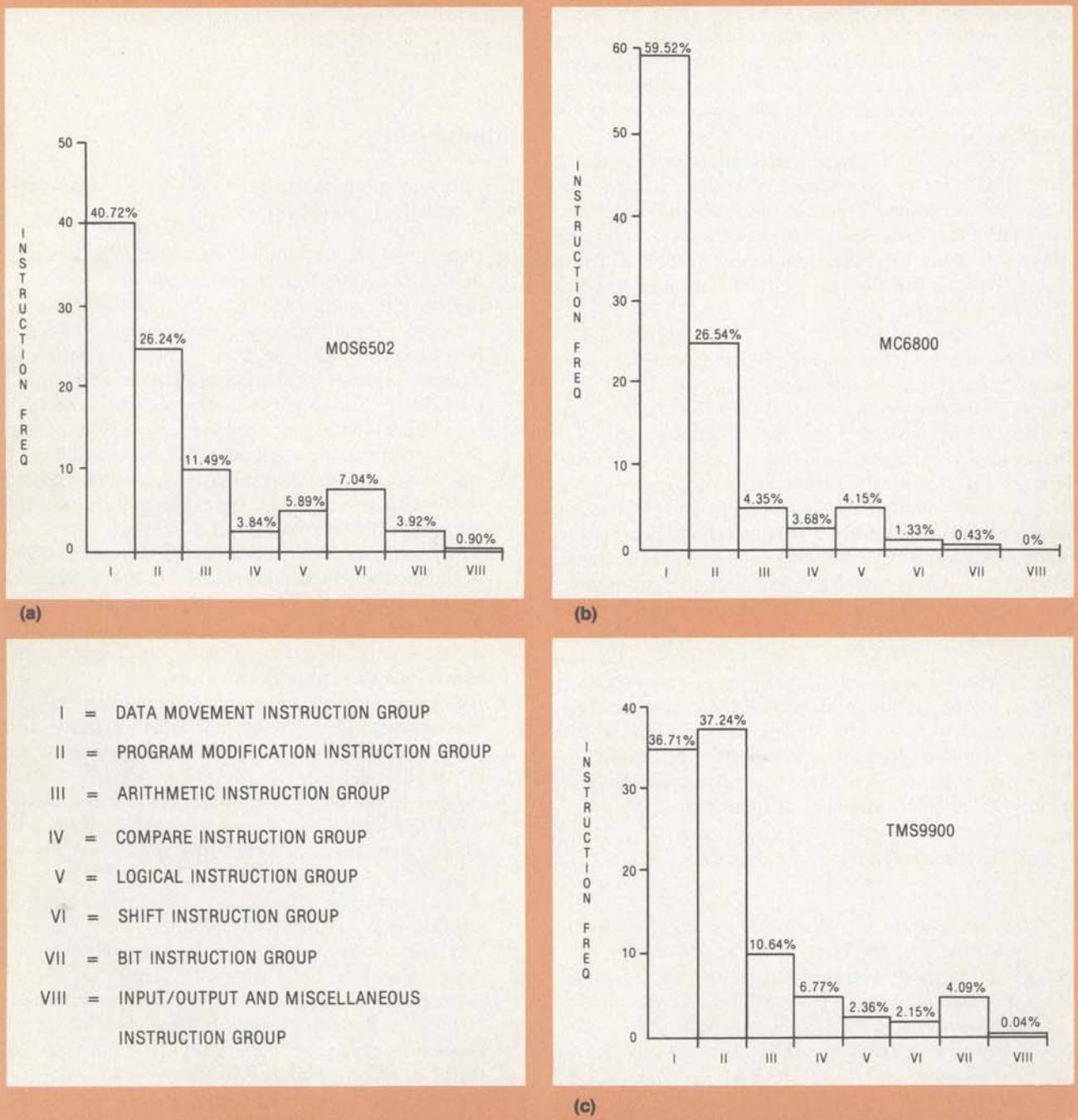
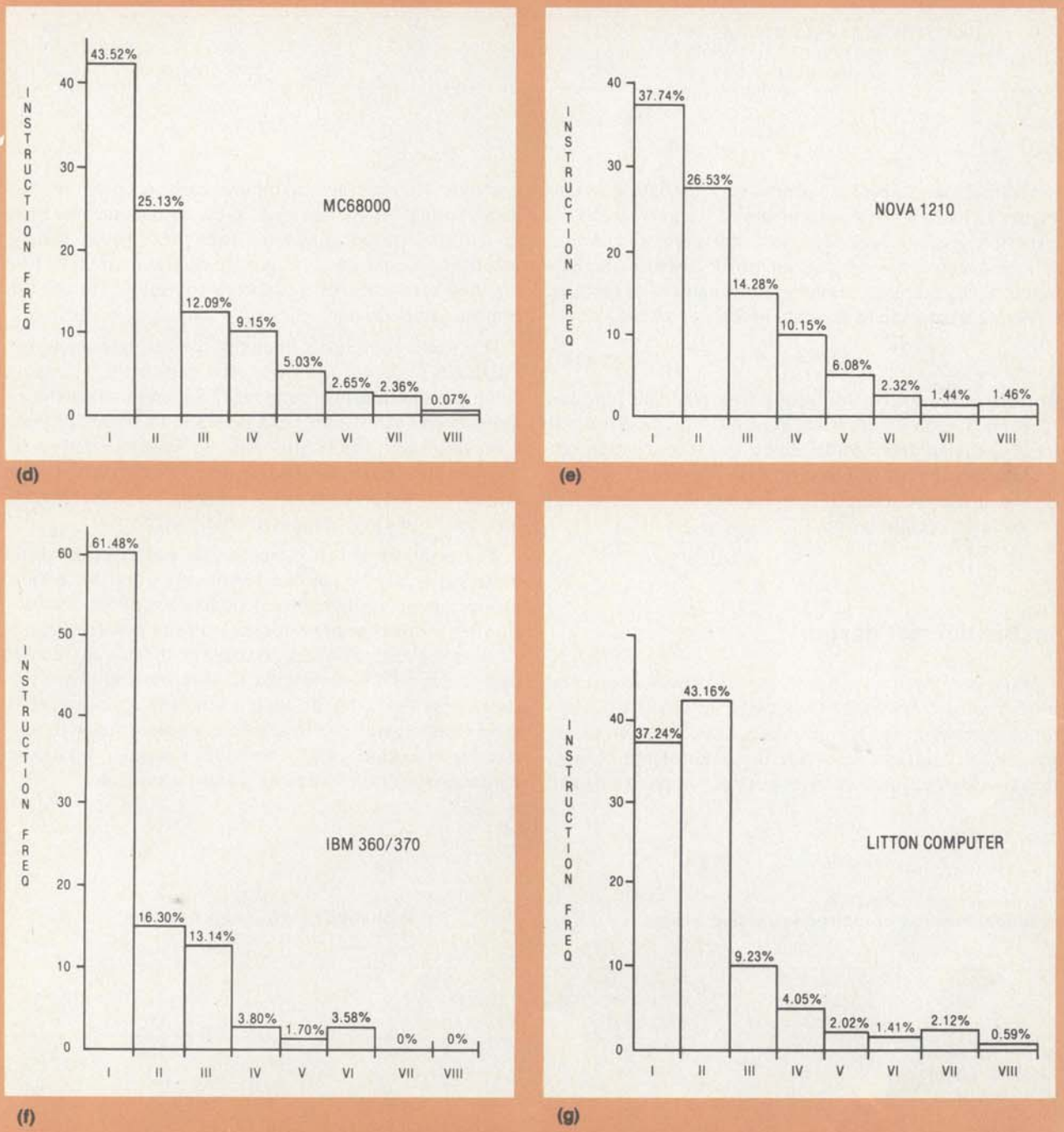


Figure 1. Instruction frequency by instruction group for (a) the MOS6502 microprocessor, a single-operand, 8-bit machine; (b) the MC6800, another single-operand, 8-bit machine; (c) the TMS9900, a double-operand, 16-bit microprocessor; (d) the MC68000, also a double-operand, 16-bit micro; (e) the Nova 1210 minicomputer, a single-

erating system libraries, process control and process monitoring routines, and system utilities.

There are two distinct ways to obtain instruction usage data: one is a static instruction-frequency count; the other is a dynamic instruction-frequency count. The static count is obtained by counting instructions as they appear in a program listing. The dynamic count is obtained by counting instructions as they are executed by the computer. There is an excellent correlation between static and dynamic instruction frequency counts, as shown by Myers[20] and Alexander and Wortman.[5]

The data displayed in Figure 1 are based on static frequency counts. The figure shows the distribution of instruction usage by instruction group for seven machines. In each part of Figure 1, the instruction groups are given roman numeral designations—these appear along the



operand, 16-bit machine; (f) the IBM 360/370, a double-operand, 32-bit computer; and (g) the Litton computer. All data were derived by the author,[21] except those for the IBM 360/370[5] and the Litton computer.[22]

Table 3.
Summary of instruction group means.

ASSEMBLY LANGUAGE PROGRAMS BY COMPUTER TYPE	I DATA MOVE	II PROG. MODIF.	III ARITH.	IV COMP.	V LOGIC.	VI SHIFT	VII BIT	VIII I/O & MISC.
1. MOS6502 (8-BIT)	40.72	26.24	11.49	3.84	5.85	7.04	3.92	0.90
2. MC6800 (8-BIT)	59.52	26.54	4.35	3.68	4.15	1.33	0.43	0.00
3. TMS 9900 (16-BIT)	36.71	37.24	10.64	6.77	2.36	2.15	4.09	0.04
4. MC68000 (16-BIT)	43.52	25.13	12.09	9.15	5.03	2.65	2.36	0.07
5. NOVA1210 (MINI)	37.74	26.53	14.28	10.15	6.08	2.32	1.44	1.46
6. IBM 360/370	61.48	16.30	13.14	3.80	1.70	3.58	0.00	0.00
7. LITTON COMPUTER	37.24	43.16	9.23	4.05	2.20	1.41	2.12	0.59
SUBTOTAL MEAN = 100%	45.28	28.73	10.75	5.92	3.91	2.93	2.05	0.44
STANDARD DEV.	9.89	8.14	3.02	2.57	1.69	1.82	1.46	0.53
VARIANCE	97.72	66.27	9.11	6.63	2.86	3.32	2.14	0.28

horizontal axis. Table 3 is a summary of the data shown in Figure 1. Table 4 shows a summary of the average, by instruction group. Table 4 shows that three instruction groups account for 84 percent of all instructions executed. Five groups account for the remaining 16 percent.

Fitting a function to the data of Table 4 yields

$$f(n) = 2^{-n} \qquad \text{(Equation 2.)}$$

where n is the instruction group number. The function $f(n)$ of Equation 2 is shown in Figure 2. This function will serve as the guideline in designing a new instruction set. The most emphasis will be placed on the groups with the highest projected usage. Table 5 shows the function $f(n)$ for each instruction group.

Instruction set design

What is an optimal instruction set? How is an optimal instruction set designed? One might answer that an optimal instruction set is one that contains the fewest instructions. Minsky[23] showed, using automata theory, that two instructions, *decrement* (and jump if zero) and *add one*, ". . . can do anything an existing computer can do." Turing[24] demonstrated that a simple machine with an infinite serial magnetic tape and some simple algorithms could perform any computational task. But are these theoretical instruction sets optimal? The answer is most obviously no!

Due to the complexity of computer instruction sets, no satisfactory answer has been provided to the question, What is an optimal instruction set? A number of machine-independent algorithms have been shown to be optimal; however, such algorithms can be implemented in a number of different instruction sets. Memory utilization and execution speed vary depending on the programmer, instruction set, and computer system used.

The position of this paper is that a near-optimal instruction set can be constructed based on the instruction group data previously presented. A near-optimal instruction set is one that provides the user the powerful functions he requires. The thesis is that a near-optimal instruction set (or NOIS, as we will call it) is one that allows the user (programmer) to do what is required in an easy, efficient manner, with minimal memory usage, and with reasonable execution speed. The NOIS is a general-purpose instruction set rather than a special-purpose one.

Table 4.
Statistical average of instruction group usage.

		FREQUENCY OF INSTRUCTION GROUP USAGE, %	INSTRUCTION GROUP FREQ. OF USE, CUM. %
I	DATA MOVEMENT GROUP	45	45
II	PROGRAM MOD. GROUP	29	74
III	ARITHMETIC GROUP	10	84
IV	COMPARE GROUP	6	90
V	LOGICAL GROUP	4	94
VI	SHIFT GROUP	3	97
VII	BIT GROUP	2	99
VIII	I/O AND MISC. GROUP	1	100
	TOTAL	100%	

Table 5.
Probability instruction function.

GROUP	$f(n)$	CUM. %
I	50.00%	50.00%
II	25.00	75.00
III	12.50	87.50
IV	6.25	93.75
V	3.13	96.88
VI	1.56	98.44
VII	0.78	99.22
VIII	0.39	99.61

The NOIS provides

- pure instructions,
- pure address modes,
- pure address modifiers,
- an efficient and uniformly coded instruction format,
- orthogonality between instructions and addressing usage, and
- a uniform and consistent appearance to the user.

A pure instruction is one that contains only instruction information and is not contaminated with address-mode or other noninstruction information. A pure address mode or pure address modifier contains only addressing information. The power of this instruction set design is achieved by maintaining purity in all the instruction format fields.

The fewest number of bits should be used to huffman-encode every field in the instruction format; however, in no case should these fields be mixed to obtain additional efficiency. The efficiency obtained by mixing format fields is sacrificed to provide control-unit design efficiency. In the long term this provides improvements in both memory space and execution speed.

Pure instruction formats are effective in providing characteristics such as orthogonality. Orthogonality allows each instruction to use every address mode or modifier in exactly the same manner. This uniformity applies to addressing, word sizes, access methods, and condition-code testing and setting. Orthogonality must be maintained above format-field encoding efficiency. Uniformity, consistency, and purity must be maintained over *all* other considerations. These design restrictions do not have a significant effect on execution speed.

Contrary to advertisements in popular trade journals, a large instruction set is a liability, not an asset. An instruction should be added to an instruction set only when a new function cannot possibly be supported by existing instructions, or when the function can be supported, but only with significant deterioration of programmer or computer system efficiency.

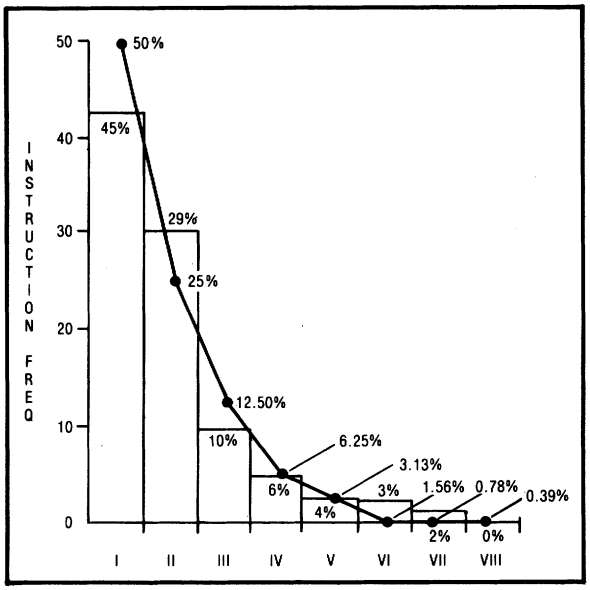


Figure 2. Generalized instruction usage function $f(n) = 2^{-n}$.

Instructions. The designer, when unencumbered by parody, previous architectures, and program-compatibility restrictions, has a rich opportunity to create a near-optimal instruction set. He begins by selecting a single function for each of the eight instruction groups, one that, with the required operands, can encompass the functions of the whole group.

Data movement group. A single instruction will be used to provide the functions required by the data movement group:

MOVE, source, destination

MOVE moves data from the source address to the destination address. The effective address (EA) of both the source- and destination-address fields is determined by the address mode and address modifier.

The MOVE instruction accounts for 50 percent of all instructions executed (see Table 5).

Program modification group. The functions required by the program modification group can also be provided by the MOVE instruction. To provide JUMP or BRANCH operations, the source address is the address of where to BRANCH to, and the destination address is the program counter (PC):

MOVE address, PC

To provide a JUMP or BRANCH operation on condition, the instruction is written

MOVE source, destination, CC

where CC is the condition code. The movement of data from source address to destination occurs only when the condition code is satisfied.

The MOVE instruction may also provide a JUMP or BRANCH to a subroutine on condition:

MOVE source 1, dest 1, source 2, dest 2, CC

where source 1 = program counter, destination 1 = stack or first word of the subroutine, source 2 = subroutine address, destination 2 = program counter, and CC = condition code. The MOVE instruction is executed only if the condition code is satisfied. A RTN (return subroutine on condition) is provided by a two-operand MOVE instruction and with a condition code:

MOVE source, destination, CC

where source = stack or first word of the subroutine, destination = program counter, and CC = condition code.

Note that the first two instruction groups, or 75 percent of all instructions executed, are satisfied by the MOVE instruction and various address types. Decoding of the instruction field (opcode) is simplified significantly by the large reduction in the number of instructions required.

Arithmetic group. Arithmetic instructions require three address fields:

- source 1 = address of value 1 (arith),
- source 2 = address of value 2 (arith), and
- destination = addr of result (arith) = value 1 op value 2.

For addition, an ADD instruction is provided:

ADD source 1, source 2, destination

Subtraction is provided by the instruction:

SUB source 1, source 2, destination

Multiplication and division are provided by the instructions:

MULT source 1, source 2, destination

DIV source 1, source 2, destination

The MULT and DIV instructions create problems related to results, value sizes, precision, and sign extension, however. Such problems have been solved before—their solutions are well-documented in the literature.

Five instructions now account for 87.5 percent of all instructions executed (Table 5).

Compare group. All of the instructions in the compare instruction group may be satisfied by the instruction:

SUB source 1, source 2, destination

where source 1 = value 1, source 2 = value 2, and destination = status register = value 1 − value 2. An arithmetic COMPARE is really just a subtraction that has the status register as its destination address.

Five instructions now account for 93.75 percent of all instructions executed.

Logical group. The logical instruction group cannot be supported by any of the previously defined instructions. The address fields required are

- source 1 = address of value 1 (boolean),
- source 2 = address of value 2 (boolean), and
- destination = result (boolean) = value 1 operator value 2.

The three essential boolean operators are AND, OR, and XOR (the exclusive OR):

- AND = source 1, source 2, destination,
- OR = source 1, source 2, destination, and
- XOR = source 1, source 2, destination.

The eight instructions defined thus far now represent 96.88 percent of all instructions executed.

Shift group. The shift instruction group also cannot be supported by any previously defined instruction. A SHIFT instruction, then, must be added to the instruction set:

SHIFT source, destination, type, count

The address modes to be provided are

- source = address value to be shifted,
- destination = destination address of the shifted value,
- direction = shift direction (right/left),
- type = shift type (arith/logical), and
- count = shift count 1 through maximum − 1.

Nine instructions now represent 98.44 percent of all instructions executed.

Bit group. Of all the instruction groups, this one is the most widely acclaimed and least used. Only one instruction, MOVE bit (MOVEB), is provided for this group:

MOVEB source, destination

where source = source bit address, and destination = destination bit address.

Ten instructions now account for 99.22 percent of all instructions executed.

Input/output and miscellaneous group. No additional instructions are provided for this group. All the microprocessors analyzed used memory-mapped I/O, which opens pins on the integrated circuit package for other important functions without placing a significant restraint on the I/O capability of the microprocessor. Interrupts are handled by hardware; the return from an interrupt uses a four-operand MOVE instruction to place the return address in the program counter and the status in the status register.

Since no additional instructions are needed for I/O, just ten instructions account for 100 percent of all instructions executed.

Summary of basic instructions. By group, the ten instructions are

I	Data movement	**MOVE**
II	Program modification	(MOVE)
III	Arithmetic	**ADD, SUB, MULT,** and **DIV**
IV	Compare	(SUB)
V	Logical	**AND, OR, XOR**
VI	Shift	**SHIFT**
VII	Bit	**MOVEB**
VIII	I/O and misc.	none

Extended instructions. There are a few other instructions that are not essential to the instruction set, but which add significantly to its programming flexibility. By examining the data on arithmetic instruction frequency, we can see that 2.5 percent of the instructions are increment instructions and 2.5 percent are decrement instructions. These high percentages justify the extension of the set to include such instructions. The instructions to increment by one to four and decrement by one to four are

INC value, destination

DEC value, destination

Combined instructions. Shustek[17] and Stritter[19] performed research to determine if any two or more instructions could be effectively combined into a single instruction. The author researched this possibility on the previously discussed microprocessors and concluded that very few useful combined instructions could be identified. Only two were determined to be significant enough to be included in the instruction set:

I/DBRC source 1, destination, CC

MOVEM source 1, source 2, destination, count

I/DBRC (increment/decrement and BRANCH on condition code) tests the source to be equal to the condition code. If the equality is satisfied, the BRANCH to the destination address is not taken. If the equality is not

satisfied, the branch is taken and the source address is incremented or decremented by one, prior to the BRANCH to the destination address. The I/DBRC instruction is very useful in loop control.

The MOVEM (MOVE multiple) instruction moves the number of bytes in the count field from the source address(es) to the destination address(es). MOVEM is useful in string manipulation.

Restrictiveness and efficiency

One might argue that the instruction set is unnecessarily restrictive; however, it is adequate to efficiently perform every necessary function. The question that then arises is, How well? This was determined by comparing the number of NOIS instructions required to implement various compiler functions to the number of MC68000 instructions required to implement those functions. Programs were first written in MC68000 Pascal and compiled. The compiled code was then disassembled into MC68000 assembler mnemonics and also coded in NOIS assembler mnemonics. The results of this comparison are shown in Table 6.

No attempt was made to optimize the code as generated by the MC68000 Pascal compiler. The MC68000 instructions were replaced by the NOIS instructions on a line-by-line basis. With some optimization, the efficiency of the NOIS instructions over the MC68000 instructions could have been increased by another 15 to 20 percent over that indicated by Table 6. This optimization would have made the NOIS 25 percent to 42 percent more efficient than the MC68000 instructions.

When instructions are separated from instruction formats and analyzed as pure instructions, an interesting number of things become clear. One is that the power in a computer system resides in surprisingly few instructions. In the case of the NOIS, there are only ten basic and four extended instructions. If the instruction groups for the NOIS follow the $f(n)=2^{-n}$ function, one instruction, MOVE, will represent 75 percent of all instructions executed, and four instructions, MOVE, ADD/SUB, MULT, and DIV, will represent 87.5 percent of all instructions executed.

A single bit may be used for the MOVE operation code, and a maximum of four to eight bits for all other operation codes. This minimal encoding frees all the remaining bits for use in encoding the address mode, address modifier, word size, and condition code fields.

The power of the NOIS is provided by a few pure instructions and a very rich addressing capability—further research has revealed that most of the power comes from the richness of the latter. Moreover, the importance and capability of pure addressing modes are almost as significant as the pure instructions. Address usage data is the basis on which to develop addressing modes and modifiers.

Table 6.
Number of instructions used in assembly language routines—NOIS vs. MC68000.

PASCAL CONSTRUCT	NOIS INSTRUCTIONS VS. MC68000 INSTRUCTIONS
FOR	−22%
WHILE	− 9%
IF THEN/ELSE	−15%
REPEAT-UNTIL	−10%

Floating-point instructions were not included in the NOIS. Such instructions can be provided on a separate chip. Other capabilities, such as transcendental and BCD functions, can also be placed on a separate chip.

The near-optimal instruction set is based on an analysis of instruction usage by groups rather than by individual instructions. The grouping of instructions by function allowed addressing modes and individual machine idiosyncrasies to be stripped out. The function $f(n)=2^{-n}$, where n is the instruction group number, allowed the author to predict how the NOIS will be used.

The greatest effort was placed on creating and optimizing the instructions in the first four groups, which represent 93.75 percent of all instructions that will be executed. Less effort was allocated to the remaining instructions in the other groups.

The NOIS is a powerful yet simple tool. Its instructions are few and easy to use and remember. They are also orthogonal to the addressing modes.

A simulator for the NOIS was written in Pascal and was used to explore various algorithms. This exercise produced concise and efficient code. The next task will be to build the hardware needed to execute the near-optimal instruction set. ■

Acknowledgments

The author would like to acknowledge a number of people at Brigham Young University, Wicat, Inc., and Novell Data Systems, Inc., for their assistance. The number of individuals is so large that their names cannot be listed here, unfortunately. There are two individuals who must be recognized above all others—Dr. Dustin Hueston of Wicat and Dr. Jens J. Jonsson of Brigham Young University.

References

1. C. C. Foster, "A View of Computer Architecture," *Comm. ACM,* Vol. 15, No. 5, July 1972, pp. 557-565.
2. K. L. Bowles, *Problem Solving Using PASCAL,* Springer-Verlag, New York, 1977.
3. E. H. Herbst, N. Metropolis, and M. B. Wells, "Analysis of Problem Codes on the MANIAC," *Math. Tables Other Aids Computer,* Vol. 9. No. 49, Jan. 1945, pp. 14-20.
4. C. C. Foster, R. H. Gonter, and E. M. Riseman, "Measures of Op-Code Utilization," *IEEE Trans. Computers,* Vol. C-20, No. 5, May 1971, pp. 582-584.
5. A. G. Alexander and D. B. Wortman, "Static and Dynamic Characteristics of XPL Programs," *Computer,* Vol. 8, No. 11, Nov. 1975, pp. 41-46.
6. D. E. Knuth, "Computer Programming as an Art," *Comm. ACM*, Vol. 17, No. 12, Dec. 1974, pp. 667-673.
7. J. C. Gibson, "The Gibson Mix," IBM Tech. Report TR-00.2043, June 18, 1970.
8. R. A. Arbuckle, "Computer Analysis and Throughput Evaluation," *Computers and Automation*, Jan. 1966, pp. 12-19.
9. D. P. Agarwal, "Design of an Efficient Instruction Set," tech. report, Dept. of Computer Science, Carnegie-Mellon University, 1973.
10. A. Lunde, "Evaluation of Instruction Set Processor Architecture by Program Tracings," tech. report, Dept. of Computer Science, Carnegie-Mellon University, July 1974.
11. H. Schreiber, "Hardware Measurement of CPU Activities," *Modelling and Performance Evaluation of Computer Systems,* North-Holland Pub. Co., Amsterdam, 1977.
12. C. C. Foster and R. Gonter, "Conditional Interpretation of Operation Codes," *IEEE Trans. Computers,* Vol. C-20, No. 1, Jan. 1971, pp. 108-111.
13. D. W. Ashley, "A Methodology for Large Systems Performance Prediction," IBM Tech. Report TR-00.1773, Sept. 10, 1968.
14. *MC68000 Microprocessor User's Manual*, 2d ed., Motorola Semiconductor Products, Inc., Jan. 1980.
15. D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples,* McGraw-Hill, New York, 1982, pp. 54-55.
16. A. G. Alexander, "How a Programming Language Is Used," Tech. Report CSRG-10, Computer Research Group, University of Toronto, Feb. 1972.
17. L. J. Shustek, "Analysis and Performance of Computer Instruction Sets," PhD dissertation, Stanford University, Aug. 1978.
18. M. Shima, "Demystifying Microprocessor Design," *IEEE Spectrum*, Vol. 16, No. 7, July 1979, pp. 22-30.
19. E. Stritter and T. Gunter, "A Microprocessor Architecture for a Changing World: The Motorola 68000," *Computer*, Vol. 12, No. 2, Feb. 1979, pp. 43-52.
20. G. J. Myers, *Advances in Computer Architecture*, John Wiley and Sons, New York, 1978, pp. 273-291.
21. D. A. Fairclough, "Microprocessor Instruction Groups," unpublished paper, Dept. of Electrical Eng., Brigham Young University, Apr. 1980.
22. C. C. Church, "Computer Instruction Repertoire—Time for a Change," *AFIPS Conf. Proc.*, Vol. 36, 1970 SJCC, pp. 343-349.
23. M. L. Minsky, *Computation of Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, N.J., 1967, p. 207.
24. A. M. Turing, "On Computable Numbers," *Proc. London Math. Soc.*, Ser. 2, Vol. 42, 1936, pp. 230-265.

Dennis A. Fairclough is an assistant professor in the Department of Electrical Engineering at Brigham Young University. He does research, teaches, consults, and publishes in the areas of computer architecture, high-performance microprocessor systems, and intelligent disk subsystems. He received the BSEE in 1962 from the University of Utah, the MSEE in 1968 from the University of Santa Clara, and is completing work for a PhD in electrical engineering from Brigham Young University. He is a member of the IEEE, the ACM, Phi Kappa Phi, Tau Beta Pi, and Eta Kappa Nu.

Fairclough's address is Dept. of Electrical Engineering, 468 Clyde Bldg., Brigham Young University, Provo, UT 84602.