# DECODING 650X opcodes

H. T. Gordon,
Dept. of Etomology
University of California
Berkeley, CA. 94720

*Dear Mr. Warren,* *Aug. 2, 1977*

*This is a write-up on 650X opcodes. The two programs in it are system-independent and so not right for KIM-1 USER NOTES (that doesn't like subroutines anyhow).*

*By the way, my earlier note on a STRINGOUT revision (that you published) had a few typos in the program. I've not bothered to send corrigenda because anyone who knows what he's doing will see them right away and few novices will read DDJ. In the programs I send herewith, with more complex logic, typos might give potential users a few headaches. I agree that direct reproduction of teletype output would be better, but some of us churchmice still don't have them.*

*Sincerely,*
*H. T. Gordon*

Three recently-published programs (a debugger by Larry Fish in the Aug. '77 Kilobaud, and relocators by Ralph Sherman in the April '77 *DDJ* and by Jim Butterfield in the #4 '77 Kim-1 user notes) include routines for calculating the number of bytes required by a 650X opcode. All use quite different logic, and none is richly documented or coded as an independent subroutine. This decoding operation is a wheel that has probably been reinvented many times (I did it as an early programming exercise long ago, as a not-very-efficient 51-byte subroutine). The following table shows the intricacy of the problem. It lists the 16 opcode types from XØ to XF, roughly in order of usage frequency in programs (650X programmers try to avoid using 3-byte codes!). Decoding execution time will be shorter if common codes are the earliest decoded, when this is compatible with an efficient bit-sifting routine. Types XØ and X9 are unusual in that the number of bytes is determined by X (the term $X_o$ means an even number and $X_1$ an odd number). Although the last 4 types are all illegal, coding errors may cause them; since they make up 60% of all illegal opcodes and are easy to sift out, this may be worth doing (but only the Butterfield program does it).

The Sherman program uses mostly (AND, CMP) logic. It sifts out all 1-byte opcodes in 4 steps: ØØ, then 2Ø, then (4,6)Ø, then X(8,A), then all 3-byte opcodes in 3 steps: $X_1$ (C,D,E,F), then $X_1$ (9,B), then $X_o$ (C,D,E,F). Residuals are 2-byte opcodes. The Butterfield program uses a sequence of seven (AND,EOR) siftings in an indexed loop, addressing a 22-byte table of operands: first the illegals X(3,7,B,F), then 2Ø, then the $X_1$Ø branches, then (Ø,4,6)Ø and (Ø-7)8, then (8-F)8 and XA, then $X_1$9, then X(C,D,E). Residuals are 2-byte opcodes. Although ingenious and powerful, the program optimizes byte-economy at the cost of longer execution time.

| Type | X | Bytes | # legal | # illegal |
|---|---|---|---|---|
| XØ | Ø, 4, 6 | 1 | 3 | Ø |
| XØ* | $X_1$ | 2 | 8 | Ø |
| XØ | $X_o > 7$ | 2 | 3 | 1 |
| XØ | 2 | 3 | 1 | Ø |
| X8 | | 1 | 16 | Ø |
| XA | | 1 | 10 | 6 |
| X1 | | 2 | 16 | Ø |
| X5 | | 2 | 16 | Ø |
| X6 | | 2 | 16 | Ø |
| X9 | $X_o$ | 2 | 7 | 1 |
| X9 | $X_1$ | 3 | 8 | Ø |
| X4 | | 2 | 7 | 9 |
| X2 | | 2 | 1 | 15 |
| XC | | 3 | 8 | 8 |
| XD | | 3 | 16 | Ø |
| XE | | 3 | 16 | Ø |
| X3 | | | Ø | 16 |
| X7 | | | Ø | 16 |
| XB | | | Ø | 16 |
| XF | | | Ø | 16 |

*includes all branch opcodes

The Fish program relies mostly on the 650X BIT instruction. Although suboptimally coded, it heightened my awareness of the power of BIT, not merely for detecting the presence or absence of single bits but (equally important) the simultaneous absence of 2 or more bits. The original program required 6 bit-masks in zero-page and had one error (that was corrected in a much more efficient revision sent to me by the author). I shall not analyze his bit-sifting operations, except to note that the very clever idea of splitting them into 2 branches (one for types X(Ø-7), the other for X(8-F)) was his. The following revision (further optimized and coded as a subroutine by me) saves both program bytes and execution time. The subroutine expects to find an opcode in the accumulator, and returns the correct number of bytes in the X register.

```
Ø21Ø  A2 Ø1     BYTNUM LDX #$Ø1 (sets 1-byte exit)
Ø212  2C 25 Ø2  BYTNOX BIT TRICK+4 (tests bit 3)
Ø215  DØ ØF            BNE HALFOP  (all X(8-F))
Ø217  2C 22 Ø2         BIT TRICK+1 (tests bits Ø-4, 7)
Ø21A  DØ 15            BNE 2BYTE  (all but (Ø,2,4,6)Ø)
Ø21C  C9 2Ø            CMP #$2Ø  (compare $2Ø)
Ø21E  FØ 1Ø            BEQ 3BYTE  (3-byte if =)
Ø22Ø  6Ø               RTS   (3 residuals 1-byte)
Ø221  A2 9F     TRICK  LDX #$9F  (hidden data *)
Ø223  Ø5 14            ORA DUMMY       "    "
Ø225  Ø8               PHP             "    "
Ø226  2C 23 Ø2  HALFOP BIT TRICK+2 (tests bits Ø,2)
Ø229  FØ Ø7            BEQ 1BYTE  (all X(8,A))
Ø22B  2C 24 Ø2         BIT TRICK+3 (tests bits 2,4)
Ø22E  FØ Ø1            BEQ 2BYTE  (all Xo(9,B))
                (3-byte residuals X1(9,B) and X(C,D,E,F))
Ø23Ø  E8        3BYTE  INX
Ø231  E8        2BYTE  INX
Ø232  6Ø        1BYTE  RTS
```

*These are valid instructions that cannot be reached in program execution, but 4 of the 5 bytes serve as data operands for the BIT instructions, eliminating a data table. This trickery (suggested by a novel step in the Butterfield program, branching to a ØØ operand as a BRK instruction) would hopefully pass inspection by simple assemblers or debuggers!

The operation should be fast since neither of its 2 branches involves more than 3 bit-tests and 3 branchings; most of the common opcodes are decoded even faster. Of its 35 bytes, the 5 "trick" bytes serve to make it self-contained and functional in any 560X system. In any actual system, however, not all of them may be necessary, since most of them have large ROM programs that are a treasurehouse of bytes, at fixed addresses that make them usable as BIT masks. The subroutine would then become system-dependent; e.g., in a KIM-1 system there is an Ø8 at 1EB3 and a 14 at 1C95, so one could save 3 bytes by using only Ø5 9F in the TRICK sequence. If one can find all required mask bytes in ROM, the program will need only 30 bytes and become fully relocatable.

The main program can set the X register (e.g., to ØØ or FF) and bypass the BYTNUM setting by using a JSR BYTNOX. Operation affects only the X and status registers, e.g. the Z flag is set only by X(8,A) and is = bit 3 if both bits Ø and 2 are = Ø, while the V flag (unused by BYTNUM) is always = bit 6. The main program can add any or all of the special operations of the Sherman and Butterfield programs. The special handling of ØØ would be invoked by a BEQ after loading the opcode. Isolation of branch opcodes would be done after the return by 6 bytes: AND #$1F, CMP #$1Ø, BEQ BRANCH. I am less enthusiastic about the screening-out of 64 of the 104 illegals, and I have therefore developed an independent legality-testing subroutine.

There are some special problems in legality testing. E.g., early versions of the 650X lacked the ROR instruction and had only 147 legal opcodes instead of the 152 in the current version. There are 2 kinds of "illegals": many are interpreted as valid instructions and are executed by the 650X, while others seem to be blind alleys that halt further operations. E.g., "valid illegals" such as XF cause execution of *both* of the legals XD and XE, while "invalids" such as X2 (where X ≠ A) fail to execute. (I sent a note on this to *BYTE* long ago, that was accepted but has not yet been printed.) Also, there is added logical complexity in decoding the 6 types whose legality is determined by X, as shown in the following table of legal X values:

X2 only X = A
XØ all X except 8
X9 all X except 8
XA all $X_o$, plus 9 and B
XC all $X_o$ (except Ø), plus B
X4 all $X_o$ (except Ø,4,6), plus 9 and B

My attempts to program this using the BIT, that was so effective in BYTNUM, were so inefficient that I changed to a somewhat unusual logic, relying on a sequence of LSRs (that right-shift the opcode, lowest bit into the carry flag) to create extensive branch decisions. Like most first tries, the program must be suboptimal, especially since I have not had the advantage of seeing other legality programs (although the specs for the ECD MicroMind imply that such testing is done in their loading from tape cassettes).

The program assumes that an opcode is in the accumulator. It acts as a filter, causing a program break if the code is illegal. Although operation destroys the byte in the accumulator, it is preserved intact in the X register, so that it can be restored by a TXA in the main program after the return.

```
Ø24Ø  AA         OPLEGL  TAX
Ø241  4A                 LSR A   (bit Ø → carry)
Ø242  9Ø Ø9              BCC TYPEØ2   (all evens)
Ø244  4A                 LSR A   (odds, bit 1 → carry)
Ø245  BØ 14              BCS ILLEGA   (all X(3,7,B,F)
Ø247  8A                 TXA   (restore opcode)
Ø248  C9 89              CMP #$89 (compare to 89)
Ø24A  FØ ØF              BEQ ILLEGA (89 is illegal)
Ø24C  6Ø                 RTS   (all other X(1,5,9,D))
Ø24D  4A         TYPEØ2  LSR A (evens, bit 1 → carry)
Ø24E  9Ø 17              BCC TYPEØ (all X(Ø,4,8,C))
Ø25Ø  4A                 LSR A   (bit 2 → carry)
Ø251  9Ø Ø1              BCC TYPE2A   (all X(2,A))
Ø253  6Ø                 RTS   (all X(6,E))
Ø254  4A         TYPE2A  LSR A   (bit 3 → carry)
Ø255  BØ Ø5              BCS TYP4AC   (all X(A))
Ø257  C9 ØA              CMP #$ØA (tests for X = A)
Ø259  FØ Ø4              BEQ LEGALA   (A2 is legal)
Ø25B  ØØ         ILLEGA  BRK   (other X2 illegal)
Ø25C  4A         TYP4AC  LSR A   (bit 4 → carry)
Ø25D  BØ Ø1              BCS ODDX   (all odd X)
Ø25F  6Ø         LEGALA  RTS   (residual even X)
Ø26Ø  29 Ø6      ODDX    AND #$Ø6 (tests X = 9,B)
Ø262  C9 Ø4              CMP #$Ø4   (must = Ø4)
Ø264  DØ 15              BNE NOTLEG (illegal X1)
```

```
Ø266  6Ø                RTS  (legals X = 9,B)
Ø267  4A       TYPEØ    LSR A  (bit 2 → carry)
Ø268  BØ Ø8             BCS TYPE4C  (all X(4,C))
Ø26A  4A                LSR A  (bit 3 → carry)
Ø26B  BØ Ø4             BCS LEGIT  (all X(8))
Ø26D  C9 Ø8             CMP #$Ø8  (tests 8Ø)
Ø26F  FØ ØA             BEQ NOTLEG (8Ø is illegal)
Ø271  6Ø       LEGIT    RTS  (all XØ legals)
Ø272  4A       TYPE4C   LSR A  (bit 3 → carry)
Ø273  FØ Ø6             BEQ NOTLEG (Ø4, ØC illegals)
Ø275  9Ø Ø5             BCC TYPE4 (other X4)
Ø277  C9 Ø9             CMP #$Ø9  (tests 9C)
Ø279  DØ E1             BNE TYP4AC (residual XC)
Ø27B  ØØ       NOTLEG   BRK  (9C is illegal)
Ø27C  29 ØD    TYPE4    AND #$ØD  (tests 44, 64)
Ø27E  C9 Ø4             CMP #$Ø4  (must = Ø4)
Ø28Ø  DØ DA             BNE TYP4AC (residual X4)
Ø282  ØØ                BRK  (44, 64 illegals)
```

When OPLEGL was tested (on a KIM-1, with a simple program that caused each BRK to display the illegal opcode for a few seconds) all 104 illegals were correctly identified. Nearly half of the 67 program bytes are required by X(4,A,C). Minor restructuring could save a few bytes, but I have not bothered because other programmers may now feel challenged to create a subroutine that will be both more byte- and time-efficient.

I have noted with regret the common tendency to bury complex logic inside special-purpose main programs instead of coding it as subroutines. This seems desirable to me *only* when it is vital to attain the absolute minimum execution time. The saving of 4 bytes needed by a JSR and RTS is a trivial gain. Even when its originator cannot conceive that a logic block could ever be useful in any other context (and who can be *certain* of that?), subroutining may offer greater structural flexibility, intelligibility, and ease of debugging and modification. Especially in ROMS (unalterable, but with a wonderful "always-there" character) rich internal subroutining can greatly increase the power of a system; KIM-1 users have exercised great ingenuity in accessing much of the programming in the 2K ROM, a task made more difficult by the failure of its designers to anticipate this. Furthermore, a microprocessor may be incorporated in many diverse systems (especially true of the 8080 and 650X chips), so that main programs are very often system-dependent. To the extent that they use system-independent subroutines, their adaptation to systems other than the one for which they were developed is facilitated.

*Dear Dr. Warren,* *August 5, 1977*

*Enclosed is a one-page, one-paragraph addition to the MS I sent you a few days ago. It is an afterthought prompted by reading Stork's simulation program, in the issue of KILOBAUD I received after sending you my MS. Like Adam Osborne, I find instruction sets fascinating. They are where the real power resides. Although a primitive set, used by experts and provided with powerful auxiliaries, will outperform a superior design that lacks these enrichments, in the long run class will tell.*

*Sincerely,*
*H.T. Gordon*

Like everyone else's, most of my main programs are system-dependent and involve routine operations. Whatever elegance there is must reside in the subroutines, codable in countless ways. Separately publishing these makes them available to any program in any 650X system, and may also focus attention on some elements of software design in all systems (in this instance, the advantages of branched- vs. linear-sequence sift/sort operations). Opcode decoding can be useful in non-650X systems; e.g., a debugging program-execution-simulator by Lee Stork in the Sept. '77 KILOBAUD has an opcode-byte-count routine in 8080 assembly language, using a linear sequence of 14 bit-tests (6 ANI and 8 CPI) and 14 jump-on-conditions. It is likely that this decoding existed previously, hidden in the mass of 8080 software. The absence of relative-branch instructions in the 8080 set seems strange to users of later designs of microprocessors (although I suppose 8080/Z80 users would feel handicapped by their limited range!). Still, minicomputers (and their micro copies) do without them, and the creation of a status register and a flock of jump-on-condition instructions was one of many brilliant innovations by Intel designers in the evolution of the 8008/8080 chip. One wonders what heights the Z80 might have reached, had these same designers not felt constrained to maintain software-compatibility with the 8080. When one sees how willing users are to rewrite logic blocks, instead of hunting for them in older software, the compatibility argument looks very weak! Although BASIC interpreters are not cheap, many versions exist for the 8080 and even for the 650X.