2.3.3 Converting between Power-of-Two Radices

Binary numbers are often expressed in hexadecimal—and sometimes octal—to improve their readability. Because $16 = 2^4$, a group of 4 bits (called a **hextet**) is easily recognized as a hexadecimal digit. Similarly, with $8 = 2^3$, a group of 3 bits (called an **octet**) is expressible as one octal digit. Using these relationships, we can therefore convert a number from binary to octal or hexadecimal by doing little more than looking at it.

EXAMPLE 2.9 Convert 110010011101_2 to octal and hexadecimal.

110 010 011 101 6 2 3 5	Separate into groups of three for the octal conversion.
	$110010011101_2 = 6235_8$
1100 1001 1101 C 9 D	Separate into groups of 4 for the hexadecimal conversion.
	$110010011101_2 = C9D_{16}$

If there are too few bits, leading zeros can be added.

2.4 SIGNED INTEGER REPRESENTATION

We have seen how to convert an unsigned integer from one base to another. Signed numbers require additional issues to be addressed. When an integer variable is declared in a program, many programming languages automatically allocate a storage area that includes a sign as the first bit of the storage location. By convention, a "1" in the high-order bit indicates a negative number. The storage location can be as small as an 8-bit byte or as large as several words, depending on the programming language and the computer system. The remaining bits (after the sign bit) are used to represent the number itself.

How this number is represented depends on the method used. There are three commonly used approaches. The most intuitive method, signed magnitude, uses the remaining bits to represent the magnitude of the number. This method and the other two approaches, which both use the concept of **complements**, are introduced in the following sections.

2.4.1 Signed Magnitude

Up to this point, we have ignored the possibility of binary representations for negative numbers. The set of positive and negative integers is referred to as the set of **signed integers**. The problem with representing signed integers as binary values is the sign—how should we encode the actual sign of the number? **Signed-**

magnitude representation is one method of solving this problem. As its name implies, a signed-magnitude number has a sign as its left-most bit (also referred to as the high-order bit or the most significant bit) while the remaining bits represent the magnitude (or absolute value) of the numeric value. For example, in an 8-bit word, -1 would be represented as 10000001, and $+1$ as 00000001. In a computer system that uses signed-magnitude representation and 8 bits to store integers, 7 bits can be used for the actual representation of the magnitude of the number. This means that the largest integer an 8-bit word can represent is $2^7 - 1$ or 127 (a zero in the high-order bit, followed by 7 ones). The smallest integer is 8 ones, or -127. Therefore, N bits can represent $-(2^{(N-1)} - 1)$ to $2^{(N-1)} - 1$.

Computers must be able to perform arithmetic calculations on integers that are represented using this notation. Signed-magnitude arithmetic is carried out using essentially the same methods as humans use with pencil and paper, but it can get confusing very quickly. As an example, consider the rules for addition: (1) If the signs are the same, add the magnitudes and use that same sign for the result; (2) If the signs differ, you must determine which operand has the larger magnitude. The sign of the result is the same as the sign of the operand with the larger magnitude, and the magnitude must be obtained by subtracting (not adding) the smaller one from the larger one. If you consider these rules carefully, this is the method you use for signed arithmetic by hand.

We arrange the operands in a certain way based on their signs, perform the calculation without regard to the signs, and then supply the sign as appropriate when the calculation is complete. When modeling this idea in an 8-bit word, we must be careful to include only 7 bits in the magnitude of the answer, discarding any carries that take place over the high-order bit.

EXAMPLE 2.10 Add 01001111_2 to 00100011_2 using signed-magnitude arithmetic.

```
          1 1 1 1    ⇐ carries
0   1 0 0 1 1 1 1        (79)
0 + 0 1 0 0 0 1 1      + (35)
0   1 1 1 0 0 1 0        (114)
```

The arithmetic proceeds just as in decimal addition, including the carries, until we get to the seventh bit from the right. If there is a carry here, we say that we have an overflow condition and the carry is discarded, resulting in an incorrect sum. There is no overflow in this example.

We find $01001111_2 + 00100011_2 = 01110010_2$ in signed-magnitude representation.

Sign bits are segregated because they are relevant only after the addition is complete. In this case, we have the sum of two positive numbers, which is positive.

Overflow (and thus an erroneous result) in signed numbers occurs when the sign of the result is incorrect.

In signed magnitude, the sign bit is used only for the sign, so we can't "carry into" it. If there is a carry emitting from the seventh bit, our result will be truncated as the seventh bit overflows, giving an incorrect sum. (Example 2.11 illustrates this overflow condition.) Prudent programmers avoid "million dollar" mistakes by checking for overflow conditions whenever there is the slightest possibility that they could occur. If we did not discard the overflow bit, it would carry into the sign, causing the more outrageous result of the sum of two positive numbers being negative. (Imagine what would happen if the next step in a program were to take the square root or log of that result!)

EXAMPLE 2.11 Add 01001111_2 to 01100011_2 using signed-magnitude arithmetic.

Last carry	1	←		1	1	1	1	⇐ carries	
overflows and is	0		1	0	0	1	1	1	1	(79)
discarded.	0	+	1	1	0	0	0	1	1	+ (99)
	0		0	1	1	0	0	1	0	(50)

We obtain the erroneous result of 79 + 99 = 50.

Dabbling on the Double

The fastest way to convert a binary number to decimal is a method called **double-dabble** (or **double-dibble**). This method builds on the idea that a subsequent power of two is double the previous power of two in a binary number. The calculation starts with the leftmost bit and works toward the rightmost bit. The first bit is doubled and added to the next bit. This sum is then doubled and added to the following bit. The process is repeated for each bit until the rightmost bit has been used.

EXAMPLE 1

Convert 10010011_2 to decimal.

Step 1: Write down the binary number, leaving space between the bits.

1 0 0 1 0 0 1 1

Step 2: Double the high-order bit and copy it under the next bit.

1	0	0	1	0	0	1	1
	2						
× 2							
2							

Step 3: Add the next bit and double the sum. Copy this result under the next bit.

1	0	0	1	0	0	1	1
	2	4					
	+ 0						
	2						
× 2	× 2						
2	4						

Step 4: Repeat Step 3 until you run out of bits.

1	0	0	1	0	0	1	1	
	2	4	8	18	36	72	146	
	+ 0	+ 0	+ 1	+ 0	+ 0	+ 1	+ 1	
	2	4	9	18	36	73	147	⇐ The answer: $10010011_2 = 147_{10}$
× 2	× 2	× 2	× 2	× 2	× 2	× 2		
2	4	8	18	36	72	146		

When we combine hextet grouping (in reverse) with the double-dabble method, we find that we can convert hexadecimal to decimal with ease.

EXAMPLE 2

Convert $02CA_{16}$ to decimal.

First, convert the hex to binary by grouping into hextets.

0	2	C	A
0000	0010	1100	1010

Then apply the double-dabble method on the binary form:

1	0	1	1	0	0	1	0	1	0
	2	4	10	22	44	88	178	356	714
	+ 0	+ 1	+ 1	+ 0	+ 0	+ 1	+ 0	+ 1	+ 0
	2	5	11	22	44	89	178	357	714

×2	×2	×2	×2	×2	×2	×2	×2	×2
2	4	10	22	44	88	178	356	714

$02CA_{16} = 1011001010_2 = 714_{10}$

As with addition, signed-magnitude subtraction is carried out in a manner similar to pencil and paper decimal arithmetic, where it is sometimes necessary to borrow from digits in the **minuend**.

≡ **EXAMPLE 2.12** Subtract 01001111_2 from 01100011_2 using signed-magnitude arithmetic.

		0	1	1	2		⇐ borrows	
0		1	~~1~~	~~0~~	~~0~~	~~0~~	1	1	(99)
0	−	1	0	0	1	1	1	1	− (79)
0		0	0	1	0	1	0	0	(20)

We find $01100011_2 - 01001111_2 = 00010100_2$ in signed-magnitude representation.

≡ **EXAMPLE 2.13** Subtract 01100011_2 (99) from 01001111_2 (79) using signed-magnitude arithmetic.

By inspection, we see that the subtrahend, 01100011, is larger than the minuend, 01001111. With the result obtained in Example 2.12, we know that the difference of these two numbers is 0010100_2. Because the subtrahend is larger than the minuend, all that we need to do is change the sign of the difference. So we find $01001111_2 - 01100011_2 = 10010100_2$ in signed-magnitude representation.

We know that subtraction is the same as "adding the opposite," which equates to negating the value we wish to subtract and then adding instead (which is often much easier than performing all the borrows necessary for subtraction, particularly in dealing with binary numbers). Therefore, we need to look at some examples involving both positive and negative numbers. Recall the rules for addition: (1) If the signs are the same, add the magnitudes and use that same sign for the result; (2) If the signs differ, you must determine which operand has the larger magnitude. The sign of the result is the same as the sign of the operand with the larger magnitude, and the magnitude must be obtained by subtracting (not adding) the smaller one from the larger one.

≡ **EXAMPLE 2.14** Add 10010011_2 (−19) to 00001101_2 (+13) using signed-magnitude arithmetic.

The first number (the augend) is negative because its sign bit is set to 1. The second number (the addend) is positive. What we are asked to do is in fact a subtraction. First, we determine which of the two numbers is larger in magnitude and use that number for the augend. Its sign will be the sign of the result.

$$\begin{array}{cr} & 0\ 1\ 2 \quad \Leftarrow \text{borrows} \\ 1 & 0\ 0\ \cancel{1}\ \cancel{0}\ \cancel{0}\ 1\ 1 \quad (-19) \\ 0\ - & \underline{0\ 0\ 0\ 1\ 1\ 0\ 1} \quad \underline{+\,(13)} \\ 1 & 0\ 0\ 0\ 0\ 1\ 1\ 0 \quad (-6) \end{array}$$

With the inclusion of the sign bit, we see that $10010011_2 - 00001101_2 = 10000110_2$ in signed-magnitude representation.

EXAMPLE 2.15 Subtract 10011000_2 (-24) from 10101011_2 (-43) using signed-magnitude arithmetic.

We can convert the subtraction to an addition by negating -24, which gives us 24, and then we can add this to -43, giving us a new problem of $-43 + 24$. However, we know from the addition rules above that because the signs now differ, we must actually subtract the smaller magnitude from the larger magnitude (or subtract 24 from 43) and make the result negative (since 43 is larger than 24).

$$\begin{array}{cr} & 0\ 2 \\ & 0\ \cancel{1}\ 0\ 1\ 0\ 1\ 1 \quad (43) \\ -\ & \underline{0\ 0\ 1\ 1\ 0\ 0\ 0} \quad \underline{-\,(24)} \\ & 0\ 0\ 1\ 0\ 0\ 1\ 1 \quad (19) \end{array}$$

Note that we are not concerned with the sign until we have performed the subtraction. We know the answer must be negative. So we end up with $10101011_2 - 10011000_2 = 10010011_2$ in signed-magnitude representation.

While reading the preceding examples, you may have noticed how many questions we had to ask ourselves: Which number is larger? Am I subtracting a negative number? How many times do I have to borrow from the minuend? A computer engineered to perform arithmetic in this manner must make just as many decisions (though a whole lot faster). The logic (and circuitry) is further complicated by the fact that signed magnitude has two representations for zero, 10000000 and 00000000 (and mathematically speaking, this simply shouldn't happen!). Simpler methods for representing signed numbers would allow simpler and less expensive circuits. These simpler methods are based on radix complement systems.

2.4.2 Complement Systems

Number theorists have known for hundreds of years that one decimal number can be subtracted from another by adding the difference of the subtrahend from all nines and adding back a carry. This is called taking the nine's complement of the subtrahend, or more formally, finding the **diminished radix complement** of the subtrahend. Let's say we wanted to find 167 − 52. Taking the difference of 52 from 999, we have 947. Thus, in nine's complement arithmetic we have 167 − 52 = 167 + 947 = 1114. The "carry" from the hundreds column is added back to the units place, giving us a correct 167 − 52 = 115. This method was commonly called "casting out 9s" and has been extended to binary operations to simplify computer arithmetic. The advantage that complement systems give us over signed magnitude is that there is no need to process sign bits separately, but we can still easily check the sign of a number by looking at its high-order bit.

Another way to envision complement systems is to imagine an odometer on a bicycle. Unlike cars, when you go backward on a bike, the odometer will go backward as well. Assuming an odometer with three digits, if we start at zero and end with 700, we can't be sure whether the bike went forward 700 miles or backward 300 miles! The easiest solution to this dilemma is simply to cut the number space in half and use 001–500 for positive miles and 501–999 for negative miles. We have, effectively, cut down the distance our odometer can measure. But now if it reads 997, we know the bike has backed up 3 miles instead of riding forward 997 miles. The numbers 501–999 represent the **radix complements** (the second of the two methods introduced below) of the numbers 001–500 and are being used to represent negative distance.

One's Complement

As illustrated above, the diminished radix complement of a number in base 10 is found by subtracting the subtrahend from the base minus one, which is 9 in decimal. More formally, given a number N in base r having d digits, the diminished radix complement of N is defined to be $(r^d - 1) - N$. For decimal numbers, $r = 10$, and the diminished radix is $10 - 1 = 9$. For example, the nine's complement of 2468 is $9999 - 2468 = 7531$. For an equivalent operation in binary, we subtract from one less the base (2), which is 1. For example, the one's complement of 0101_2 is $1111_2 - 0101 = 1010$. Although we could tediously borrow and subtract as discussed above, a few experiments will convince you that forming the one's complement of a binary number amounts to nothing more than switching all of the 1s with 0s and vice versa. This sort of bit-flipping is very simple to implement in computer hardware.

It is important to note at this point that although we can find the nine's complement of any decimal number or the one's complement of any binary number, we are most interested in using complement notation to represent negative numbers. We know that performing a subtraction, such as 10 − 7, can also be thought