Binary Numbering Systems

Introduction

Binary numbering systems are used in virtually all digital systems, including digital signal processing (DSP), networking, and computers. Before you choose a numbering system, it is important to understand the advantages and disadvantages of each system. This application note describes the following numbering systems, the advantages and disadvantages of each, and how to convert between different systems.

- Unsigned integer
- Two's complement integer
- Unsigned fractional
- Two's complement signed fractional
- Gray code
- Signed-magnitude
- Offset two's complement
- One's complement
- Floating point
- Block floating point

Table 1 summarizes the binary numbering systems described in this application note. For each numbering system, the number range is provided for an N-bit number. For fractional numbering systems, the number range is provided for an $N + M$-bit number, where N is the integer part of the number and M is the fractional part of the number.

Table 1. Summary of the Binary Numbering Systems			
System	**Number Range**	**Advantages**	**Disadvantages**
Unsigned Integer	0 to $2^N - 1$	Universal numbering system. Easy to perform arithmetic operations such as addition and subtraction.	Cannot store negative numbers.
Two's Complement Integer	$-2^{(N-1)}$ to $2^{(N-1)} - 1$	Stores both positive and negative numbers. Easy to perform arithmetic with regular adders.	Requires one extra bit of storage space when only positive numbers are necessary.
Unsigned Fractional	0 to $2^N - 2^M$	Stores positive numbers greater than and less than 1. Operations are identical to unsigned integer operations.	Cannot store negative numbers.
Two's Complement Signed Fractional	$-2^{(N-1)}$ to $2^{(N-1)} - 2^{-M}$ in 2^{-M} steps	Stores positive and negative numbers both greater than and less than 1. Operations are identical to two's complement operations.	–
Gray Code	0 to $2^{(N-1)}$	Only one bit changes between adjacent numbers, which facilitates interfaces with physical systems.	Difficult to perform arithmetic operations without first converting to one of the systems listed above.
Signed-Magnitude	$-2^{(N-1)} - 1$ to $2^{(N-1)} - 1$	Useful for applications that require the magnitude to be distinct from the sign.	Difficult to perform arithmetic operations (although easier than with Gray code).
Offset Two's Complement	$-2^{(N-1)}$ to $2^{(N-1)} - 1$	Used by many analog-to-digital (A/D) and digital-to-analog (D/A) converters. Easy to perform arithmetic operations.	–
One's Complement	$-2^{(N-1)} - 1$ to $2^{(N-1)} - 1$	Easy to perform negations.	Difficult to perform arithmetic operations other than negations.
Floating Point	See "Floating Point" on page 13.	Very large dynamic range.	Requires more hardware to perform arithmetic.
Block Floating Point	See "Block Floating Point" on page 15.	Large dynamic range and requires minimal hardware.	All numbers have the same exponent at any given time.

Table 2 shows a 3-bit binary number and its equivalent decimal value for each numbering system.

Table 2. Values for a 3-Bit Binary Number

Binary Number	Unsigned Integer	Two's Complement	Gray Code	Signed Magnitude	Offset Two's Complement	One's Complement
000	0	0	0	0	−4	0
001	1	1	1	1	−3	1
010	2	2	3	2	−2	2
011	3	3	2	3	−1	3
100	4	−4	7	−0	0	−3
101	5	−3	6	−1	1	−2
110	6	−2	4	−2	2	−1
111	7	−1	5	−3	3	0

Unsigned Integer

The best known numbering system is unsigned integer representation. Like the decimal numbering system, unsigned integers use a simple binary place value. The position of a digit determines its value (i.e., the place value of a digit is $2^{position}$). See Figure 1. This representation is exactly like the decimal numbering system, in which the place value is $10^{position}$.

Figure 1. Bit Position

Binary Number	1	1	1	1	1
Position	4	3	2	1	0
Place Value	2^4	2^3	2^2	2^1	2^0

Table 3 shows the decimal value for each bit in a 5-bit unsigned integer.

Table 3. Unsigned Integer Values

Position	Place Value	Decimal Value
0	2^0	1
1	2^1	2
2	2^2	4
3	2^3	8
4	2^4	16

Table 4 shows how to determine the decimal value of an unsigned integer.

Table 4. Unsigned Integer Conversion		
Unsigned Integer	**Decimal Value**	**Conversion**
01000	8	$0 + 2^3 + 0 + 0 + 0 = 8$
10011	19	$2^4 + 0 + 0 + 2^1 + 2^0 = 16 + 2 + 1 = 19$
11011	27	$2^4 + 2^3 + 0 + 2^1 + 2^0 = 16 + 8 + 2 + 1 = 27$

You can easily perform arithmetic operations on unsigned numbers by following the same rules used for decimal number operations. However, for binary numbers, the digit is carried after 1 rather than after 9 (i.e., when two 1s are added together, a 0 is placed in the corresponding position and a 1 is carried to the next position). Figure 2 shows how to add 2 unsigned integers together.

Figure 2. Unsigned Integer Addition

```
    1 1 0 1 1           27
+   1 0 0 1 1    =    + 19
    1 0 1 1 1 0         46
```

The unsigned integer numbering system is widely used. This numbering system's main limitation is that it can only store the numbers ranging from 0 to $(2^N - 1)$; most signal processing systems need to store both positive and negative numbers.

Two's Complement Integer

The most commonly used numbering system that can store both positive and negative numbers is two's complement integer. This system is similar to unsigned integers, except the sign of the most significant bit (MSB) is negated. For example, for an *N*-bit number, bit 0 has a value of 2^0, bit 1 has a value of 2^1, bit $N - 2$ has a value of $2^{(N-2)}$, and bit $N - 1$ (i.e., the MSB) has a value of $-2^{(N-1)}$. Table 5 shows the decimal value for each position in a 5-bit two's complement integer.

Table 5. Two's Complement Integer Values for a 5-Bit Number		
Position	Place Value	Decimal Value
0	2^0	1
1	2^1	2
2	2^2	4
3	2^3	8
4 (MSB)	-2^4	−16

Table 6 shows how to determine the decimal value of a two's complement number.

Table 6. Two's Complement Integer Conversion		
Two's Complement Integer	Decimal Value	Conversion
01000	8	$0 + 2^3 + 0 + 0 + 0 = +8$
11000	−8	$-2^4 + 2^3 + 0 + 0 + 0 = -16 + 8 = -8$
10000	−16	$-2^4 + 0 + 0 + 0 + 0 = -16$
10111	−9	$-2^4 + 0 + 2^2 + 2^1 + 2^0 = -16 + 0 + 4 + 2 + 1 = -9$

Two's complement integers represent numbers ranging from $-2^{(N-1)}$ to $2^{(N-1)} - 1$. To negate a two's complement integer, you simply invert the bits and add 1. For example, the following steps show how to negate the number 9 to −9.

1. Substitute the binary values of the decimal number: 9 = 01001

2. Invert the bits: 10110

3. Add 1:   (10110 + 1) = 10111

As shown in Table 6, 10111 is equal to −9.

The biggest advantage of the two's complement numbering system is that adding and subtracting two's complement numbers is the same as adding and subtracting unsigned numbers. However, sign extension must be performed before the operation and any carry out of the adder should be ignored. Figure 3 shows how to add two 2-bit two's complement numbers.

Figure 3. Two's Complement Integer Addition

Positive + Positive	Negative + Positive	Negative + Negative

```
Positive + Positive        Negative + Positive        Negative + Negative

    1      0 0 1               −1      1 1 1               −1      1 1 1
  + 1    + 0 0 1              + 1    + 0 0 1              − 2    + 1 1 0
    2      0 0 0 1               0      1 0 0 0             −3      1 1 0 1
```

In Figure 3, the digits shaded in blue represent the bits used for sign
extension. The blue digits are the carry out of the 3-bit adder, which must
be ignored. Other than the sign extension and ignoring the carry-out of
the MSB of the adder, addition for two's complement numbers is identical
to addition for unsigned integer numbers.

Unsigned Fractional

In DSP and other systems, it is often necessary to store numbers that have
both an integer and a fractional component. Because some bit positions
can be negative, the unsigned fractional numbering system can store
numbers greater than and less than 1. The place value of a digit in the
unsigned fractional numbering system is $2^{position}$, where the position can
be positive or negative (see Figure 4). Therefore, the unsigned fractional
numbering system is a superset of the unsigned integer numbering
system.

Figure 4. Bit Position

Binary Number	1	1	1	1	1
Position	2	1	0	−1	−2
Place Value	2^2	2^1	2^0	2^{-1}	2^{-2}

Table 7 shows the decimal value for each bit in an unsigned fractional
number.

Table 7. Unsigned Fractional Values

Position	Place Value	Decimal Value
−2	2^{-2}	0.25
−1	2^{-1}	0.50
0	2^0	1
1	2^1	2
2	2^2	4

Table 8 shows how to determine the decimal value of an unsigned
fractional number.

Table 8. Unsigned Fractional Conversion		
Unsigned Fractional Number	Decimal Value	Conversion
01001	2.25	$0 + 2^1 + 0 + 0 + 2^{-2} = 2 + 0.25 = 2.25$
11011	6.75	$2^2 + 2^1 + 0 + 2^{-1} + 2^{-2} = 4 + 2 + 0.5 + 0.25 = 6.75$
00010	0.5	$0 + 0 + 0 + 2^{-1} + 0 = 0.5$

Unsigned fractional numbers use a convenient notation to keep track of the location of a radix point (i.e., the binary or decimal point); a number with N bits to the left of the radix point and M bits to the right is said to be an $N.M$ number (e.g., an *8.3* number has 8 digits to the left of the radix point and 3 digits to the right).

If all numbers in your system have the same M value (i.e., the same fractional bit-width), arithmetic operations are straightforward. For example, you can add an *8.3* number to a *12.3* number by using a 15-bit binary adder. Figure 5 shows how to add 00100101.101 (37.625) to 001101110011.001 (883.125).

Figure 5. Addition with the Same M Values

```
      37.625              00100101.101
   + 883.125         +  001101110011.001
     920.750           001110011000.110
```

To add numbers with different M values, you must add extra zeros to keep the radix points aligned. For example, to add an *8.3* number to a *6.5* number, you must pad the *8.3* number with zeros to create an *8.5* number. Thus, you must use a 13-bit adder rather than an 11-bit adder to add the two numbers. Figure 6 shows how to add 11011011.110 (219.750) to 110111.11011 (55.84375).

Figure 6. Addition with Different M Values

Digits used for padding are highlighted in blue.

```
     219.75000           11011011.11000
   + 055.84375        +  00110111.11011
     275.59375          100010011.10101
```

Other than aligning the radix points, there is no difference between the unsigned integer and unsigned fractional numbering systems. In fact, an unsigned integer number is simply an *N.0* unsigned fractional number (i.e., unsigned integer numbers have no bits to the right of the radix point). Any hardware built for integer numbers will work with fractional numbers.

Two's Complement Signed Fractional

Like unsigned fractional, the two's complement signed fractional numbering system uses an *N.M* notation and must have the radix points aligned during arithmetic operations.

Figure 7 shows how to add an *8.3* number to a *5.5* number.

Figure 7. Two's Complement Signed Fractional Addition

Digits used for padding are highlighted in blue.

```
    −36.25000        111011011.11000
 +   13.90625      + 000001101.11101
    −22.34375        100010011.10101
```

Again, the numbers are padded with zeros and sign-extended to obtain the correct result.

Gray Code

Gray code is a numbering system that is used mainly in real-world sensing applications. The fundamental feature of Gray code is that only one bit changes at a time as you progress sequentially through the numbers. See Table 9.

Table 9. Gray Code Values	
Gray Code Number	**Decimal Value**
001	1
011	2
010	3
110	4
111	5

To understand why a numbering system in which only one bit changes at a time is useful, consider the optical sensor shown in Figure 8. This sensor is a shaft encoder that gives the physical position (i.e., rotation) of the shaft. For simplicity, the sensor is shown with only a 2-bit code. The sensor has 2 light-emitting diodes (LEDs) and 2 photo detectors for sensing the position of the shaft. The number of bits represented by the wheel depends on how closely you can space the cut-outs in the wheel without the wheel becoming too flimsy, the diameter of the light beam, and other physical parameters. Figure 8 shows a binary shaft encoder with 2 signals: s1 and s0.
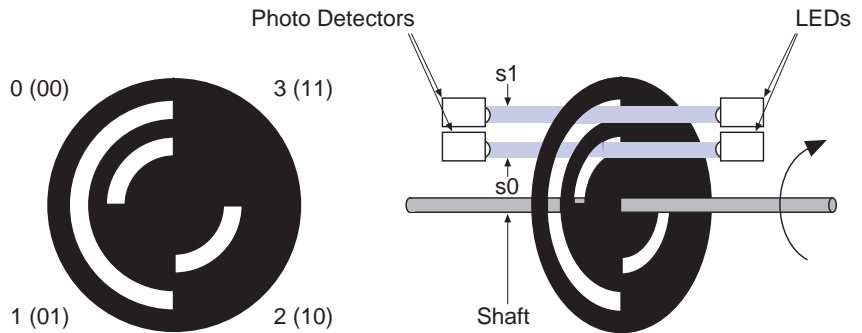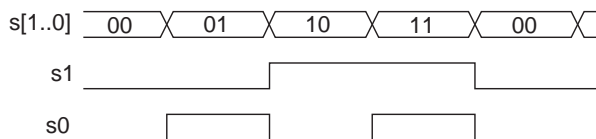
Figure 8. Straight Binary Shaft Encoder

Figure 9 shows the ideal waveforms of s1 and s0 as the wheel cycles through one full rotation.

Figure 9. Ideal Waveforms of s1 & s0

In contrast, the real waveforms generated by the optical sensors vary slowly and the sensors are always misaligned by at least a small amount, which causes the signals to resemble the waveforms shown in Figure 10.

Figure 10. Waveforms Caused by Misaligned Sensors
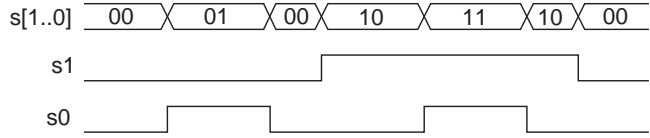
When the optical sensors are misaligned, `s1` and `s0` may not switch at the same instant. Therefore, intervening codes appear in the sensed shaft position. In Figure 10, the `s[1..0]` signal should have cycled through the shaft position sequence of 00 ➜ 01 ➜ 10 ➜ 11 ➜ 00. However, it actually cycles through the sequence 00 ➜ 01 ➜ 00 ➜ 10 ➜ 11 ➜ 10 ➜ 00, which is incorrect. This error could be catastrophic if the shaft encoder is, for example, sensing the rudder position on an airplane.

The fundamental problem with non-Gray code is that more than 1 bit changes when going from one position to another, such as going from position 1 (01) to position 2 (10). In 2-bit Gray code, the decimal sequence 0 ➜ 1 ➜ 2 ➜ 3 ➜ 0 is represented as 00 ➜ 01 ➜ 11 ➜ 10 ➜ 00. Notice that only 1 bit changes between any 2 adjacent numbers, which prevents the problems incurred with simultaneous switching in misaligned sensors. Figure 11 shows the shaft encoder for 2-bit Gray code.

Figure 11. Shaft Encoder for 2-Bit Gray Code
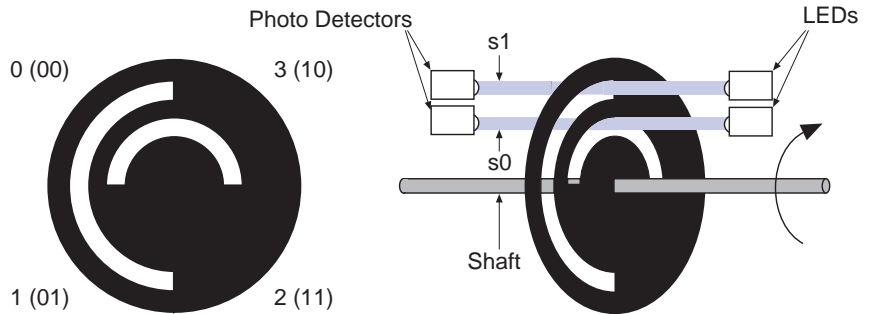
When the sensors on this encoder are misaligned, even by a significant amount, there are no extra codes in the sequence 0 ➜ 1 ➜ 2 ➜ 3 ➜ 0, as shown in Figure 12.

Figure 12. Gray Code Sequence with Significant Misalignment
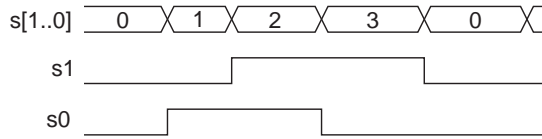
Unfortunately, it is difficult to perform arithmetic on Gray code numbers. It is easiest to perform operations by first converting the numbers from Gray code to either unsigned or signed two's complement numbers after sensing.

You can create longer Gray code sequences by concatenating the smaller sequence with a reversed version of the same sequence, and setting the MSB to 0 and 1. For example, the following steps show how to build a 3-bit Gray code:

1.   For the first 4 numbers, append a 0 to the MSB of the 2-bit Gray code.

2.   For the last 4 numbers, reverse the order of the 2-bit Gray code and append a 1 to the MSB.

This method ensures that only 1 bit changes between any two adjacent numbers. See Table 10.

Table 10. Creating Longer Gray Code Sequences

Decimal Value	2-Bit Gray Code	3-Bit Gray Code
0	00	000
1	01	001
2	11	011
3	10	010
4	–	110
5	–	111
6	–	101
7	–	100

Signed-Magnitude

Signed-magnitude is useful for applications in which the sign and magnitude of a number must be accessed separately. In signed-magnitude systems, the MSB represents the sign of the number (i.e., 0 = positive, 1 = negative) and all other bits represent the magnitude. This notation is similar to decimal notation, which uses a + and – as the sign bit and uses the remaining bits to represent the magnitude.

The signed-magnitude and two's complement numbering systems both use the MSB to determine the sign. However, do not confuse signed-magnitude with two's complement. Although the MSB can be used to determine the sign in two's complement, the other bits do not represent the magnitude when the sign is negative. In addition, two's complement has only one representation of zero, whereas the signed-magnitude numbering system has two (i.e., +0 and –0).

To perform any complex arithmetic operations on signed-magnitude data, it is usually easiest to convert the numbers to two's complement, perform the operations, and then convert the numbers back to the signed-magnitude system, if required.

Offset Two's Complement

The offset two's complement numbering system is used by many D/A and A/D converters. The distinguishing characteristic of this numbering system is that the numbers move from –4 to 3 monotonically as you count in binary from 000 to 111—there are no jumps or discontinuities. In contrast, the decimal value of two's complement numbers goes from 0 to 3 and then counts from –4 to –1 as the binary number progresses from 000 to 111.

To convert numbers from two's complement to offset two's complement, you simply invert the MSB, as shown in Table 11.

Table 11. Converting to Offset Two's Complement		
Decimal Value	Two's Complement	Offset Two's Complement
–4	100	000
–3	101	001
–2	110	010
–1	111	011
0	000	100
1	001	101
2	010	110
3	011	111

One's Complement

The one's complement numbering system is seldom used because it has the same drawbacks as all non-two's complement numbers—it is difficult to perform arithmetic and it has two representations of zero (i.e., +0 and –0).

When you add a one's complement number to its inverse, the answer is not zero, which creates an inconsistent algebra. In fact, as Table 2 on page 3 shows, the two representations of zero are 000 and 111, which makes zero sensing more difficult (i.e., you must use an N-bit OR and an N-bit AND gate). To negate a number in the one's complement numbering system, you simply invert all the bits (i.e., you do not add a 1 like in two's complement).

Floating Point

All of the previous numbering systems have one common characteristic: they have a relatively small dynamic range between the largest and smallest numbers they can store. The power dynamic range in decibels (dB) of all previous numbering systems is approximately $20\log(2^N)$, where N is the total number of bits in the numbering system. For an 8-bit number, the dynamic range is 48.1 dB; for a 16-bit number, the dynamic range is 96.3 dB. In many cases, you may need larger dynamic ranges. For example, you need a large dynamic range to perform a simple physical calculation, such as figuring out the mass (in grams) of 1,000 atoms of carbon.

The following equation can be used to calculate the mass of carbon.

$m = n \times w / \text{N}$

where: m = Mass (grams)
      n = Number of atoms = 1,000
      w = Atomic mass (grams/mole) = 12.0107
      N = Avogadro's number (1/mole) = 6.022×10^{23}

Calculating the mass with a fixed-point numbering system is similar to performing calculations without scientific notation. To represent both Avogadro's number to 23 decimal places and the atomic mass accurate to 0.0001 simultaneously, you need 28 decimal digits of accuracy to represent the input operands, and another 19 digits to represent the result—a total of 47 decimal digits. To perform the same calculation in binary, you would require 157 bits of accuracy. To illustrate, the following calculation shows how to calculate the mass with fixed-point arithmetic (all 47 digits are used to illustrate the fact that you would similarly have to keep track of all 157 bits in hardware):

$m =$ 00000000000000000001000.00000000000000000000000 $\times$
     00000000000000000000012.01070000000000000000000 $/$
     60220000000000000000000.00000000000000000000000 $=$
     00000000000000000000000.00000000000000000001995

Obviously, a 157-bit multiplier/divider would be a costly circuit. To perform this calculation by hand, you can use scientific notation, as shown below:

$$m = \frac{(1.00000 \times 10^3) \times (1.20107 \times 10^1)}{1.99447 \times 10^{-20}}$$

This method of storing numbers is easier to use for numbers with differing magnitudes. The scientific notation requires only one digit to the left of the radix point, 5 digits to the right (these 6 digits are known as the mantissa), along with 2 extra digits for the exponent. This method requires a multiplier/divider with only 6 digits (or 20 bits) of accuracy.

The place values of digits next to the radix point can have extremely different magnitudes, which depend directly upon the exponent. You can think of this as a "floating" decimal point. Thus, scientific notation is simply floating-point notation for decimal calculations.

To perform this calculation in digital systems, you use the same concept of mantissa and exponent. The mantissa is the fractional part of the number, and the exponent gives the power of 2 by which the mantissa should be multiplied.

The following example shows the floating-point representation of the decimal number 3072.

3072.0 (decimal) = 110000000000.0 (binary)

You can use a 2-bit mantissa and a 4-bit exponent to represent this number in floating-point notation, as shown below:

3×2^{10} (decimal) = 11×2^{1010} (binary)

This notation requires only 6 bits to store the number, as opposed to the 12 bits used for fixed-point notation.

In floating-point format, a radix point is placed to the right of the most significant 1, and the exponent is adjusted accordingly. For example, the equation shown above is represented as 1.1×2^{1011}, rather than 11×2^{1010}. This format is called normalized floating point. Because a 1 is always to the left of the binary point in normalized floating-point numbers, the 1 can be assumed and not stored, leaving an extra bit for storing the mantissa or exponent.

For example, the number 3,072 is stored by directly concatenating the mantissa and exponent. With a 2-bit mantissa and 4-bit exponent, you obtain 111011. Using normalized floating-point format you obtain 101011 (again with 2 bits of mantissa, and 4 bits of exponent).

A variety of floating-point formats are available, each with various advantages and disadvantages. For more information on other floating-point formats, see *Functional Specification 2 (fp_add_sub Floating-Point Adder/Subtractor)* and *Functional Specification 4 (fp_mult Floating-Point Multiplier)*.

Block Floating Point

The block floating point numbering system is used in signal processing applications dealing with numbers that vary widely throughout a calculation, such as the fast Fourier transform (FFT) function. For example, the data for an FFT function may come from a 16-bit A/D converter, which has a dynamic range of 96 dB. As the FFT processes data, the numbers grow significantly throughout the calculation. Because floating-point hardware is more expensive than fixed-point hardware, many FFT calculations are performed with fixed-point hardware. You can use floating-point hardware economically if you are willing to give up dynamic range for any given set of data and to keep track of the magnitude of the entire set of data separately from each individual piece of data.

For example, an FFT function may start with 256 points of data, each with a 16-bit value between –32,768 and 32,767. On the first pass through the FFT, the values may range between –65,536 and 65,536, which requires extra precision. To keep the data 16-bits wide, you can divide all 256 points by 2 and store a 1 in a register. This method requires only 1 extra register to keep the data at 16 bits.

This method is called block floating point because the radix point on the entire block of data moves at the same time, which keeps the radix points aligned with each other. In true floating point, the radix point of every piece of data is in a different location.

The floating point numbering system should not be confused with the block floating point numbering system. They are dramatically different and require different hardware for computation.

For more information on FFTs, see the *fft Fast Fourier Transform Data Sheet*.

2610 Orchard Parkway
San Jose, CA 95134-2020
(408) 544-7000
http://www.altera.com
Applications Hotline:
(800) 800-EPLD
Customer Marketing:
(408) 894-7104
Literature Services:
(888) 3-ALTERA
lit_req@altera.com

I.S. EN ISO 9001

Altera Corporation

Printed on Recycled Paper.