Episode 2.10 – Gray Code Conversion and Applications

Welcome to the Geek Author series on Computer Organization and Design Fundamentals. I'm David Tarnoff, and in this series we are working our way through the topics of Computer Organization, Computer Architecture, Digital Design, and Embedded System Design. If you're interested in the inner workings of a computer, then you're in the right place. The only background you'll need for this series is an understanding of integer math, and if possible, a little experience with a programming language such as Java.

In our last episode, we made a distinction between the ordered sequence of binary integers and an ordered sequence of patterns of ones and zeros referred to as Gray code. Both binary numeral systems use all of the two to the n-th possible patterns of ones and zeros. In the case of unsigned binary integers, each bit position carries a weight corresponding to a different power of two thus giving it is numeric order. Gray code, however, is not a weighted binary code, rather it is a cyclic list of binary patterns ordered so that a single step forward or backward through the list results in only a single bit change in the pattern. It is suited for sequences, not for mathematical operations.

Frank Gray's original description of the code that was to be named after him can be found in his patent describing pulse code communication.[1] In the patent, he describes how a specific application of digital communications used digital values to control the deflection of an electron beam, and that errors in the binary position could cause the beam to wander. His thought was that Gray code would avoid spurious errors in the digital code by having only one bit change due to a change of unity in the integer value. Because of Gray code's inherent protection against reading incorrect values while the signal levels are in transition, it can also be useful when two digital systems using two different clock frequencies attempt to pass data between one another. The data being passed between devices can also benefit from error correction schemes based on Gray codes. Mechanical systems, both sensors and actuators, can benefit from the application of Gray code by minimizing wear and tear as they advance through the steps of a process. That said, let's see how we can generate the sequence.

As part of the last episode, we presented a method for coming up with the Gray code sequence for n bits by using the sequence for n-1 bits. We start with the 1-bit cyclic list 0 then 1 then 0 then 1 then 0 then 1 and so on. Note that as we cycle through the single bit list over and over again, each pattern differs by only one bit from the previous or subsequent patterns. This isn't particularly surprising as there's only one bit to be changed.

To generate the 2-bit sequence, we take the 1-bit sequence, 0 then 1, and follow it with the reverse of the one bit sequence, 1 then 0. To the first half of the sequence, we prepend a 0 to get 00 then 01, and to the last half of the sequence, the sequence that was reversed, we prepend a 1 to get 11 then 10. Listing them together gives us the 2-bit Gray code sequence 00, 01, 11, and 10. Like the 1-bit sequence, this sequence also allows for cyclic operation.

We can use this 2-bit sequence to get the 3-bit sequence. Follow the 2-bit sequence 00, 01, 11, and 10 with its reflection 10, 11, 01, and 00, then prepend the first half with 0 and the second half with 1. This gives us the full three-bit Gray code sequence: 000, 001, 011, 010, 110, 111, 101, and 100.

This becomes quite tedious when we try to create larger sequences. What we need is a method to map the integer position with the corresponding Gray code pattern. It turns out that there's an algorithm that allows us to do just that!

Let's start with the unsigned binary integer that represents the position. If we prepend a 0 to the most significant end of the unsigned binary value, we will have an n+1 bit number, but more importantly, we will have n boundaries between the n+1 bits. From these n overlapping pairs of bits, we will create our n bit Gray code pattern corresponding to this unsigned binary integer. For each overlapping pair, place a 0 if the adjacent bits are the same and a 1 if the adjacent bits are different. The resulting pattern is the Gray code assigned to the position identified by the unsigned binary integer.

Okay, let's try an example. What is the eighth element in the 4-bit Gray code sequence? First, converting 8 to a 4-bit unsigned binary value gives us 1000. Prepending a zero to the beginning of this value gives us 01000. We now use the overlapping pairs of bits from this pattern to determine our Gray code. The first or most significant bit is 0, and the second bit is 1. These are different, so the first bit of our Gray code pattern is 1. The second and third bits, 1 and 0, are also different, so the second bit of our Gray code pattern is also 1. The third and fourth bits are both zero. Since they are the same, the third bit of our Gray code pattern is 0. And lastly, the fourth and fifth bits are the same, so the fourth bit of our Gray code pattern is 0. This means that the eighth element in a 4-bit Gray code sequence is 1100.

This is an excellent opportunity to introduce a concept in computing called logic gates. In mathematics, Boolean algebra allows us to generate binary results based on the true or false values of one or more binary signals. In computing, we can use Boolean algebra to perform bit-level operations like the one we just used to convert the unsigned integer to the binary pattern of the Gray code. There are a handful of binary operators used in Boolean algebra, but the one of interest here is called the exclusive-OR, sometimes referred to as XOR. When the exclusive-OR operation is applied to a pair of bits, the resulting output is zero if the bits are the same and a one if the bits are different. Our software has access to these operations, and as a result, we can write code to generate Gray code. Using a programming language such as Java or C, the exclusive-OR operator is the caret, or ^.

So to duplicate our algorithm in software, begin by making a copy of the unsigned integer. Next, shift all of the bits in the copy one position to the right. If you recall from Episode 2.2, Unsigned Binary Conversion, this can be done using an operator called a right shift, which is represented with two greater than symbols next to each other. In that episode, we used it to perform fast division by a power of two. Here, however, we are going to use it to move each bit of the copy one bit right so that it lines up with the neighboring bit to the right of the original unsigned value.

If we place the shifted binary copy directly below the original binary value, we see that the bit positions line up so that copies of the neighboring bits appear in each column. If we apply the exclusive-OR operation in a column by column or bit position by bit position fashion, the result will be the correct Gray code pattern for that integer. In the vernacular of programming, this last operation is called a bit-wise exclusive-OR. In a programming language such as Java or C, the code would look like:

```
grayCode = position ^ (position >> 1)
```

So what about going in the other direction? How can we get the unsigned binary integer representing the position from its Gray code? Although the code to convert from Gray code to the unsigned integer position is a bit more involved, the algorithm itself is simple, and once again, takes advantage of the exclusive-OR operation. Start by copying the most significant bit of the Gray code as the most significant

bit of the binary code. The next bit of the binary code equals the exclusive-OR of the previous bit of the binary code with the next bit position of the Gray code. Let's try that again by trying to convert the 4-bit Gray code pattern 1100 back to its position number, which should be eight.

The most significant bit of the binary code is the same as the most significant bit of the Gray code pattern, in other words, 1. We exclusive-OR this 1 with the next bit of the Gray code, which is 1, to get the next bit of the binary code. One exclusive-ORed with 1 is 0. We exclusive-OR this 0 with the next bit of the Gray code, which is 0, to get 0 as the next bit of the binary code. Lastly, we exclusive-OR this 0 with the last bit of the Gray code, which is 0, to get 0 as the last bit of the binary code. This gives us 1000 as our unsigned binary integer position of the Gray code pattern 1100.

That brings us to the end of another episode of the Geek Author series on Computer Organization and Design Fundamentals. In our next episode, we're going to start doing some math with these binary numbers. No worries though – for now we'll stick to addition and subtraction.

For transcripts, links, or other podcast notes, please check us out at intermation.com where you will also find links to our Instagram, Twitter, Facebook, and Pinterest pages. Until then remember that while the scope of what makes a computer is immense, it's all just ones and zeros.

References:

1 – Gray, Frank (1953-03-17). "Pulse code communication" (PDF). U.S. patent no. 2,632,058
https://edisciplinas.usp.br/pluginfile.php/4331530/mod_resource/content/1/US2632058.pdf