Hal T. Gordon
University of CA, Berkeley
Berkeley CA 94720

Instruction Sets Examined and Compared

Ever wonder how the 8080, Z-80, 6800, 6502 and 2650 differ from each other? Some observations about this question resulted from research that went into the following article.

The capability of a microcomputer system largely resides in the instruction set of its CPU chip, although support chips, peripherals and skillful programming are needed for the full realization of this capability. Even users who program only in high-level languages ultimately rely on the power of an instruction set, since the interpreter or compiler programs they need must be written in the machine language of a specific microprocessor.

In volume II of his *Introduction to Microcomputers*, Adam Osborne presents detailed summaries of the instruction sets of many microprocessor designs, and three articles by Lance Leventhal in *Kilobaud* (July, August and September 1977) contain useful discussion of sets at a more general level. It seems to me that none of these ever quite comes to grips with the question: What is it that makes one set more or less versatile than another?

Although Osborne encodes a simple benchmark program in each of many sets, he rightly stresses that this is not an adequate criterion for overall evaluation. If this were done for a much larger and more diverse set of test programs (including complex ones), you would have empirical criteria of capability: (1) Which instruction set encodes them (on the average) with the fewest program bytes? (2) Which runs fastest? (3) Which requires the least programming and debugging time?

Since all these criteria are highly dependent on programming expertise, they ought to be first measured for a group of experts (to estimate the true potential of each set) and then for a group of amateurs (to estimate performance at the user level). It is unlikely that this objective testing will ever be done. All we shall have to go by are the claims of the manufacturers of CPUs and systems, and the more-or-less educated guesses of experts (who are not always in agreement).

This article is a nonexpert view, based on a long-standing fascination with instruction sets. I look upon these as intellectual works of art that reflect not merely the technical experience but the imagination and personality of their creators.

> "People have favorite friends, books, tools and games. Using an instruction set is very much like playing a game."

Before going on, I should answer two questions. First, do the differences between instruction sets *really* lead to significant differences in performance? Unquestionably they do, but – since performance is a complex concept, and each design has its unique strengths, and the instruction set is by no means the *only* design element that determines performance – we cannot easily rank chips in order of performance. There is one exception: The Z-80 will *always* perform as well as or better than the 8080.

Second, do I have a personal bias? Yes, I prefer the simplicity of the 6502. This does not mean that this set is the "best" or that I am unable to appreciate others (the Z-80 is certainly more capable) or that I think everyone should react as I do. On the contrary, many users will find other sets more attractive. People have favorite friends, books, tools and games. Using an instruction set is very much like playing a game.

Many different designs exist. *All* can do *everything*, and do it very well. In such circumstances, the fact that one may excel the others in many ways ceases to be an *overriding* consideration, and subjective factors (hard to explain to a computer!) can enter in. If power were the sole determinant, the PDP-8 mini would have vanished when the PDP-11 was created, and the Z-80 would by now have obliterated the 8080. I expected this to happen, not realizing that if

you have all the capability you need, why bother to get more?

Instruction Op Codes

When the control unit of a CPU "reads" an instruction operation code from a program, it copies its bit-pattern into its *control register*. There, it triggers a complex logic-network of gates, causing specific, planned modifications of the bit-pattern in one or more on-chip registers and/or external memory locations. On-chip operations run at lightning speed (propagation delays measured in nanoseconds). Communication with external locations, enabled by a clock signal, is slower because the bus lines have longer paths and higher capacitances.

Although the number of *possible* operations encodable in logic-networks is extremely large, there is a practical limit to the number that can be fitted into one LSI chip. Every microprocessor designer gives much thought to selecting only the ones he believes will be very useful.

Another design goal is to have one-byte op codes (of which, with eight bits, there can be no more than 256) to minimize the time involved in accessing program memory. Most existing designs do not use all the 256 possible eight-bit patterns as op codes (the 8080 uses 244, the 6800 uses 197, the 6502 uses only 151). However, some of the unused patterns will be executed by the control unit of the CPU; such "illegal" instructions may yield odd or even useful results.

There are three fundamental types of instructions: (1) those that simply *move* (actually, *copy*) a bit-pattern from one location into another; (2) those I shall refer to as "thinking operations" that *modify* or *analyze* bit-patterns; and (3) those that cause a *jump* or *branch* to an instruction other than the next one in sequence. Many types are so useful that *all* sets have them (e.g., MOVEs or logical ANDs between an on-chip register and any external memory location). Other types are omitted in some designs, or in all but one.

The strength of the Z-80 largely rests on its having the greatest variety of types, omitting relatively few of those present in other designs and adding many unique ones. No one will deny that the Z-80 set is more capable than that of the 8080, since it *includes* all of the 8080 instructions and *adds* to it many useful others. The inclusion of an older set, however, is not wholly positive since you retain not only its strengths but also its weaknesses. Also (as Adam Osborne has pointed out), the 8080 set uses 244 of the possible 256 one-byte op codes.

To enlarge the set, the Z-80 needs 382 two-byte and 62 three-byte op codes that load and run more slowly. This is one reason why the Z-80 *needs* a faster clock and high-speed memory. Four of the 12 bit-patterns not used in the 8080 set are used by the Z-80 as the first byte of its multibyte op codes, while the remaining eight are used as new one-byte op codes. *How* the Z-80 designers used this precious residue of fast codes is a valuable lesson in what *really* enhances a set, as we shall shortly see.

Strange as it may seem, an instruction type can be *too* powerful. The 8080 set has eight jump-on-condition instructions that allow the program to leap to *any* location in memory, but require a two-byte (absolute) address. Since these are among the most often-used instructions, such addressing increases both program bytes and execution time (especially in loops). Also, if a program is relocated in memory, every address has to be altered.

In the sets of the 6800 and 6502, there are only *relative* branch-on-condition instructions, whose op codes require only *one* address byte (interpreted as a signed binary number that is added to the program counter) but are limited to leaps in the range of +127 to -128 from the current program counter address. It is possible (though not efficient) for 6800/6502 programs to emulate the long 8080 conditional leaps by combining a conditional branch with their unconditional jump-absolute instruction, but in actuality this is almost never necessary.

> "The value of relative-branching is proved by the fact that Z-80 designers used six of their eight new one-byte op codes to create relative-branch instructions."

For example, the 2K ROM monitor of the MOS Technology KIM-1 has 752 instructions. Of these, 31 (4.1 percent of the total) are jump-absolutes, not one of which is conditional. The range limit of the 103 relative-branch instructions (13.7 percent of the total) is easily handled by careful program structuring (i.e., locating every block so that it lies within the range of the branchings to it).

If we compare the 687-byte 8080-Simulator Program by Lee Stork (September 1977 *Kilobaud*), we find that 14 (5 percent) of its 283 instructions are unconditional jumps and 39 (13.8 percent) are conditional ones. Of the latter, 26 are within a ± 127 range, and most (probably all) of the others could be brought within this range by program restructuring (although in an 8080 program there is no reason to do so).

The value of relative-branching is proved by the fact that Z-80 designers used six of their eight new one-byte op codes to create relative-branch instructions. One of these is unconditional, like the BRA (BRAnch) of the 6800 set, a fast short-range replacement for the 8080 JMP. It is interesting that the 6502 — with its vast supply of unused op codes — did not include a BRA. It can easily emulate it (at the cost of one more byte) by a "forced branch": clear a flag, then branch-if-flag-clear. Neither the 6502 nor the Z-80 adopted the 6800 BRS (unconditional relative-branch-to-subroutine). In fact, BRA and BRS can in no way eliminate their two-byte address equivalents (JMP and JSR), the essential long-leap instructions of the 6800/6502 sets.

Conditional jump (or branch) instructions occur frequently in programs because they are the decision/switching points. A *simple* condition, indicated by a single status flag bit, has two instructions: jump-if-flag-set (to 1) and jump-if-flag-reset (to 0). Two or more flag bits show a *complex* condition.

Many instructions alter more than one flag. For example, the COMPARE instruction, in effect, subtracts the content of some location X from the content of the accumulator A, but alters only the status register. If A $<$ X, the carry flag is set by the 8080, Z-80 and 6800, while if A $\geqslant$ X the carry is cleared (but in the 6502 the carry status is the exact *opposite*). If A = X, the zero flag is set. Only the 6800 has single instructions that (by testing two flags) branch if A $>$ X (BHI) or A $\leqslant$ X (BLS). The others need a sequence of two instructions.

For example, to jump to the address HAWAII if A $\leqslant$ X, the 8080 needs a JC HAWAII followed by a JZ HAWAII, while the 6800 needs only a BLS HAWAII. The Signetics 2650 COMPARE does not involve the carry, but two condition-code

bits set to 00 if A = X, 01 if A > X, and 10 if A < X. It therefore needs two branch instructions to act on either A ≥ X or A ≤ X.

It is noteworthy that four of the six new Z-80 relative-branch instructions test the carry and zero flags, allowing it to react to the most important conditions much faster than the 8080. The other addition (DJNZ) decrements the B register and branches if it is not zero, allowing this register to efficiently control loops.

The 8080 set also includes eight conditional jump-to-subroutines and eight conditional returns. It is hard to tell how useful these are. All 16 CALLs in the Lee Stork program referred to above, and three of its four returns, are unconditional. Such instructions are not indispensable since the 6800/6502 get along well without them. However, the Signetics 2650 has six conditional subroutine calls (three absolute and three relative) and one conditional return.

All sets have the classic "thinking instructions": the logical AND, OR and exclusive OR that compare two bit-patterns on a bit-by-bit basis (always eight independent comparisons) and the arithmetic add, subtract and compare that treat the bit-pattern as a binary number. All have some rotate instructions that allow another kind of bit analysis and modification. The 6800 and 6502 also have arithmetic and logical shifts. The Z-80 includes *everything*, plus two tricky new ones (RRD and RLD).

I shall not attempt to explain the varied construction and use of these operations. I feel that all sets have enough power to do the most important and often-used things efficiently, and can, if necessary, emulate anything they lack by using a sequence of instructions.

One problem with all sets is that some instructions will rarely or never be used. For example, how often are the seven MOV R,R 8080 instructions (that move the content of one of the on-chip registers into the *same* register!) ever used in programs? This is one reason the mere *number* of instructions is not an ideal index of power.

The 6502 has one of the smallest sets, but even so, 37 (24%) of its 151 instruction op codes are not used in the KIM-1 ROM monitor. The percentage of non-utilization is likely to be much higher for the giant Z-80 set, especially since many of its new instructions are better than equivalent ones in its 8080 subset. However, the statistics of usage frequency (except for zero usage) are likely to be misleading.

Some instructions are essential, even though not often used, while others may be frequently used simply because more effective ones are not present in the set. Programmers learn to make do with whatever is available, and even tend to adopt a "mental subset" of instructions that they like – even when a task could be programmed as well or better by using less-favored instructions.

Experts can recognize programs written by amateurs, because they fail to use the full power of the set, and may even recognize a program written by a fellow-expert by its characteristic skillful exploitation of some instruction types. As with English writing, each tends to develop an individual style because every set is rich enough to allow one to "say" the same thing in a great variety of ways.

Not only do some instructions get neglected; others are (also for psychological reasons) not fully exploited, at least until one programmer breaks the ice. I recently discovered an example of such a "programmer mental block" involving the BIT instruction (absent in the 8080, present in different forms in the 6800, 6502 and Z-80). This is a logical AND between the primary accumulator (the on-chip register involved in the greatest number of instructions) and another location, which alters only the status register (whereas the conventional AND replaces the bit-pattern in the accumulator by the ANDed pattern).

Users of AND or BIT think of one of the two bit-patterns as a "mask" to clear or test bits in the other pattern. For example, if the bit-pattern being tested is X0X0X0X0, a mask of 01010101 will set the zero flag, thereby revealing that bits 0, 2, 4 and 6 were all zero.

"The 6502 has one of the smallest sets, but even so, 37 (24 percent) of its 151 instruction op codes are not used in the KIM-1 ROM monitor."

The 6800 BIT has both immediate addressing (obviously to test an unknown bit-pattern in the accumulator by the mask of the program operand byte) and memory addressing (to test bits in a memory location by a mask previously loaded into the accumulator). In the 6502 set the BIT-immediate is omitted, creating the false impression that BIT can now only test bits in memory (although BIT logic neither knows nor cares where the "mask" is).

As far as I know, the first violation of this "test only bits in memory" rule – based on a mask concept existing only in the human mind – was in the 6502 Tracer program by Larry Fish (August 1977 *Kilobaud*), which used masks preset in memory locations to test bits in the accumulator. This is very useful, though not quite as fast or convenient as the 6800 BIT-immediate.

BIT is interesting as a specialized instruction that can easily be emulated by the conventional AND (although not exactly in its enhanced 6502 version, which also sets the overflow flag equal to bit 6 of the memory location), but will save some bytes and time in a program. The designers of the Signetics 2650 did not include BIT in its set. Instead, they added the unique TMI instruction, which also nondestructively compares the bit-pattern in an on-chip register with an immediate-operand bit-pattern.

If all the ones in the operand are also ones in the register, two "condition-code" bits in a status register are cleared. TMI is a kind of "reverse-BIT" that can test any "internal pattern" of ones, instead of "internal patterns" of zeros. It is harder to emulate with conventional logic, since the tested pattern must be complemented before being ANDed with the mask.

A set with both TMI and BIT would have no peer in its bit-analysis capability. This statement may come as a shock to admirers of the Z-80 who know that it has no less than 80 distinct BIT instructions in its arsenal! Although the Z-80 BIT has the same name, it is a less powerful instruction because it can test only single bits, not internal patterns of zero bits. The 6800/6502 BIT operates between two locations. Either one may contain a mask of any of the 256 possible eight-bit patterns. The Z-80 BIT operates on a single location without an explicit mask (the single-bit being tested is implied in the op code).■

Next time, we'll continue our examination of instruction sets.