The (200, 200, 200), which is a light gray, can be generated using black ink. The remaining (50, 0, 0) can be generated using a small amount of cyan, and using no magenta or yellow ink at all, thus saving precious color ink. A CMY color space extended with black is known as a ***CMYK color space*** (the "K" comes from the last letter in the word "black." "K" is used instead of "B" to avoid confusion with the "B" from "blue").

An RGB to CMYK converter can thus be described as:

```
K = Minimum (C, M, Y)
C2 = C – K
M2 = M – K
Y2 = Y – K
```

where C, M, and Y are defined as earlier. We thus create the circuit in Figure 4.68 for converting an RGB color space to a CMYK color space. We've used the *RGBtoCMY* component from Figure 4.67. We've also used two instances of the *MIN* component that we created in Example 4.12 to compute the minimum of two numbers; using two such components computes the minimum of three numbers. Finally, we use three more subtractors to remove the K value from the C, M, and Y values. In a real printer, the imperfections of ink and paper require even more adjustments. A more realistic color space converter multiplies the R, G, and B values by a series of constants, which can be described using matrices:

```
|C|   |m00 m01 m02 |   |R|
|M|= |m10 m11 m12 |* |G|
|Y|   |m20 m21 m22 |   |B|
```

Further discussion of such a matrix-based converter is beyond the scope of this example. ◀

Representing Negative Numbers: Two's Complement

The subtractor design in the previous section assumed we only dealt with positive input numbers and positive results. But in many systems, we may obtain results that are negative, and in fact, our input values may even be negative numbers. We thus need a way to represent negative numbers using bits.

One obvious but not very effective representation is known as ***signed-magnitude***. In this representation, the highest-order bit is used only to represent the number's sign, with `0` meaning positive and `1` meaning negative. The remaining low-order bits represent the magnitude of the number. In this representation, and using 4-bit numbers, `0111` would represent +7, while `1111` would represent –7. Thus, four bits could represent –7 to 7. (Notice, by the way, that both `0000` and `1000` would represent 0, the former representing `0`, the latter `-0`.) Signed-magnitude is easy for humans to understand, but doesn't lend itself easily to the design of simple arithmetic components like adders and subtractors. For example, if an adder's inputs use signed-magnitude representation, the adder would have to look at the highest-order bit, and then internally perform either an addition or a subtraction, using different circuits for each.

Instead, the most common method of representing negative numbers and performing subtraction in a digital system actually uses a trick that allows us to *use an adder to perform subtraction*. Using an adder to perform subtraction would enable us to keep our simple adder, and to use the same component for both addition and subtraction.

The key to performing subtraction using addition lies in what are known as *complements*. We'll first introduce complements in the base ten number system just so you can

We are introducing ten's complement just for intuition purposes—we'll actually be using two's complement.

familiarize yourself with the concept, but bear in mind that the intention is to use complements in base two, not base ten.

Consider subtraction involving two single-digit base ten numbers, say 7 – 4. The result should be 3. Let's define the ***complement*** of a single-digit base ten number A as *the number that when added to* A *results in a sum of ten*. So the complement of 1 is 9, of 2 is 8, and so on. Figure 4.69 provides the complements for the numbers 1 through 9.

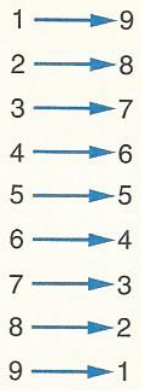
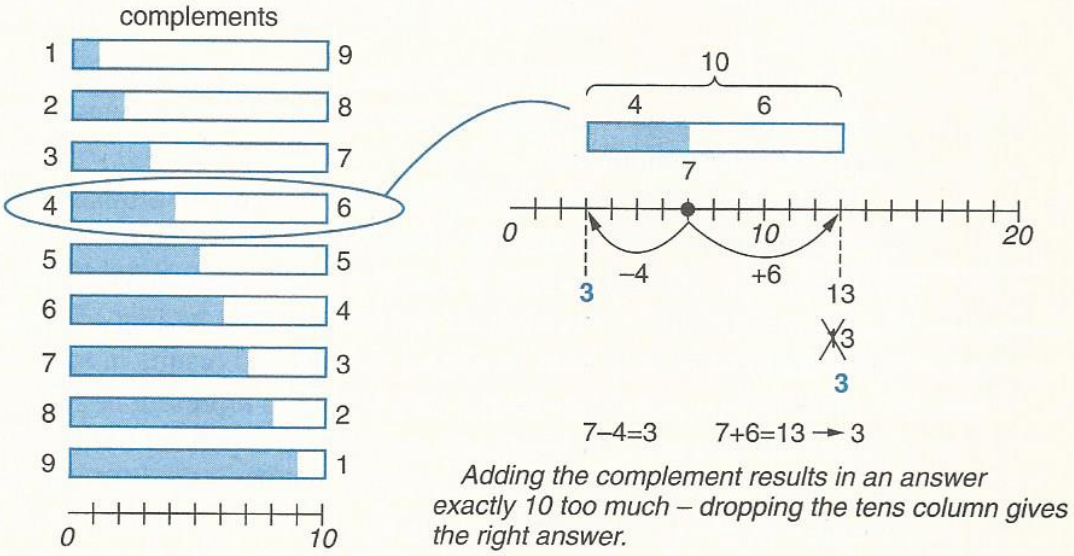


Figure 4.69 Complements in base ten.

The wonderful thing about a complement is that you can use it to perform subtraction using addition, by replacing the number being subtracted with its complement, then by adding, and then by finally throwing away the carry. For example:

7 – 4 —> 7 + 6 = 13 —> ~~1~~3 = 3

We replaced 4 by its complement, 6, and then added 6 to 7 to obtain 13. Finally, we then threw away the carry, leaving 3, which is the correct result. Thus, *we performed subtraction using addition.*



Adding the complement results in an answer exactly 10 too much – dropping the tens column gives the right answer.

Figure 4.70 Subtracting by adding—subtracting a number (4) is the same as adding the number's complement (6) and then dropping the carry, since by definition of the complement, the result will be exactly 10 too much. After all, that's how the complement was defined—the number plus its complement equals 10.

A number line helps us visualize why complements work, as shown in Figure 4.70.

Complements work for any number of digits. Say we want to perform subtraction using two two-digit base ten numbers, perhaps 55 – 30. The complement of 30 would be the number that when added to 30 results in 100, so the complement of 30 is 70. 55 + 70 is 125. Throwing away the carry yields 25, which is the correct result for 55 – 30.

So using complements achieves subtraction using addition.

"Not so fast!" you might say. In order to determine the complement, don't we have to perform subtraction? We know that 6 is the complement of 4 by computing 10 – 4 = 6. We know that 70 is the complement of 30 by computing 100 – 30 = 70. So haven't we just moved the subtraction to another step—the step of computing the complement?

Two's complement can be computed simply by inverting the bits and adding 1—thus avoiding the need for subtraction when computing a complement.

Yes. Except, it turns out that *in base two, we can compute the complement in a much simpler way—just by inverting all the bits and adding 1.* For example, consider computing the complement of the 3-bit base-two number 001. The complement would be the number that when added to 001 yields 1000—you can probably see that the complement should be 111. Using the same method for computing the complement as we did in base ten, we compute the two's complement of 001 as: 1000 - 001 = 111—so 111 is the complement of 001. However, it just so happens that if we invert all the bits of 001 and add 1, we get the same result! Inverting the bits of 001 yields 110; adding 1 yields 110+1 = 111—the correct complement.

Thus, to perform a subtraction, say 011 – 001, we would perform the following:

```
011 – 001
—> 011 + ((001)'+1)
= 011 + (110+1)
= 011 + 111
= 1010 (throw away the carry)
—> 010
```

That's the correct answer, and didn't involve any subtractions—only an invert and additions.

We omit discussion as to why one can compute the complement in base two by inverting the bits and adding 1—for our purposes, we just need to know that that trick works for binary numbers.

There are actually two types of complements of a binary number. The type we've been using above is known as the ***two's complement,*** obtained by inverting all the bits of the binary number and adding 1. Another type is known as the ***one's complement***, which is obtained simply by inverting all the bits, without adding a 1. The two's complement is much more commonly used in digital circuits and results in simpler logic.

Two's complement leads to a simple way to represent negative numbers. Say we have four bits to represent numbers, and we want to represent both positive and negative numbers. We can choose to represent positive numbers as 0000 to 0111 (0 to 7). Negative numbers would be obtained by taking the two's complement of the positive numbers, because a – b is the same as a + (–b). So –1 would be represented by taking the two's complement of 0001, or (0001)'+1 = 1110+1 = 1111. Likewise, -2 would be (0010)'+1 = 1101+1 = 1110. -3 would be (0011)'+1 = 1100+1 = 1101. And so on. –7 would be (0111)'+1 = 1000+1 = 1001. Notice that the two's complement of 0000 is 1111+1 = 0000. Two's complement representation has only one representation of 0, namely, 0000 (unlike signed-magnitude representation, which had two representations of 0). Also notice that we can represent –8 as 1000. So two's complement is slightly asymmetric, representing one more negative number than positive numbers. A 4-bit two's-complement number can represent any number from –8 to +7.

The highest-order bit in two's complement acts as a sign bit: 0 means positive, 1 means negative.

Say you have 4-bit numbers and you want to store –5. –5 would be (0101)'+1 = 1010+1 = 1011. Now you want to add –5 to 4 (or 0100). So we simply add: 1011 + 0100 = 1111, which is –1—the correct answer.

Note that negative numbers all have a 1 in the highest-order bit; thus, the highest-order bit in two's complement is often referred to as the ***sign bit***, 0 indicating a positive number, 1 a negative number.

If you want to know the magnitude of a two's complement negative number, you can obtain the magnitude by taking the two's complement again. So to determine what number 1111 represents, we can take the two's complement of 1111: (1111)'+1 = 0000+1 = 0001. We put a negative sign in front to yield –0001, or –1.

A quick way for humans to mentally figure out the magnitude of a negative number in 4-bit two's complement (having a 1 in the high order bit) is to subtract the magnitude of the three lower bits from 8. So for 1111, the low three bits are 111 or 7, so the magnitude is 8 – 7 = 1, which in turn means that 1111 represents –1. For an 8-bit two's complement number, we would subtract the magnitude of the lower 7 bits from 128. So 10000111 would be –(128–7) = –121.

To summarize, we can represent negative numbers using two's complement representation. Addition of two's complement numbers proceeds unmodified—we just add the numbers. Even if one or both numbers are negative, we simply add the numbers. We perform subtraction of A – B by taking the two's complement of B and then adding that two's complement to A, resulting in A + (–B). We compute the two's complement of B by simply inverting the bits of B and then adding 1.