Decoding Efficiency & Speed:

Pros & Cons of Table Look-Up

BY H. T. GORDON
College of Agricultural Sciences
University of California
Berkeley, CA 94720

Dear Jim,

Received: 78 Jan 15

While some of your readers may have become as weary as I am with my repeated attacks on the 650X legality-bytecount problem, this is in a way a programming guinea-pig for working-out optimization principles, relevant to all *problems. Even your non-650X readers may find something of value in the enclosed MS, as it touches on some universal problems: the chioce between decoding-logic or table-look-up programs, the extent to which subroutines should be designed to be non-destructive and purely-informative, and the uses of antibugging code. My ultimate goal is to develop scanning-debugging to a higher level. For this, the basic components of the debugger must be fast, efficient, and above al bug-free.*
Sincerely,
H. T. Gordon

The question of how to optimize a decoding subroutine, previously touched on in my notes in issues #17 and #19 of *DDJ* (1977) using decoding of the number of bytes required by 650X opcodes as an example, is re-opened by 2 new solutions: CNTBYT (written by Jim Butterfield and reproduced here with his permission) and BYTCNT (my own response to his challenge). Both optimize coding efficiency, and it is unlikely that a solution in fewer than 28 bytes exists. Both sort the 151 legal opcodes into the same 6 groups as my earlier subroutine BYTNUM. However, CNTBYT classes illegals of type X_0B as 3 byte codes while the others class them as 2 byte codes (which is how the 650X control unit executes them, but this is immaterial since all routines class illegals of type X2 as 2 byte codes although most of them are not executable).

The important differences are in structure and timing. CNTBYT uses an elegant and highly instructive linear sequence of bit-manipluations that cause each of 5 classes of opcodes in turn to become zero and exit, while the 6th class exits as non-zero. BYTCNT emulates BYTNUM by first sorting into subclasses X(0-7) and X(8-F) by a BIT test of bit 3, then shifts to destructive logic in each branch to attain higher code efficiency. Unlike BYTNUM, both new routines destroy the accumulator.

All routines have different path lengths (measured in clock cycles from entry to exit, including the 6 cycles for the main program JSR and 6 cycles for the RTS) for each of the 6 opcode classes. An index of relative speed (not weighted for difference in opcode usage frequency) is easily calculated by dividing Σ (N X T) by Σ N, where N is the number of legal opcodes in a class and T the path length in cycles for that class. BYTNUM has the fastest index, 30.6 cycles (range 28 to 37). BYTCNT is slower, 32.5 cycles (range 29 to 39). CNTBYT is slower still, 39.3 cycles (range 23 to 42). The delay is caused by its having to sort out 3 small classes (containing only 11 of the 151 opcodes) before it can work on the 3 large ones. The lesson to be drawn from this is that linear-sequence logic is *not inherently slower* (as I implied in my previous *DDJ* notes) and might even be *faster* if its first stages could sift out the majority of the items being sorted. However, the range of sorting times is necessarily wider. One additional point: to make the 2 new routines fully equivalent to BYTNUM, one would have to add a PHA to store and a PLA to restore the accumulator, at the cost of 2 more bytes and 7 more cycles. However, the main program might be able to restore the accumulator more efficiently.

One interesting point about all these "decoding" routines is that, although their branch instructions subdivide opcodes into 6 classes, they return with 3 byte-count classes *re-encoded* (as 01, 02, and 03) in the X register, so that a little more decoding logic is required to *use* this information. A second point, that I have emphasized before but that deserves reiteration, is that they all produce *additional* information as an incidental effect. E.g., CNTBYT and BYTCNT cause *only* 1 byte opcodes to set the Z flag, i.e., this type is *truly* decoded. It would be easy to recode BYTNUM to have the same effect, by omitting its inner RTS and relocating its TRICK sequence outside the subroutine. This would save 1 byte and very slightly reduce its speed. CNTBYT returns with the carry flag always equal to bit 7 of the opcode, and the sign flag always cleared. BYTCNT sets the carry flag equal to bit 4 for all 1 byte opcodes; it clears the carry for opcodes of type (0,1) (0,7) and sets it for all others. The BIT instruction in BYTCNT and BYTNUM sets the V flag equal to bit 6 for all opcodes; in BYTNUM it also causes all 1-byte opcodes to return with the N flag equal to bit 7.

I still prefer and recommend BYTNUM, although in my KIM system it takes 32 bytes (that would be 31 in the minor revision suggested above). In a well designed system, every one of the 256 possible bit patterns would be present in ROM (for use as numerical constants, BIT masks, etc.) and BYTNUM could be coded in 30 (or 29) bytes, a small price to pay for non-destructive logic and higher speed. Unfortunately (as I showed in a recent note in *Kim User Notes*) the KIM-ROM omits 81 of the possible patterns, including 39 of the 151 legal opcodes.

In the letter in which Jim Butterfield sent me his CNTBYT program, he also pointed out that a table-look-up operation would optimize timing. One way to do this is to move the opcode into an index register to get the content of a unique location in a 256 byte table. A content of 00 would identify an illegal (setting the Z flag), while contents of 01, 02, or 03 would give the byte-count of a legal. A 7 byte program insert (XINFOA, cf. listing) would do the work of *both* BYTNUM and OPLEGL at a cost of (7+M) main program bytes and (10+N) cycles. The values of M and N depend on whether the opcode needs to be restored in the accumulator (if not, both are zero) and on the kind of coding the main program is using to pick up a sequence of opcode bytes (most likely the LDA (INDIRECT), Y instruction with M = 2 and N = 5). Since the existing (not optimized) decoding logic version of OPLEGL needs 66 subroutine bytes and has a speed index of 37 cycles (range 24 to 63), the XINFOA insert would run more than 4 times faster than the OPLEGL + BYTNUM combination. To anyone enamored of speed (as I am) this is very attractive. An added superiority is that the main program can decide what to do with an illegal. If a program BREAK is desired, the branch offset ILLEG can be directed to the nearest 00 operand, otherwise to an illegality handling routine. One of several flaws in OPLEGL is that it forces a

BREAK for illegals. A well designed subroutine should return information and *not* make decisions.

Nothing comes free, and the price of the XINFOA speed is a very long table, with every opcode a *unique* class, vulnerable to bugs if located in RAM. Ideally an info table like TAB256 should be in ROM, but it uses only the 2 low-order bits and one hesitates to waste 75% of the bits in an expensive ROM. The solution would be to encode *other* information in the high-order bits, and AND them out to retrieve the legality/bytecount data. The dilemma is: *what* other information?

I briefly explored another approach, fully packing the infobit pairs of TAB256 into a 64 byte table. Retrieval needs a 36 byte program, too long for a main progarm insert, and with a speed index of 63 cycles when made into a subroutine. This is not good enough to be worth presenting here.

A more promising use of info tabling is my new legality-testing subroutine, HYLEGL, that has many advantages when compared with OPLEGL. It needs only 64 bytes, saves and restores the accumulator, and returns with the carry flag set by illegals and cleared by legals. Its speed index is 44 cycles (range 36 to 71). A revision of OPLEGL to make it fully equal to HYLEGL would be about 10 bytes longer and 2 cycles slower.

HYLEGL is a "hybrid" operation that retains the decoding logic of OPLEGL for the 128 "odd-number" opcodes because this is both fast and efficient. For the 128 "evens" it encodes legality as single infobits packed in a 16 byte table, INFTAB, a 1 bit meaning illegal and a 0 bit legal. A table-addressing index from 0 to F is generated by ANDing bits 4321 of the opcode, bit 0 (always 0 for evens) having been right-shifted out. The 16 infobytes correspond to the original opcode sequence: $X_0$0, $X_0$2,. . . .X_0E, $X_1$0. . . .X_1E. Since 7 of the infobytes contain no 1 bits, legality is proved by the setting of the Z flag at the load into the accumulator, so no retrieval is needed. For the remaining 9, the correct infobit is moved into the carry flag by a sequence of from 1 to 8 ASLs, controlled by analyzing the status of bits 765 of the original opcode (stored in the temporary zero-page location COPY).

Since INFTAB contains 128 unique classes, it is as vulnerable to bugs as any other table in RAM. Since it is short, it is much easier to check and also more likely to get itself fitted into a ROM. Decoding logic in RAM is of course not immune to bugs, but when a large class is involved the bug will show up very quickly. In a unique class, rarely use-tested (such as the illegals individually weeded out by OPLEGL) a bit-error bug may lurk for a long time before inflicting damage. Worse, the damage may be neither suspected nor detected; the output will look right but be wrong, and may even create a *new* hidden bug in other programs.

HYLEGL highlights some "gray areas" of subroutine design, where programming philosophies differ. The X register is used in its table look-up segment, but the pre-call X content is saved and restored at a cost of 4 bytes and 6 cycles. Its operations destroy the pre-call content of the accumulator, but this is saved and restored at a cost of 3 bytes and 7 cycles. It can be argued that only the main program really knows what needs to be saved and how and where to save it and when to restore it. By omitting the saves, HYLEGL would need only 57 bytes and 34 cycles, but greater precaution would be needed in the writing of the main program (perhaps not an evil). Should subroutines try to leave no traces of their work, other than the intended ones? Despite its saves, HYLEGL *does* leave traces, in the zero-page locations STORX and COPY and as incidental effects on the N, Z, and V flags. However, the PLP and PHP instructions of the 650X make status saving especially easy for the main program, and low zero-page locations are commonly viewed as temporary scratchpad area. My current view is that subroutines should be written to save on-chip registers, since it is easy for a user to remove any save instructions he does not want.

In order to make the high speed of the XINFOA/TAB256 approach immediately available, I am adding to the package subroutine MKTABL, that uses HYLEGL and BYTNUM to construct a *temporary* TAB256 in a RAM page. MKTABL is long (52 bytes) because it has an unusually high content of antibugging code and also serves as a testing/debugging program for the legality/bytecount subroutines. A large table in RAM is not only insecure but hazardous to generate. The data generating routines may be yielding some incorrect values. The write operation may be overwriting a RAM page other than the intended one, or trying to write in nonexistent RAM (that in systems not fully address-decoded may prove to be existent RAM with valuable stuff in it) or in write-protected RAM or in ROM.

Before calling MKTABL, a main program is expected to write into location BASEHI the number of the RAM page into which the table will be written. MKTABL compares this to the unique page number in its own coding; only if they are the same will table writing be permitted. It also sets the low base address for the table to 00, to ensure that the table will lie within the bounds of the one RAM page. This is part of the initial zeroing of 5 adjacent zero-page locations, that will contain useful information when MKTABL exits. The lowest one, COPY, is used by HYLEGL and will contain $FF at a correct exit; each run of HYLEGL stores the opcode it has worked on, unaltered except for the 72 "evens" that require infobit retrieval (that causes one ASL of COPY).

The next higher location (LEGCNT) contains $97 at a correct exit, since this counter is incremented before each call to BYTNUM, of which there will be 151 if HYLEGL is working properly. The next location (CHKLO) is a 1 byte checksum that contains $41 at a correct exit. The next location (TABGEN) indirectly counts the number of table-writes, containing either 00 (if no table was written) or $FF. This is a bit trickier than a simple count. After each write, the current value of the opcode is moved from the Y index into the accumulator, destroying the infobyte content so that the subsequent reloading of the infobyte from the table location is a test that the correct value was indeed stored there. The actual content of TABGEN is the current opcode value, from Y, that if all is well is also the count of writes.

Finally, the content of BASELO should always be 00, and BASEHI should contain the number of the unique permissible RAM page if the main program commanded a table-write, and some other number if not. A simple way to implement a no-table command would be an LDY #$06 followed by a JSR SETEX to bypass the first MKTABL instruction, since SETEX would zero BASEHI.

The terminal instructions of MKTABL control the 2 flags that will tell the main program what has happened. The BIT sets the V flag if a table has been written (TABGEN = $FF), and clears it if not. The crucial instructions compare the checksum in CHKLO with the correct value $41, setting the Z flag if all is well, clearing it if not. A great variety of errors, either in MKTABL itself or in the subroutines it calls, can cause a wrong checksum. Much information on the kind of error can be derived from the content of the 6 zero-page locations, of the accumulator (should be $41) and of the X and Y registers (both should be 00). If a table was written, very detailed information on operation will be found in the infobyte sequence.

The antibugging logic in MKTABL, although far from a total bugproofing, is clearly costly in memory and time. Programs that aim to optimize timing, such as the XINFOA/

TAB256 combination, can tolerate little or none. (An example of very light antibugging would be addition of an AND #$03, at a cost of 2 bytes and 2 cycles, to erase the 6 high-order bits of the infobyte.) The major insurace against error for such programs is a thorough testing by an antibug test program, both before and after each use. In service programs resident in RAM, like MKTABL, error-proofing can be given priority. It may be desirable to trade off more memory to attain higher speed, but one should never trade off reliability! On this subject, the chapters on antibugging, testing, and debugging in E. Yourdon's *Techniques of Program Structure and Design* (Prentice-Hall, 1975) make interesting reading.

To sum up, tables can indeed play useful roles beyond their classic one of defining arbitrary relations and rules (such as the metalogical ASCII decision that $41 represents the letter A). These other roles have been obscured by the fact that it is simpler to implement a *de novo* calculation of a math function than a look-up in a gigantic table. With tables of more modest size, table look-up can (as Jim Butterfield suggested) maximize speed, at an acceptable (though relatively high) memory cost. When decoding logic proves to be very intricate (many small or unique classes) the use of tables may also be more code-efficient. Furthermore, programs may be easier to write because similar retrieval logic may be applicable to a variety of tables. There is probably some trade-off point (more that 35 bytes of decoding logic, more than 10 classes decoded?) where the entabling of pre-decoded information becomes an optimal solution of the dimensions of the table can be kept within reasonable bounds (no more than 32 bytes?).

(coding for CNTBYT and BYTCNT)

```
Ø23Ø  A2 Ø1     CNTBYT LDX #$Ø1 (initialize X)
   2  49 2Ø            EOR #$2Ø (flip bit 5)
   4  FØ 13            BEQ THREE (only $2Ø)
   6  29 9F            AND #$9F (bits 5,6 out)
   8  FØ 11            BEQ ONE  (only (Ø,4,6)Ø)
   A  ØA               ASL A  (shift bit 7 out)
   B  49 12            EOR #$12 (flip bits 3,Ø)*
   D  FØ ØB            BEQ TWO  (only X09)
   F  29 1A            AND #$1A (bits 4,1 out)*
Ø241  49 Ø2            EOR #$Ø2 (flip bit Ø)*
   3  FØ Ø6            BEQ ONE  (X(8,A))
   5  29 1Ø            AND #$1Ø (all but flipped
                                 bit 3 out)*
   7  DØ Ø1            BNE TWO (resid. X(Ø-7))
   9  E8       THREE   INX      (resid. X(9-F))
   A  E8        TWO    INX
Ø24B  6Ø        ONE    RTS

 * comment refers to original opcode bits
   before left-shift at Ø23A

Ø23Ø  A2 Ø1     BYTCNT LDX #$Ø1
   2  2C 36 Ø2         BIT BYTCNT+6 (test bit 3)
   5  DØ Ø8            BNE HALFOP (all X(8-F))
   7  C9 2Ø            CMP #$2Ø
   9  FØ ØE            BEQ THREE (only $2Ø)
   B  29 9F            AND #$9F  (bits 5,6 out)
   D  DØ ØB            BNE TWO (all except
                                (Ø,4,6)Ø)
   F  29 15    HALFOP  AND #$15 (retains only
                                 bits Ø,2,4)
Ø241  C9 Ø1            CMP #$Ø1
   3  FØ Ø5            BEQ TWO  (X0(9,B))
   5  29 Ø5            AND #$Ø5 (bit 4 out)
   7  FØ Ø2            BEQ ONE  (X(8,A) and
                                (Ø,4,6)Ø)
   9  E8         THREE INX  (resid. X(9-F))
   A  E8          TWO   INX
Ø24B  6Ø          ONE   RTS
```

```
              XINFOA TAX
                     LDA TAB256, X
                     BEQ ILLEG
                     TAX

Ø26E  48       HYLEGL PHA (save opcode in stack)
   F  85 1Ø           STA COPY (store work-copy)
Ø271  4A              LSR A  (bit Ø to carry)
   2  9Ø ØA           BCC TABLOP (do evens/table)
   4  4A              LSR A  (bit 1 to carry)
   5  BØ Ø5           BCS ILLEX (all X(3,7,B,F))
   7  C9 22           CMP #$22  (LSRed $89)
   9  FØ Ø1           BEQ ILLEX  ($89 illegal)
   B  18              CLC  (legals clear carry)
   C  68        ILLEX PLA  (restore opcode in A)
   D  6Ø              RTS
   E  86 ØF    TABLOP STX STORX (save X register)
Ø28Ø  29 ØF           AND #$ØF (bits 7,6,5 out)
   2  AA              TAX  (4321 to X index)
   3  BD 9E Ø2        LDA INFTAB,X (load infobyte)
   6  FØ 12           BEQ EXIT (all Ø bits, legal)
   8  Ø6 1Ø           ASL COPY (bit 7 to carry)
   A  BØ Ø4           BCS STEPA (bit 7 = 1)
   C  ØA              ASL A (leftshift infobyte)
   D  ØA              ASL A
   E  ØA              ASL A
   F  ØA              ASL A
Ø29Ø  24 1Ø     STEPA BIT COPY (bits 6,5 to N,V)
   2  3Ø Ø2           BMI STEPB (bit 6 = 1)
   4  ØA              ASL A
   5  ØA              ASL A
   6  7Ø Ø1     STEPB BVS STEPC (bit 5 = 1)
   8  ØA              ASL A
   9  ØA        STEPC ASL A  (infobit to carry)
   A  A6 ØF     EXIT  LDX STORX (restore X)
   C  68              PLA  (restore opcode)
Ø29D  6Ø              RTS
```

INFTAB table for even opcodes

```
Ø29E  1Ø DF ØD ØØ ØØ ØØ Ø1 ØØ
Ø2A6  ØØ FF CF ØØ ØØ CF DF 1Ø
```

(coding for MKTABL)

```
Ø2AF  AØ Ø5     MKTABL LDY #$Ø5 (zero 5 loci)
Ø2B1  A2 ØØ      SETEX LDX #$ØØ (notable entry)
   3  96 ØF     ZEROIT STX COPY-1, Y
   5  88               DEY
   6  DØ FB           BNE ZEROIT
   8  D8               CLD  (clear decimal mode)
   9  98        OPBYTE TYA  (opcode from Y to A)
   A  2Ø 6E Ø2         JSR HYLEGL (test legality)
   D  A2 ØØ            LDX #$ØØ  (rezeroes X)
   F  BØ Ø5            BCS MOVXA (illegal bypass)
Ø2C1  E6 11            INC LEGCNT (count legals)
   3  2Ø 1Ø Ø2         JSR BYTNUM (bytecount to X)
   6  8A        MOVXA  TXA (bytecount from X to A)
   7  A2 XX            LDX #$XX (RAM table page #)
   9  E4 15            CPX BASEHI (compare page #
                                   set by program)
   B  DØ Ø7            BNE NOTABL (nowrite if ≠)
   D  91 14            STA (BASELO),Y  (write)
   F  98               TYA  (opcode from Y to A)
Ø2DØ  85 13            STA TABGEN (count writes)
   2  B1 14            LDA (BASELO),Y (read table)
   4  18        NOTABL CLC  (clear carry for add)
   5  65 12            ADC CHKLO (bytecount sum)
   7  85 12            STA CHKLO (store checksum)
   9  C8               INY   (next opcode)
   A  DØ DD            BNE OPBYTE
   C  24 13     EXIT   BIT TABGEN (sets V if table)
   E  A9 41            LDA #$41 (valid checksum)
Ø2EØ  C5 12            CMP CHKLO (compare actual)
Ø2E2  6Ø               RTS
```