Conundrums, Puzzles, and Posers

Generating an 8-Bit Gray Code

When moving between states in a standard binary sequence, multiple bits may change from 0 to 1 or vice versa; for example, three bits change value when moving from %0011 to %0100 (where the % character is used to indicate a binary number). In the physical world, there is no way to ensure that all the bits will transition at exactly the same time, so a system may actually pass through a sequence of intermediate states. For example, an intended state change of %0011 to %0100 could actually result in the sequence %0011 to %0111 to %0100.

One way to avoid this problem is to use a Gray code, in which only a single bit changes when moving between states as illustrated in Figure 1.

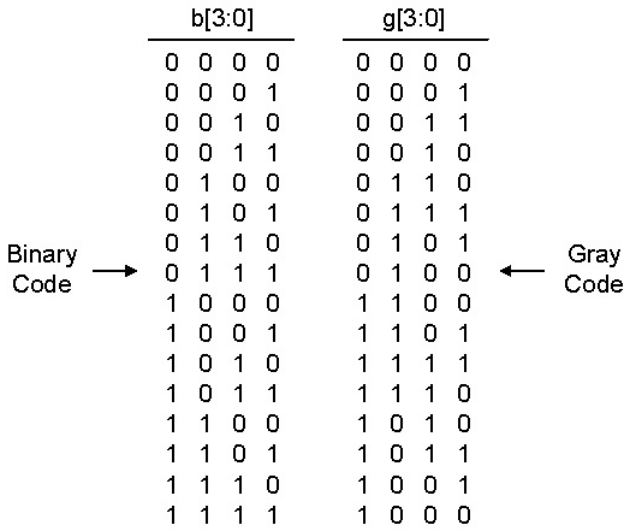

Figure 1. 4-bit binary code versus 4-bit Gray code.

Gray codes are of use for a variety of applications, such as encoding the angle of a mechanical shaft (possibly in an industrial control application), where a disc attached to the shaft is patterned with areas of conducting material as illustrated in Figure 2.

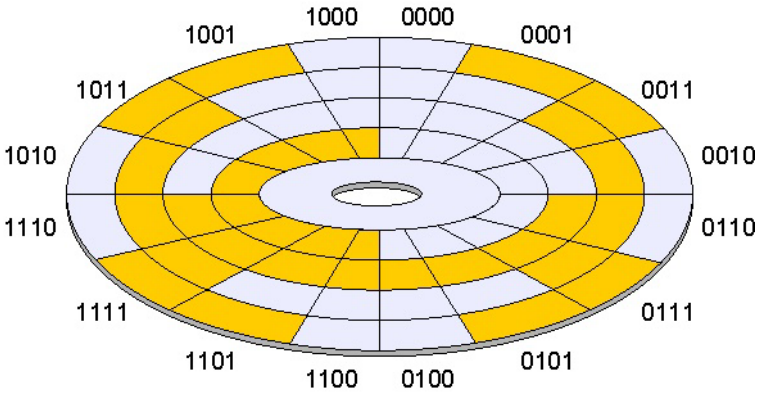

Figure 2. 4-bit Gray code used for shaft-angle encoding.

The conducting (golden) areas are arranged in concentric circles, where each circle represents a binary digit (the inner circle represents bit 3 and the outer circle represents bit 0 in this

example). A set of electrical contacts, one for each of the circles, is used to detect the logic values represented by the presence or absence of the conducting areas. A digital controller can use this information to determine the angle of the shaft. The precision of the measurement depends on the number of bits (circles). A 4-bit value representing 16 unique states allows the shaft's angle to be resolved to 22.5 degrees, while a 10-bit value representing 1,024 unique states allows the shaft's angle to be resolved to 0.35 degrees. Using a gray code sequence to define the conducting and non-conducting areas ensures that no intermediate values are generated as the shaft rotates.

Commencing with a state of all zeros, a gray code can be generated by always changing the least significant bit that results in a new state. An alternative method which may be easier to remember and use is as follows:

1) Commence with the simplest gray code possible; that is, for a single bit.
2) Create a mirror image of the existing gray code below the original values.
3) Prefix the original values with 0s and the mirrored values with 1s.
4) Repeat steps (2) and (3) until the desired width is achieved.

An example of this mirroring process used to generate a 4-bit gray code is illustrated in Figure 3.

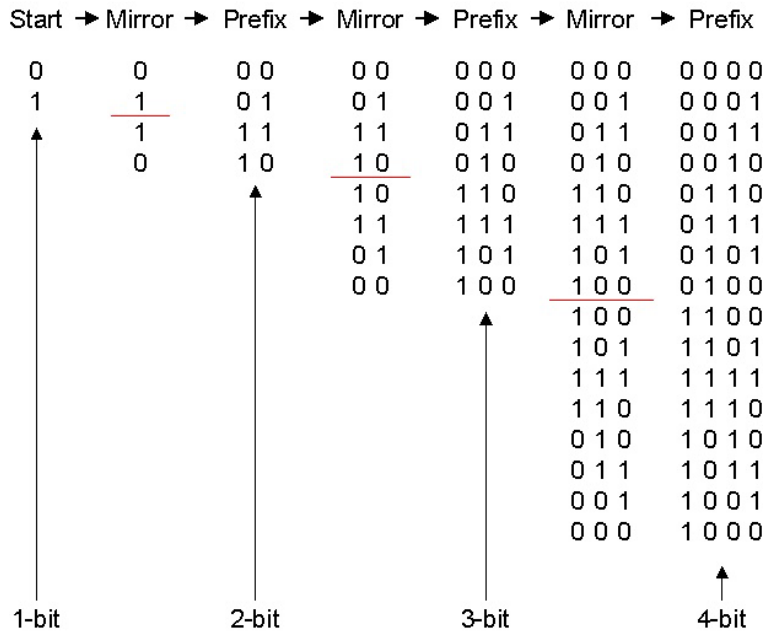

Figure 3. Using the mirroring technique to generate a 4-bit gray code.

It is often required to convert a binary sequence into a gray code or vice versa. Can you think of a way to perform the following task in the DIY Calculator's assembly code using the fewest number of opcodes and/or using the smallest number of bytes?

1) Load the accumulator with binary 00000000 ($00) then count up to binary 11111111 ($FF).
2) For each binary value generate a corresponding Gray code value.
3) When you get to 11111111 ($FF) return to 00000000 ($FF) and start again.

One possible solution is described on the next page … can you come up with something better?

Don't look at the following solution until you've tried working this out for yourself, otherwise you'll spoil the fun!

Assembly code (8 opcodes, 19 bytes of memory)

```
            .ORG  $4000     # Set origin
            LDA  %00000000 # Initial binary code

LOOP:       STA  [TEMP]    # Store binary code
            SHR            # Shift right 1 bit
            AND  %01111111 # See notes

            XOR  [TEMP]    # This is the gray code!

            LDA  [TEMP]    # Restore binary value
            INCA           # Increment binary value
            JMP  [LOOP]    # Do it all again

TEMP:       .BYTE          # Temporary location
            .END
```

This implementation is really very simple. For each binary code, a corresponding Gray code can be generated by (a) storing a copy of the original binary code, (b) shifting the original binary code one bit to the right, and (c) XOR-ing the two values together.

The AND instruction is used to ensure that g[7] = b[7].

Note that, as an alternative, we could have reduced this to only 6 instructions by means of a look-up table approach, but this would have required a lot more memory [take a look at the "count the ones" puzzle in the *Conundrums, Puzzles, and Posers* section of the *Programs and Routines* page of the DIY Calculator website (www.diycalculator.com) for an example of a look-up-table solution].

If you want to learn more about Gray codes, including the way in which the above XOR-based solution was derived, check out my book *Bebop To The Boolean Boogie (An Unconventional Guide To Electronics)* Second Edition (ISBN: 0-7506-7543-8).

If you come up with an alternative technique, please send it to max@diycalculator.com and I'll add it in to this paper (crediting you, of course).

Note: The correct grammar and spelling for Gray codes is "Gray" (not "gray", "Grey", or "grey"). This is because Gray codes are named after Frank Gray, who patented their use for shaft encoders in 1953. You can find a really interesting paper on this at the following URL:

http://www.faqs.org/faqs/ai-faq/genetic/part6/section-1.html