[6.57] Sum binary '1' digits in an integer

Some applications need to find the number of '1' digits in a binary integer. For example, this sum is needed to calculate a parity bit. Another example would be a game or a simulation in which positions are stored as '1' digits in an integer, and you need to find the total number of pieces in all positions. The following TI Basic function will find the number of 1's in the input argument *n*.

```
sum1s(n)
Func
©(n) sum of binary 1's in n, n<2^32
©Must use Exact or Auto mode!
©26aprilØ2/dburkett@infinet.com

local t,k,s

{Ø,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4}→t

Ø→s
for k,1,8
 s+t[(n and ØhF)+1]→s
 shift(n,⁻4)→n
endfor
return s

EndFunc
```

For example: `sum1s(Ø)` returns 0 `sum1s(ØhF)` returns 4
`sum1s(7)` returns 3 `sum1s(2^32-1)` returns 32

The last example results in a *Warning: Operation requires and returns 32-bit value*, but the correct result is returned. 32-bit integers are interpreted as 2's compliment signed integers, so the decimal range for valid input arguments is -2,147,483,648 to 2,147,483,647. This is 0h0 to 0hFFFF. (Recall that the TI-89/TI-92 Plus use *0b* to prefix binary integers, and *0h* to prefix base-16 integers).

sum1s() uses a table lookup (in the list *t*) to find the number of 1's in each 4-bit nibble of the input integer *n*. Each pass through the loop processes one nibble. The nibble is extracted by and-ing the current value of *n* with 0hF. I add 1 to the nibble value since list indices start at 1, not zero. After summing the correct list element, the input argument is shifted right four bits, so I can use the same bit mask of 0hF to extract the next nibble. The elements of list *t* are the number of 1s in all possible nibbles, in sequential order. For example, the nibble 0b0111 is decimal 7 which accesses the eighth element of t, which is 3.

There are many other ways to accomplish this task. A comprehensive survey of seven methods is the article *Quibbles and Bits* by Mike Morton (Computer Language magazine, December 1990). I chose the table lookup method because it has a simple TI Basic implementation. The number of loop iterations can be reduced by increasing the number of bits processed, but this increases the table size. We could process eight bits at a time in four loop iterations, but the table would have 256 entries. The method in *sum1s()* seems to be a good tradeoff between table size and loop iterations.

sum1s() executes in about 0.3 seconds/call. You could speed up *sum1s()* about 8% by making *t* a global variable and initializing it before running *sum1s()*. An even faster version would be coded in C. Unfortunately, a limitation in AMS 2.05 prevents this simple implementation, which would eliminate the *For* loop overhead:

```
sum(seq(t[(shift(n,-k*4) and ØhF)+1],k,Ø,7))
```