An Explanation of the Double-Dabble Bin-BCD Conversion Algorithm

by C.B. Falconer. 2004-04-16
mailto:cbfalconer@worldnet.att.net

The algorithm starts from this description, shifting left, and converting an 8 bit value to BCD. If a prospective BCD digit is five or larger, add three before shifting left.

HUNDREDS		TENS		UNITS		BINARY	
0000		0000		0000		11111111	Start
0000		0000		0001		11111110	Shift 1
0000		0000		0011		11111100	Shift 2
0000		0000		0111		11111000	Shift 3
0000		0000		1010		11110000	ADD-3 to UNITS
0000		0001		0101		11110000	Shift 4
0000		0001		1000		11110000	ADD-3 to UNITS
0000		0011		0001		11100000	Shift 5
0000		0110		0011		11000000	Shift 6
0000		1001		0011		11000000	ADD-3 to TENS
0001		0010		0111		10000000	Shift 7
0001		0010		1010		10000000	ADD-3 to UNITS
0010	<--	0101	<--	0101	<--	00000000	Shift 8

and the result is the BCD encoded value 255.

Bear in mind that the weights of BCD digit bits are 8,4,2,1 multiplied by the decimal position, i.e. 1, 10, 100, etc.

The rationale for the ADD-3 rule is that whenever the shifted value is 10 or more the weight of that shifted out bit has been reduced to 10 from 16. To compensate, we add 1/2 of that 6, or 3, before shifting. We detect the 10 value after shifting by the fact that the value before shifting is 5 or greater. Notice that the rule ensures that the various digits can never hold any non-BCD values.

The shifting is the double part of the algorithm. The ADD-3 is the dabble part. It might be better named dabble-double. Alas, history has decreed otherwise.

Dividing by 10 and saving the remainder

Next, we modify to extract the lsd (least sig. digit) and divide by 10, by adding a shift connection from the units high order bit to the binary low order bit. We also mark the highest order converted bit by x (for 0) and by X (for 1).

```
HUNDREDS     TENS      UNITS        BINARY
  0000       0000       x000       11111111   Start
  0000       000x       0001       1111111x   Shift 1
  0000       00x0       0011       111111x0   Shift 2
  0000       0x00       0111       11111x00   Shift 3
  0000       0x00       1010       11111x00   ADD-3 to UNITS
  0000       x001       0101       1111x001   Shift 4
  0000       x001       1000       1111x001   ADD-3 to UNITS
  000x       0011       0001       111x0011   Shift 5
  00x0       0110       0011       11x00110   Shift 6
  00x0       1001       0011       11x00110   ADD-3 to TENS
  0x01       0010       0111       1x001100   Shift 7
  0x01       0010       1010       1x001100   ADD-3 to UNITS
  x010       0101      -0101       x0011001   Shift 8
                       |      <--          ^
                       v___________________|
```

The arrows show the bit shifting path. The binary is now 0x19 in hex, with a value of 25 decimal. The UNITS digit is 5, which is the remainder after division of 255 by 10. We haven't really used the HUNDREDS or TENS register.

Varying the size of the binary register

Now, try it with a 10 bit number, and move the UNITS register to the right of the BINARY register. Since we no longer have a TENS register we can't do the ADD-3 to it, but we leave it in the annotations for future use. The input binary value is 1023.

```
 BINARY      UNITS
1111111111   x000   Start
111111111x   0001   Shift 1
11111111x0   0011   Shift 2
1111111x00   0111   Shift 3
1111111x00   1010   ADD-3 to UNITS
111111x001   0101   Shift 4
111111x001   1000   ADD-3 to UNITS
11111x0011   0001   Shift 5
1111x00110   0011   Shift 6
1111x00110   0011   ADD-3 to TENS (dummy)
111x001100   0111   Shift 7
111x001100   1010   ADD-3 to UNITS
11x0011001   0101   Shift 8
11x0011001   1000   ADD-3 to UNITS
1x00110011   0001   Shift 9
x001100110   0011   Shift 10
|         <--   ^
v________________|
```

The binary value has become 0x066 or 102 decimal. The units digit is now 3, so the system has divided by 10 and extracted the remainder. The count of shifts is dictated by the length of the binary register, or by when the 'x' marker reaches the left hand bit of the binary register. This is the point at which all the original binary bits have been 'used'.

This is sufficient to do BCD conversions, extracting the least significant digit in N operations from an N-bit binary. By repeating it for each digit we end up with roughly N/4 * N operations for the complete conversion, or O(N*N).

In this form the algorithm is useful for division by 10.

Modifications to do binary-bcd conversions in place

The next version is identical, but we have broken the binary register up into 4 bit groups.

```
   ----BINARY----
   THOU HUND TENS   UNITS
     11 1111 1111   x000   Start
     11 1111 111x   0001   Shift 1
     11 1111 11x0   0011   Shift 2
     11 1111 1x00   0111   Shift 3
     11 1111 1x00   1010   ADD-3 to UNITS
     11 1111 x001   0101   Shift 4
     11 1111 x001   1000   ADD-3 to UNITS
     11 111x 0011   0001   Shift 5
     11 11x0 0110   0011   Shift 6
     11 11x0 0110   0011   ADD-3 to TENS (dummy)
     11 1x00 1100   0111   Shift 7
     11 1x00 1100   1010   ADD-3 to UNITS
     11 x001 1001   0101   Shift 8
     11 x001 1001   1000   ADD-3 to UNITS
     1x 0011 0011   0001   Shift 9
     x0 0110 0110   0011   Shift 10
     |           <--   ^
     v___________________|
```

Now alter the ADD-3 rule to say - Add 3 whenever the digit value is 5 or more, AND the digit is entirely to the right of the x marker, inclusive. Notice that the ADD-3 to TENS suddenly is back in effect. So are two new dabbles, marked by <-- below.

```
      ----BINARY----
INDEX THOU HUND TENS   UNITS
  0     11 1111 1111   x000   Start
  1     11 1111 111x   0001   Shift 1
```

```
  2     11 1111 11x0   0011   Shift 2
  3     11 1111 1x00   0111   Shift 3
  4     11 1111 1x00   1010   ADD-3 to UNITS
  4     11 1111 x001   0101   Shift 4
  5     11 1111 x001   1000   ADD-3 to UNITS
  5     11 111x 0011   0001   Shift 5
  6     11 11x0 0110   0011   Shift 6
  7     11 11x0 1001   0011   ADD-3 to TENS (live)
  7     11 1x01 0010   0111   Shift 7
  8     11 1x01 0010   1010   ADD-3 to UNITS
  8     11 x010 0101   0101   Shift 8
  9     11 x010 0101   1000   ADD-3 to UNITS
  9     11 x010 1000   1000   ADD-3 to TENS  <--
  9     1x 0101 0001   0001   Shift 9
 10     1x 1000 0001   0001   ADD-3 to HUND  <--
 10     x1 0000 0010   0011   Shift 10
        |           <--   ^
        v__________________|
```

and we come out with the BCD conversion to 1023. Notice that all the "ADD-3"s are associated with the following shift. The above has an INDEX column added to correspond. Now we can see when to energize the ADD-3s for each BCD column, in terms of the index value. Units are available immediately, TENS after 4 or more, HUND after 8 or more, and we never can dabble the THOU column. Looks like the rule there should be 12 or more. This bears a startling resemblance to (index DIV 4) indicating a BCD digit.

Notice that the conversion has been done in place, with only one 4 bit BCD digit of auxiliary storage. At some size of the binary this will not be enough, but the solution is more zero bits on the left of the binary value. This increases the max index value on conversion.

Since the various ADD-3s are limited to a single BCD digit, and they do not interact, they can all be done in parallel, and in fact in parallel with the following shift operation. This means that converting an N bit value to BCD requires no more than N operations, provided that N bit value has sufficient leading zero bits. This is now O(N), which is a large improvement for large binary numbers.

A purely software attack will probably not be able to do all the ADD-3s in parallel, but in hardware this is simply a set of gates in the shift path.

We can also, if convenient, think of the added units digit as being supplied by an initial 4 bit left rotary shift of the binary register.

Problems with software implementation

We can normally implement left shifts fairly easily in software, with some complications to implement carrying bits across word or octet boundaries. This is probably simplified as much as possible in C by using unsigned chars, and limiting their range to 0..255. Combined with a carry in and a carry out variable, unlimited sized binary shifts can be handled.

A practical problem arises with the dabble portion. We could mask off each four bit section of an octet, compare the value to 5, and modify accordingly. This is fairly computationally intensive.

Another possibility is to treat the octet as a whole, and pre-prepare a translation table with 256 entries. This should be able to handle the whole octet in one operation. However two table will be needed, depending on whether or not the left hand portion of the octet (see the x bit above) is to be modified.

The byte sex of the binary values has to be settled, and it will normally control the order of the BCD values that result. It will usually be convenient for the most significant BCD digit to appear in the lowest address, which in turn implies that the most significant binary bit appears in the lowest address. The reverse is, of course, perfectly feasible.

Other considerations

A little thought will show that the above operations are all reversible. So they can be used for BCD to binary conversion, or to multiply a binary value by 10 and add in a digit.

Even the value 10 is not essential. It can easily be replaced by any even number. Greater care is needed for odd values.