The sign bit is 1, so the number is negative. Converting to decimal gives

37A (hex) = −134 (dec)

Notice that the hexadecimal number is not written with a negative sign, even though it may be interpreted as a negative number.

ASCII Characters

ASCII

Because computer memories are binary, alphabetic characters must be coded to be stored in memory. A widespread binary code for alphabetic characters is the *American Standard Code for Information Interchange*, also known as *ASCII* (pronounced *askey*).

ASCII contains all the uppercase and lowercase English letters, the 10 numeric digits, and special characters such as punctuation signs. Some of its symbols are nonprintable and are used mainly to transmit information between computers or to control peripheral devices.

ASCII is a seven-bit code. Because there are $2^7 = 128$ possible combinations of seven bits, there are 128 ASCII characters. FIGURE 3.25 shows all these characters. The first column of the table shows the nonprintable characters, whose meanings are listed at the bottom. The rest of the table lists the printable characters.

Example 3.30 The sequence 000 0111, which stands for *bell*, causes a terminal to beep. Two other examples of nonprintable characters are *ACK* for *acknowledge* and *NAK* for *negative acknowledge*, which are used by some data transmission protocols. If the sender sends a packet of information over the channel that is detected error-free, the receiver sends an ACK back to the sender, which then sends the next packet. If the receiver detects an error, it sends a NAK back to the sender, which then resends the packet that was damaged in the initial transmission.

Example 3.31 The name

```
Tom
```

would be stored in ASCII as

101 0100
110 1111
110 1101

If that sequence of bits were sent to an output terminal, the word "Tom" would be displayed.

FIGURE 3.25
The American Standard Code for Information Interchange (ASCII).

Char	Bin	Hex	Char	Bin	Hex	Char	Bin	Hex	Char	Bin	Hex
NUL	000 0000	00	SP	010 0000	20	@	100 0000	40	`	110 0000	60
SOH	000 0001	01	!	010 0001	21	A	100 0001	41	a	110 0001	61
STX	000 0010	02	"	010 0010	22	B	100 0010	42	b	110 0010	62
ETX	000 0011	03	#	010 0011	23	C	100 0011	43	c	110 0011	63
EOT	000 0100	04	$	010 0100	24	D	100 0100	44	d	110 0100	64
ENQ	000 0101	05	%	010 0101	25	E	100 0101	45	e	110 0101	65
ACK	000 0110	06	&	010 0110	26	F	100 0110	46	f	110 0110	66
BEL	000 0111	07	'	010 0111	27	G	100 0111	47	g	110 0111	67
BS	000 1000	08	(	010 1000	28	H	100 1000	48	h	110 1000	68
HT	000 1001	09	)	010 1001	29	I	100 1001	49	i	110 1001	69
LF	000 1010	0A	*	010 1010	2A	J	100 1010	4A	j	110 1010	6A
VT	000 1011	0B	+	010 1011	2B	K	100 1011	4B	k	110 1011	6B
FF	000 1100	0C	,	010 1100	2C	L	100 1100	4C	l	110 1100	6C
CR	000 1101	0D	–	010 1101	2D	M	100 1101	4D	m	110 1101	6D
SO	000 1110	0E	.	010 1110	2E	N	100 1110	4E	n	110 1110	6E
SI	000 1111	0F	/	010 1111	2F	O	100 1111	4F	o	110 1111	6F
DLE	001 0000	10	0	011 0000	30	P	101 0000	50	p	111 0000	70
DC1	001 0001	11	1	011 0001	31	Q	101 0001	51	q	111 0001	71
DC2	001 0010	12	2	011 0010	32	R	101 0010	52	r	111 0010	72
DC3	001 0011	13	3	011 0011	33	S	101 0011	53	s	111 0011	73
DC4	001 0100	14	4	011 0100	34	T	101 0100	54	t	111 0100	74
NAK	001 0101	15	5	011 0101	35	U	101 0101	55	u	111 0101	75
SYN	001 0110	16	6	011 0110	36	V	101 0110	56	v	111 0110	76
ETB	001 0111	17	7	011 0111	37	W	101 0111	57	w	111 0111	77
CAN	001 1000	18	8	011 1000	38	X	101 1000	58	x	111 1000	78
EM	001 1001	19	9	011 1001	39	Y	101 1001	59	y	111 1001	79
SUB	001 1010	1A	:	011 1010	3A	Z	101 1010	5A	z	111 1010	7A
ESC	001 1011	1B	;	011 1011	3B	[	101 1011	5B	{	111 1011	7B
FS	001 1100	1C	<	011 1100	3C	\	101 1100	5C	|	111 1100	7C
GS	001 1101	1D	=	011 1101	3D	]	101 1101	5D	}	111 1101	7D
RS	001 1110	1E	>	011 1110	3E	^	101 1110	5E	~	111 1110	7E
US	001 1111	1F	?	011 1111	3F	_	101 1111	5F	DEL	111 1111	7F

Abbreviations for Control Characters

NUL	null, or all zeros	**FF**	form feed	**CAN**	cancel
SOH	start of heading	**CR**	carriage return	**EM**	end of medium
STX	start of text	**SO**	shift out	**SUB**	substitute
ETX	end of text	**SI**	shift in	**ESC**	escape
EOT	end of transmission	**DLE**	data link escape	**FS**	file separator
ENQ	enquiry	**DC1**	device control 1	**GS**	group separator
ACK	acknowledge	**DC2**	device control 2	**RS**	record separator
BEL	bell	**DC3**	device control 3	**US**	unit separator
BS	backspace	**DC4**	device control 4	**SP**	space
HT	horizontal tabulation	**NAK**	negative acknowledge	**DEL**	delete
LF	line feed	**SYN**	synchronous idle		
VT	vertical tabulation	**ETB**	end of transmission block		

Example 3.32 The street address

```
52 Elm
```

would be stored in ASCII as

011 0101
011 0010
010 0000
100 0101
110 1100
110 1101

The blank space between `2` and `E` is a separate ASCII character. ■

The End of the Line

The ASCII standard was developed in the early 1960s and was intended for use on teleprinter machines of that era. A popular device that used the ASCII code was the Teletype Model 33, a mechanical printer with a continuous roll of paper that wrapped around a cylindrical carriage similar to a typewriter. The teleprinter received a stream of ASCII characters over a telephone line and printed the characters on paper.

The nonprintable characters are also known as *control characters* because they were originally used to control the mechanical aspects of a teleprinter. In particular, the ASCII LF control character stands for *line feed*. When the teleprinter received the LF character, it would rotate the carriage enough to advance the paper by one line. Another control character, CR, which stands for *carriage return*, would move the print head to the leftmost position of the page. Because these two mechanical operations were necessary to make the printer mechanism start at the beginning of a new line, the convention was to always use CR-LF to mark the beginning of a new line in a message that was to be sent to a teleprinter.

When early computer companies, notably Digital Equipment Corporation, adopted the ASCII code, they kept this CR-LF convention to denote the end of a line of text. It was convenient because many of those early machines used teleprinters as output devices. The convention was picked up by IBM and Microsoft when they developed the PC DOS and MS-DOS operating systems. MS-DOS eventually became Microsoft Windows, and the CR-LF convention has stuck to this day, despite the disappearance of the old teleprinter for which it was necessary.

Multics was an early operating system that was the forerunner of Unix. To simplify storage and processing of textual data, it adopted the convention of using only the LF character to denote the end of a line. This convention was picked up by Unix and continued by Linux, and is also the convention for OS X because it is Unix.

Figure 2.13 is a C program that reads a stream of characters from the input device and outputs the same stream of characters but substitutes the newline character, denoted `\n` in the string, for each space. The newline character corresponds to the ASCII LF control character. The program in Figure 2.13 works even if you run it in a Windows environment. The C standard specifies that if you output the `\n` character in a `printf()` string, the system will convert it to the convention for the operating system on which you are executing the program. In a Windows system, the `\n` character is converted to two characters, CR-LF, in the output stream. In a Unix system, it remains LF. If you ever need to process the CR character explicitly in a C program, you can write it as `\r`.

Unicode Characters

The first electronic computers were developed to perform mathematical calculations with numbers. Eventually, they processed textual data as well, and the ASCII code became a widespread standard for processing text with the Latin alphabet. As computer technology spread around the world, text processing in languages with different alphabets produced many incompatible systems. The Unicode Consortium was established to collect and catalog all the alphabets of all the spoken languages in the world, both current and ancient, as a first step toward a standard system for the worldwide interchange, processing, and display of texts in these natural languages.

Strictly speaking, the standard organizes characters into scripts, not languages. It is possible for one script to be used in multiple languages. For example, the extended Latin script can be used for many European and American languages. Version 7.0 of the Unicode standard has 123 scripts for natural language and 15 scripts for other symbols. Examples of natural language scripts are Balinese, Cherokee, Egyptian Hieroglyphs, Greek, Phoenician, and Thai. Examples of scripts for other symbols are Braille Patterns, Emoticons, Mathematical Symbols, and Musical Symbols.

Each character in every script has a unique identifying number, usually written in hexadecimal, and is called a *code point*. The hexadecimal number is preceded by "U+" to indicate that it is a Unicode code point. Corresponding to a code point is a *glyph*, which is the graphic representation of the symbol on the page or screen. For example, in the Hebrew script, the code point U+05D1 has the glyph ב.

FIGURE 3.26 shows some example code points and glyphs in the Unicode standard. The CJK Unified script is for the written languages of China, Japan,

Unicode Script	Code Point	Glyphs							
		0	1	2	3	4	5	6	7
Arabic	U+063_	ذ	ر	ز	س	ش	ص	ض	ط
Armenian	U+054_	Ձ	Ղ	Ճ	Մ	Յ	Ն	Շ	Ո
Braille Patterns	U+287_	⡠	⡡	⡢	⡣	⡤	⡥	⡦	⡧
CJK Unified	U+4EB_	京	亱	亲	亳	亴	亵	亶	亷
Cyrillic	U+041_	А	Б	В	Г	Д	Е	Ж	З
Egyptian Hieroglyphs	U+1300_	𓀀	𓀁	𓀂	𓀃	𓀄	𓀅	𓀆	𓀇
Emoticons	U+1F61_	😐	😑	😒	😓	😔	😕	😖	😗
Hebrew	U+05D_	א	ב	ג	ד	ה	ו	ז	ח
Basic Latin (ASCII)	U+004_	@	A	B	C	D	E	F	G
Latin-1 Supplement	U+00E_	à	á	â	ã	ä	å	æ	ç

FIGURE 3.26
A few code points and glyphs from the Unicode character set.

and Korea, which share a common character set with some variations. There are tens of thousands of characters in these Asian writing systems, all based on a common set of Han characters. To minimize unnecessary duplication, the Unicode Consortium merged the characters into a single set of unified characters. This Han unification is an ongoing process carried out by a group of experts from the Chinese-speaking countries, North and South Korea, Japan, Vietnam, and other countries.

Code points are backward compatible with ASCII. For example, from the ASCII table in Figure 3.25, the Latin letter S is stored with seven bits as 101 0011 (bin), which is 53 (hex). So, the Unicode code point for S is U+0053. The standard requires at least four hex digits following U+, padding the number with leading zeros if necessary.

A single code point can have more than one glyph. For example, an Arabic letter may be displayed with different glyphs depending on its position in a word. On the other hand, a single glyph might be used to represent two code points. The consecutive Latin code points U+0066 and U+0069 for f and i are frequently rendered with the ligature glyph fi.

The range of the Unicode code space is 0 to 10FFFF (hex), or 0 to 1 0000 1111 1111 1111 1111 (bin), or 0 to 1,114,111 (dec). About one-fourth of these million-plus code points have been assigned. Some values are reserved for private use, and each Unicode standard revision assigns a few more values to code points. It is theoretically possible to represent each code point with a single 21-bit number. Because computer memory is normally organized into eight-bit bytes, it would be possible to use three bytes to store each code point with the leading three bits unused.

However, most computers process information in chunks of either 32 bits (4 bytes) or 64 bits (8 bytes). It follows that the most effective method for processing textual information is to store each code point in a 32-bit cell, even though the leading 11 bits would be unused and always set to zeros. This method of encoding is called *UTF-32*, where *UTF* stands for *Unicode Transformation Format*. UTF-32 always requires eight hexadecimal characters to represent its four bytes.

Example 3.33 To determine how the letter z is stored in UTF-32, look up its value in the ASCII table as 7A (hex). Because Unicode code points are backward compatible with ASCII, the code point for the letter z is U+007A. The UTF-32 encoding in binary is obtained by prefixing zeros for a total of 32 bits as follows:

0000 0000 0000 0000 0000 0000 0111 1010

So, U+007A is encoded as 0000 007A (UTF-32). ∎

Example 3.34 To determine the UTF-32 encoding of the emoticon 😮 with code point U+1F617, simply prefix the correct number of zeros. The encoding is 0001 F617 (UTF-32). ∎

Although UTF-32 is effective for processing textual information, it is inefficient for storing and transmitting textual information. If you have a file that stores mostly ASCII characters, three-fourths of the file space will be occupied by zeros. UTF-8 is a popular encoding standard that is able to represent every Unicode character. It uses one to four bytes to store a single character and therefore takes less storage space than UTF-32. The 64 Ki code points in the range U+0000 to U+FFFF, known as the *Basic Multilingual Plane*, contain characters for almost all modern languages. UTF-8 can represent each of these code points with one to three bytes and uses only a single byte for an ASCII character.

FIGURE 3.27 shows the UTF-8 encoding scheme. The first column, labeled *Bits*, represents the upper limit of the number of bits in the code point, excluding all leading zeros. The x's in the code represent the rightmost bits from the code point, which are spread out over one to four bytes.

The first row in the table corresponds to the ASCII characters, which have an upper limit of seven bits. An ASCII character is stored as a single byte whose first bit is 0 and whose last seven bits are identical to seven-bit ASCII. The first step in decoding a UTF-8 string is to inspect the first bit of the first byte. If it is zero, the first character is an ASCII character, which can be determined from the ASCII table, and the following byte is the first byte of the next character.

If the first bit of the first byte is 1, the first character is outside the range U+0000 to U+007F—that is, it is not an ASCII character, and it occupies more than one byte. In this case, the number of leading 1's in the first byte

FIGURE 3.27
The UTF-8 encoding scheme.

Bits	First Code Point	Last Code Point	Byte 1	Byte 2	Byte 3	Byte 4
7	U+0000	U+007F	0xxxxxxx			
11	U+0080	U+07FF	110xxxxx	10xxxxxx		
16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
21	U+10000	U+1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

is equal to the total number of bytes occupied by the character. Some of the bits from the code point are stored in the first byte and some are stored in the remaining continuation bytes. Every continuation byte begins with the string 10 and stores six bits from the code point.

Example 3.35 To determine the UTF-8 encoding of the emoticon ☺ with code point U+1F617, first determine the upper limit of the number of bits in the code point. From Figure 3.27, it is in the range U+10000 to U+1FFFF; thus, the rightmost 21 bits from the code point are spread out over 4 bytes. The rightmost 21 bits from 1F617 (hex) are

0 0001 1111 0110 0001 0111

where enough leading zeros are added to total 21 bits. The last row in Figure 3.27 shows the first three bits stored in Byte 1, the next six stored in Byte 2, the next six stored in Byte 3, and the last six stored in Byte 4. Regrouping the 21 bits accordingly yields

000 011111 011000 010111

The format of Byte 1 from the table is 11110xxx, so insert the first three zeros in place of the x's to yield 11110000, and do the same for Bytes 3 and 4. The resulting bit pattern of the four bytes is

11110000 10011111 10011000 10010111

So, U+1F617 is encoded as F09F 9897 (UTF-8), which is different from the four bytes of the UTF-32 encoding in Example 3.34. ■

Example 3.36 To determine the sequence of code points from the UTF-8 byte sequence 70 C3 A6 6F 6E, first write the byte sequence in binary as

01110000 11000011 10100110 01101111 01101110

You can immediately determine that the first, fourth, and fifth bytes are ASCII characters because the leading bit is zero in those bytes. From the ASCII table, these bytes correspond to the letters p, o, and n, respectively. The leading 110 in the second byte indicates that 11 bits are spread out over 2 bytes per the second row in the body of the table in Figure 3.27. The leading 10 in the third byte is consistent, because that prefix denotes a continuation byte. Extracting the rightmost 5 bits from the second byte (first byte of the pair) and the rightmost 6 bytes from the third byte (second byte of the pair) yields the 11 bits:

00011 100110

Prefixing this pattern with leading zeros and regrouping yields

0000 0000 1110 0110

which is the code point U+00E6 corresponding to Unicode character æ. So, the original five-byte sequence is a UTF-8 encoding of the four code points U+0070, U+00E6, U+006F, U+006E and represents the string "pæon". ■

Figure 3.27 shows that UTF-8 does not allow all possible bit patterns. For example, it is illegal to have the bit pattern

11100011 01000001

in a UTF-8–encoded file because the 1110 prefix of the first byte indicates that it is the first byte of a three-byte sequence, but the leading zero of the second byte indicates that it is a single ASCII character and not a continuation byte. If such a pattern is detected in a UTF-8–encoded file, the data is corrupted.

A major benefit of UTF-8 is its self-synchronization property. A decoder can uniquely identify the type of any byte in the sequence by inspection of the prefix bits. For example, if the first two bits are 10, it is a continuation byte. Or, if the first four bits are 1110, it is the first byte of a three-byte sequence. This self-synchronization property makes it possible for a UTF-8 decoder to recover most of the text when data corruption does occur.

UTF-8 is by far the most common encoding standard on the World Wide Web. It has become the default standard for multilingual applications. Operating systems are incorporating UTF-8 so that documents and files can be named in the user's native language. Modern programming languages such as Python and Swift have UTF-8 built in so a programmer can, for example, name a variable pæon or even 😐😑. Text editors that have traditionally processed only pure ASCII text, as opposed to word processors that have always been format friendly, are increasingly able to process UTF-8–encoded text files.

3.5 Floating-Point Representation

The numeric representations described in previous sections of this chapter are for integer values. C has three numeric types that have fractional parts:

- `float` single-precision floating point
- `double` double-precision floating point
- `long double` extended-precision floating point

Values of these types cannot be stored at Level ISA3 with two's complement binary representation because provisions must be made for locating the decimal point within the number. Floating-point values are stored using a binary version of scientific notation.

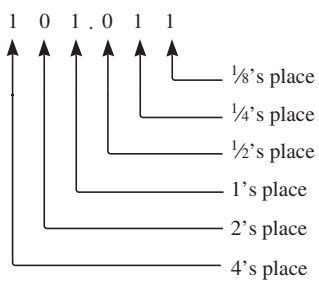
Binary Fractions

Binary fractions have a binary point, which is the base 2 version of the base 10 decimal point.

Example 3.37 FIGURE 3.28(a) shows the place values for 101.011 (bin). The bits to the left of the binary point have the same place values as the corresponding bits in unsigned binary representation, as in Figure 3.2. Starting with the 1/2's place to the right of the binary point, each place has a value one-half as great as the previous place value. Figure 3.28(b) shows the addition that produces the 5.375 (dec) value. ∎

FIGURE 3.29 shows the polynomial representation of numbers with fractional parts. The value of the bit to the left of the radix point is always the base to the zeroth power, which is always 1. The next significant place to the left is the base to the first power, which is the value of the base itself. The value of the bit to the right of the radix point is the base to the power −1. The next significant place to the right is the base to the power −2. The value of each place to the right is 1/base times the value of the place on its left.

FIGURE 3.28
Converting from binary to decimal.



(a) The place values for 101.011 (bin).

1 ⅛'s place = 0.125
1 ¼'s place = 0.25
0 ½'s place = 0.0
1 1's place = 1.0
0 2's place = 0.0
1 4's place = 4.0
5.375 (dec)

(b) Converting 101.011 (bin) to decimal.

FIGURE 3.29
The polynomial representation of floating-point numbers.

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

(a) The binary number 101.011.

$$5 \times 10^2 + 0 \times 10^1 + 6 \times 10^0 + 7 \times 10^{-1} + 2 \times 10^{-2} + 1 \times 10^{-3}$$

(b) The decimal number 506.721.

Determining the decimal value of a binary fraction requires two steps. First, convert the bits to the left of the binary point using the technique of Example 3.3 for converting unsigned binary values. Then, use the algorithm of successive doubling to convert the bits to the right of the binary point.

Example 3.38 FIGURE 3.30 shows the conversion of 6.5859375 (dec) to binary. The conversion of the whole part gives 110 (bin) to the left of the binary point. To convert the fractional part, write the digits to the right of the decimal point in the heading of the right column of the table. Double the fractional part, writing the digit to the left of the decimal point in the column on the left and the fractional part in the column on the right. The next time you double, do not include the whole number part. For example, the value 0.34375 comes from doubling 0.171875, not from doubling 1.171875. The digits on the left from top to bottom are the bits of the binary fractional part from left to right. So, 6.5859375 (dec) = 110.1001011 (bin). ∎

FIGURE 3.30
Converting from decimal to binary.

6.5859375

6 (dec) = 110 (bin)

(a) Convert the whole part.

	.5859375
1	.171875
0	.34375
0	.6875
1	.375
0	.75
1	.5
1	.0

(b) Convert the fractional part.

The algorithm for converting the fractional part from decimal to binary is the mirror image of the algorithm for converting the whole part, from decimal to binary. Figure 3.5 shows that to convert the whole part, you use the algorithm of successive division by two. The bits you generate are the remainders of the division, and you generate them from right to left starting at the binary point. To convert the fractional part, you use the algorithm of successive multiplication by two. The bits you generate are the whole part of the multiplication, and you generate them from left to right starting at the binary point.

A number that can be represented with a finite number of digits in decimal may require an endless representation in binary.

Example 3.39 FIGURE 3.31 shows the conversion of 0.2 (dec) to binary. The first doubling produces 0.4. A few more doublings produce 0.4 again. It is clear that the process will never terminate and that 0.2 (dec) = 0.001100110011 . . . (bin) with the bit pattern 0011 endlessly repeating. ∎

FIGURE 3.31
A decimal value with an unending binary representation.

	.2
0	.4
0	.8
1	.6
1	.2
0	.4
0	.8
1	.6
⋮	⋮

Because all computer cells can store only a finite number of bits, the value 0.2 (dec) cannot be stored exactly and must be approximated. You should realize that if you add 0.2 + 0.2 in a Level HOL6 language like C, you will probably not get 0.4 exactly because of the roundoff error inherent in the binary representation of the values. For that reason, good numeric software rarely tests two floating point numbers for strict equality. Instead, the software maintains a small but nonzero tolerance that represents how close two floating point values must be to be considered equal. If the tolerance is, say, 0.0001, then the numbers 1.38264 and 1.38267 would be

considered equal because their difference, which is 0.00003, is less than the tolerance.

Excess Representations

Floating-point numbers are represented with a binary version of the scientific notation common with decimal numbers. A nonzero number is normalized if it is written in scientific notation with the first nonzero digit immediately to the left of the radix point. The number zero cannot be normalized because it does not have a first nonzero digit.

Example 3.40 The decimal number −328.4 is written in normalized form in scientific notation as -3.284×10^2. The effect of the exponent 2 as the power of 10 is to shift the decimal point two places to the right. Similarly, the binary number −10101.101 is written in normalized form in scientific notation as -1.0101101×2^4. The effect of the exponent 4 as the power of 2 is to shift the binary point four places to the right. ▮

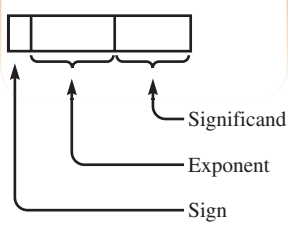
Example 3.41 The binary number 0.00101101 is written in normalized form in scientific notation as 1.01101×2^{-3}. The effect of the exponent −3 as the power of 2 is to shift the binary point three places to the left. ▮

In general, a floating point number can be positive or negative, and its exponent can be a positive or negative integer. FIGURE 3.32 shows a cell in memory that stores a floating point value. The cell is divided into three fields. The first field stores one bit for the sign of the number. The second field stores the bits representing the exponent of the normalized binary number. The third field, called the *significand*, stores bits that represent the magnitude of the value.

The more bits stored in the exponent, the wider the range of floating point values. The more bits stored in the significand, the higher the precision of the representation. A common representation is an 8-bit cell for the exponent and a 23-bit cell for the significand. To present the concepts of the floating point format, the examples in this section use a 3-bit cell for the exponent and a 4-bit cell for the significand. These are unrealistically tiny cell sizes, but they help to illustrate the format without an unwieldy number of bits.

Any signed representation for integers could be used to store the exponent. You might think that two's complement binary representation would be used, because that is the representation that most computers use to store signed integers. However, two's complement is not used. Instead, a biased representation is used for a reason that will be explained later.

FIGURE 3.32
Storage for a floating-point value.



An example of a biased representation for a five-bit cell is excess 15. The range of numbers for the cell is −15 to 16 as written in decimal and 00000 to 11111 as written in binary. To convert from decimal to excess 15, you add 15 to the decimal value and then convert to binary as you would an unsigned number. To convert from excess 15 to decimal, you write the decimal value as if it were an unsigned number and subtract 15 from it. In excess 15, the first bit denotes whether a value is positive or negative. But unlike two's complement representation, 1 signifies a positive value, and 0 signifies a negative value.

Example 3.42 For a five-bit cell, to convert 5 from decimal to excess 15, add 5 + 15 = 20. Then convert 20 to binary as if it were unsigned, 20 (dec) = 10100 (bin). Therefore, 5 (dec) = 10100 (excess 15). The first bit is 1, indicating a positive value. ∎

Example 3.43 To convert 00011 from excess 15 to decimal, convert 00011 as an unsigned value, 00011 (bin) = 3 (dec). Then subtract decimal values 3 − 15 = −12. So, 00011 (excess 15) = −12 (dec). ∎

FIGURE 3.33 shows the bit patterns for a three-bit cell that stores integers with excess 3 representation compared to two's complement representation. Each representation stores eight values. The excess 3 representation has a range of −3 to 4 (dec), while the two's complement representation has a range of −4 to 3 (dec).

Decimal	Excess 3	Two's Complement
−4		100
−3	000	101
−2	001	110
−1	010	111
0	011	000
1	100	001
2	101	010
3	110	011
4	111	

FIGURE 3.33
The signed integers for a three-bit cell.

The Hidden Bit

Figure 3.32 shows one bit reserved for the sign of the number but no bit reserved for the binary point. A bit for the binary point is unnecessary because numbers are stored normalized, so the system can assume that the first 1 is to the left of the binary point. Furthermore, because there will always be a 1 to the left of the binary point, there is no need to store the leading 1 at all. To store a decimal value, first convert it to binary, write it in normalized scientific notation, store the exponent in excess representation, drop the leading 1, and store the remaining bits of the magnitude in the significand. The bit that is assumed to be to the left of the binary point but that is not stored explicitly is called the *hidden bit*.

Example 3.44 Assuming a three-bit exponent using excess 3 and a four-bit significand, how is the number 3.375 stored? Converting the whole number part gives 3 (dec) = 11 (bin). Converting the fractional part gives 0.375 (dec) = 0.011. The complete binary number is 3.375 (dec) = 11.011 (bin), which is 1.1011×2^1 in normalized binary scientific notation. The number is positive, so the sign bit is 0. The exponent is 1 (dec) = 100 (excess 3) from Figure 3.33. Dropping the leading 1, the four bits to the right of the binary point are .1011. So, 3.375 is stored as 0100 1011. ■

Of course, the hidden bit is assumed, not ignored. When you read a decimal floating point value from memory, the compiler assumes that the hidden bit is not stored. It generates code to insert the hidden bit before it performs any computation with the full number of bits. The floating point hardware even adds a few extra bits of precision called *guard digits* that it carries throughout the computation. After the computation, the system discards the guard digits and the assumed hidden bit and stores as many bits to the right of the binary point as the significand will hold.

Not storing the leading 1 allows for greater precision. In the previous example, the bits for the magnitude are 1.1011. Using a hidden bit, you drop the leading 1 and store .1011 in the four-bit significand. In a representation without a hidden bit, you would store the most significant bits, 1.011, in the four-bit significand and be forced to discard the least significant 0.0001 value. The result would be a value that only approximates the decimal value 3.375.

Because every memory cell has a finite number of bits, approximations are unavoidable even with a hidden bit. The system approximates by rounding off the least significant bits it must discard using a rule called "round to nearest, ties to even." FIGURE 3.34 shows how the rule works for decimal and binary numbers. You round off 23.499 to 23 because 23.499 is closer to 23 than it is to 24. Similarly, 23.501 is closer to 24 than it is to 23. However, 23.5 is just as close to 23 as it is to 24, which is a tie. It rounds to 24 because 24

FIGURE 3.34
Round to nearest, ties to even.

Decimal	Decimal Rounded	Binary	Binary Rounded
23.499	23	1011.011	1011
23.5	24	1011.1	1100
23.501	24	1011.101	1100
24.499	24	1100.011	1100
24.5	24	1100.1	1100
24.501	25	1100.101	1101

is even. Similarly, the binary number 1011.1 is just as close to 1011 as it is to 1100, which is a tie. It rounds to 1100 because 1100 is even.

Example 3.45 Assuming a three-bit exponent using excess 3 and a four-bit significand, how is the number −13.75 stored? Converting the whole number part gives 13 (dec) = 1101 (bin). Converting the fractional part gives 0.75 (dec) = 0.11. The complete binary number is 13.75 (dec) = 1101.11 (bin), which is 1.10111×2^3 in normalized binary scientific notation. The number is negative, so the sign bit is 1. The exponent is 3 (dec) = 110 (excess 3). Dropping the leading 1, the five bits to the right of the binary point are .10111. However, only four bits can be stored in the significand. Furthermore, .10111 is just as close to .1011 as it is to .1100, and the tie rule is in effect. Because 1011 is odd and 1100 is even, round to .1100. So, −13.75 is stored as 1110 1100. ∎

Special Values

Some real values require special treatment. The most obvious is zero, which cannot be normalized because there is no 1 bit in its binary representation. You must set aside a special bit pattern for zero. Standard practice is to put all 0's in the exponent field and all 0's in the significand as well. What do you put for the sign? Most common is to have two representations for zero, one positive and one negative. For a three-bit exponent and four-bit significand, the bit patterns are

1 000 0000 (bin) = −0.0 (dec)
0 000 0000 (bin) = +0.0 (dec)

This solution for storing zero has ramifications for some other bit patterns. If the bit pattern for +0.0 were not special, then 0 000 0000 would be interpreted with the hidden bit as 1.0000×2^{-3} (bin) = 0.125, the smallest positive value that could be stored had the value not been reserved for zero. If this pattern is reserved for zero, then the smallest positive value that can be stored is

$$0\ 000\ 0001 = 1.0001 \times 2^{-3}\ (\text{bin}) = 0.1328125$$

which is slightly larger. The negative number with the smallest possible magnitude is identical but with a 1 in the sign bit. The largest positive number that can be stored is the bit pattern with the largest exponent and the largest significand. The bit pattern for the largest value is

$$0\ 111\ 1111\ (\text{bin}) = +31.0\ (\text{dec})$$

FIGURE 3.35 shows the number line for the representation where zero is the only special value. As with integer representations, there is a limit to how large a value you can store. If you try to multiply 9.5 times 12.0, both of which are in range, the true value is 114.0, which is in the positive overflow region.

Unlike integer values, however, the real number line has an underflow region. If you try to multiply 0.145 times 0.145, which are both in range, the true value is 0.021025, which is in the positive underflow region. The smallest positive value that can be stored is 0.132815.

Numeric calculations with approximate floating point values need to have results that are consistent with what would be expected when calculations are done with exact precision. For example, suppose you multiply 9.5 and 12.0. What should be stored for the result? Suppose you store the largest possible value, 31.0, as an approximation. Suppose further that this is an intermediate value in a longer computation. If you later need to compute half of the result, you will get 15.5, which is far from what the correct value would have been.

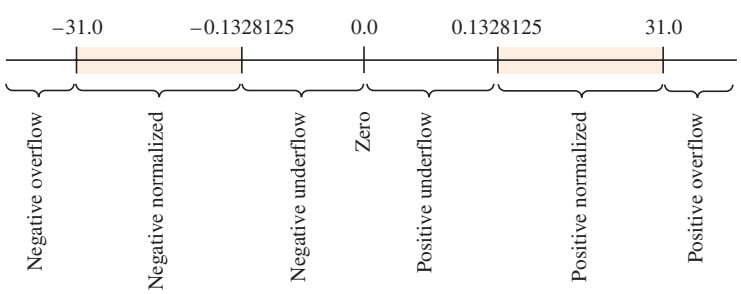


FIGURE 3.35
The real number line with zero as the only special value.

Special Value	Exponent	Significand
Zero	All zeros	All zeros
Denormalized	All zeros	Nonzero
Infinity	All ones	All zeros
Not a number	All ones	Nonzero

FIGURE 3.36
The special values in floating-point representation.

The same problem occurs in the underflow region. If you store 0.0 as an approximation of 0.021025, and you later want to multiply the value by 12.0, you will get 0.0. You risk being misled by what appears to be a reasonable value.

The problems encountered with overflow and underflow are alleviated somewhat by introducing more special values for the bit patterns. As is the case with zero, you must use some bit patterns that would otherwise be used to represent other values on the number line. In addition to zero, three special values are common—infinity, not a number (NaN), and denormalized numbers. FIGURE 3.36 lists the four special values for floating-point representation and their bit patterns.

Infinity

Infinity is used for values that are in the overflow regions. If the result of an operation overflows, the bit pattern for infinity is stored. If further operations are done on this bit pattern, the result is what you would expect for an infinite value. For example, $3/\infty = 0$, $5 + \infty = \infty$, and the square root of infinity is infinity. You can produce infinity by dividing by 0. For example, $3/0 = \infty$, and $-4/0 = -\infty$. If you ever do a computation with real numbers and get infinity, you know that an overflow occurred somewhere in your intermediate results.

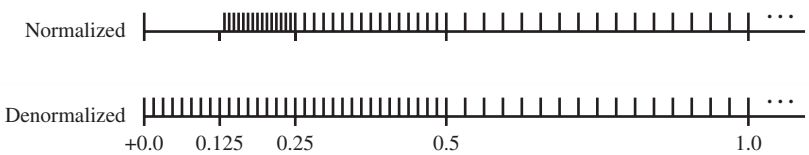
Not a number

A bit pattern for a value that is not a number is called a *NaN* (rhymes with *plan*). NaNs are used to indicate floating point operations that are illegal. For example, taking the square root of a negative number produces NaN, and so does dividing 0/0. Any floating point operation with at least one NaN operand produces NaN. For example, 7 + NaN = NaN, and 7/NaN = NaN.

Both infinity and NaN use the largest possible value of the exponent for their bit patterns. That is, the exponent field is all 1's. The significand is all 0's for infinity and can be any nonzero pattern for NaN. Reserving these bit patterns for infinity and NaN has the effect of reducing the range of values that can be stored. For a three-bit exponent and four-bit significand, the bit patterns for the largest magnitudes and their decimal values are

1 111 0000 (bin) = $-\infty$
1 110 1111 (bin) = −15.5 (dec)
0 110 1111 (bin) = +15.5 (dec)
0 111 0000 (bin) = $+\infty$

FIGURE 3.37
The real number line with and without denormalized numbers.



Denormalized numbers

There is no single infinitesimal value for the underflow region in Figure 3.35 that corresponds to the infinite value in the overflow region. Instead, there is a set of values called *denormalized values* that alleviate the problem of underflow. FIGURE 3.37 is a drawing to scale of the floating point values for a binary representation without denormalized special values (top) and with denormalized values (bottom) for a system with a three-bit exponent and four-bit significand. The figure shows three complete sequences of values for exponent fields of 000, 001, and 010 (excess 3), which represent −3, −2, and −1 (dec), respectively.

For normalized numbers in general, the gap between successive values doubles with each unit increase of the exponent. For example, in the number line on the top, the group of 16 values between 0.125 and 0.25 corresponds to numbers written in binary scientific notation with a multiplier of 2^{-3}. The 16 numbers between 0.25 and 0.5 are spaced twice as far apart and correspond to numbers written in binary scientific notation with a multiplier of 2^{-2}.

Without denormalized special values, the gap between +0.0 and the smallest positive value is excessive compared to the gaps in the smallest sequence. Denormalized special values make the gap between successive values for the first sequence equal to the gap between successive values for the second sequence. It spreads these values out evenly as they approach +0.0 from the right. On the left half of the number line, not shown in the figure, the negative values are spread out evenly as they approach −0.0 from the left.

Gradual underflow

This behavior of denormalized values is called *gradual underflow*. With gradual underflow, the gap between the smallest positive value and zero is reduced considerably. The idea is to take the nonzero values that would be stored with an exponent field of all 0's (in excess representation) and distribute them evenly in the underflow gap.

Because the exponent field of all 0's is reserved for denormalized numbers, the smallest positive normalized number becomes

$$0\ 001\ 0000 = 1.000 \times 2^{-2}\ (\text{bin}) = 0.25\ (\text{dec})$$

It might appear that we have made matters worse because the smallest positive normalized number with 000 in the exponent field is 0.1328125.

But, the denormalized values are spread throughout the gap in such a way as to actually reduce it.

When the exponent field is all 0's and the significand contains at least one 1, special rules apply to the representation. Assuming a three-bit exponent and a four-bit significand,

Representation rules for denormalized numbers

- The hidden bit to the left of the binary point is assumed to be 0 instead of 1.
- The exponent is assumed to be stored in excess 2 instead of excess 3.

Example 3.46 For a representation with a three-bit exponent and four-bit significand, what decimal value is represented by 0 000 0110? Because the exponent is all 0's and the significand contains at least one 1, the number is denormalized. Its exponent is 000 (excess 2) $= 0 - 2 = -2$ (dec), and its hidden bit is 0, so its binary scientific notation is 0.0110×2^{-2}. The exponent is in excess 2 instead of excess 3 because this is the special case of a denormalized number. Converting to decimal yields 0.09375. ∎

To see how much better the underflow gap is, compute the values having the smallest possible magnitudes, which are denormalized.

1 000 0001 (bin) = −0.015625 (dec)
1 000 0000 (bin) = −0.0
0 000 0000 (bin) = +0.0
0 000 0001 (bin) = +0.015625 (dec)

Without denormalized numbers, the smallest positive number is 0.1328125, so the gap has been reduced considerably.

FIGURE 3.38 shows some of the key values for a three-bit exponent and a four-bit significand using all four special values. The values are listed in numeric order from smallest to largest. The figure shows why an excess representation is common for floating point exponents. Consider all the positive numbers from +0.0 to +∞, ignoring the sign bit. You can see that if you treat the rightmost seven bits to be a simple unsigned integer, the successive values increase by one all the way from 000 0000 for 0 (dec) to 111 0000 for ∞. To do a comparison of two positive floating point values, say in a C statement like

```
if (x < y)
```

the computer does not need to extract the exponent field or insert the hidden bit. It can simply compare the rightmost seven bits as if they represented an integer to determine which floating point value has the larger magnitude. The circuitry for integer operations is considerably faster than that for

FIGURE 3.38
Floating-point values for a three-bit exponent and four-bit significand.

	Binary	Scientific Notation	Decimal
Not a number	1 111 nonzero		
Negative infinity	1 111 0000		$-\infty$
Negative normalized	1 110 1111 1 110 1110 . . . 1 011 0001 1 011 0000 1 010 1111 . . . 1 001 0001 1 001 0000	-1.1111×2^{3} -1.1110×2^{3} . . . -1.0001×2^{0} -1.0000×2^{0} -1.1111×2^{-1} . . . -1.0001×2^{-2} -1.0000×2^{-2}	-15.5 -15.0 . . . -1.0625 -1.0 -0.96875 . . . -0.265625 -0.25
Negative denormalized	1 000 1111 1 000 1110 . . . 1 000 0010 1 000 0001	-0.1111×2^{-2} -0.1110×2^{-2} . . . -0.0010×2^{-2} -0.0001×2^{-2}	-0.234375 -0.21875 . . . -0.03125 -0.015625
Negative zero	1 000 0000		-0.0
Positive zero	0 000 0000		$+0.0$
Positive denormalized	0 000 0001 0 000 0010 . . . 0 000 1110 0 000 1111	0.0001×2^{-2} 0.0010×2^{-2} . . . 0.1110×2^{-2} 0.1111×2^{-2}	0.015625 0.03125 . . . 0.21875 0.234375
Positive normalized	0 001 0000 0 001 0001 . . . 0 010 1111 0 011 0000 0 011 0001 . . . 0 110 1110 0 110 1111	1.0000×2^{-2} 1.0001×2^{-2} . . . 1.1111×2^{-1} 1.0000×2^{0} 1.0001×2^{0} . . . 1.1110×2^{3} 1.1111×2^{3}	0.25 0.265625 . . . 0.96875 1.0 1.0625 . . . 15.0 15.5
Positive infinity	0 111 0000		$+\infty$
Not a number	0 111 nonzero		

floating point operations, so using an excess representation for the exponent really improves performance.

The same pattern occurs for the negative numbers. The rightmost seven bits can be treated like an unsigned integer to compare magnitudes of the negative quantities. Floating-point quantities would not have this property if the exponents were stored using two's complement representation.

Figure 3.38 shows that −0.0 and +0.0 are distinct. At this low level of abstraction, negative zero is stored differently from positive zero. However, programmers at a higher level of abstraction expect the set of real number values to have only one zero, which is neither positive nor negative. For example, if the value of `x` has been computed as −0.0 vand `y` as +0.0, then the programmer should expect `x` to have the value 0 and `y` to have the value 0, and the expression `(x < y)` to be false. Computers must be programmed to return false in this special case, even though the bit patterns indicate that `x` is negative and `y` is positive. The system hides the fact that there are two representations of zero at a low level of abstraction from the programmer at a higher level of abstraction.

With denormalization, to convert from decimal to binary you must first check if a decimal value is in the denormalized range to determine its representation. From Figure 3.38, for a three-bit exponent and a four-bit significand, the smallest positive normalized value is 0.25. Any value less than 0.25 is stored with the denormalized format.

Example 3.47 For a representation with a three-bit exponent and four-bit significand, how is the decimal value −0.078 stored? Because 0.078 is less than 0.25, the representation is denormalized, the exponent is all zeros, and the hidden bit is 0. Converting to binary, 0.078 (dec) = 0.000100111.... Because the exponent is all zeros and the exponent is stored in excess 2 representation, the multiplier must be 2^{-2}. In binary scientific notation with a multiplier of 2^{-2}, 0.000100111 ... = 0.0100111 ... × 2^{-2}. As expected, the digit to the left of the binary point is 0, which is the hidden bit. The bits to be stored in the significand are the first four bits of .0100111 . . ., which rounds off to .0101. So the floating point representation for −0.078 is 1000 0101. ∎

The IEEE 754 Floating-Point Standard

The Institute of Electrical and Electronic Engineers, Inc. (IEEE), is a professional society supported by its members that provides services in various engineering fields, one of which is computer engineering. The society has various groups that propose standards for the industry. Before the IEEE proposed its standard for floating point numbers, every computer manufacturer designed its own representation for floating point values, and they all differed from each other. In the early days before networks became

prevalent and little data was shared between computers, this arrangement was tolerated.

Even without the widespread sharing of data, however, the lack of a standard hindered research and development in numerical computations. It was possible for two identical programs to run on two separate machines with the same input and produce different results because of the different approximations of the representations.

The IEEE set up a committee to propose a floating point standard, which it did in 1985. There are two standards: number 854, which is more applicable to handheld calculators than to other computing devices, and number 754, which was widely adopted for computers. The standard was revised with little change in 2008. Virtually every computer manufacturer now provides floating point numbers for their computers that conform to the IEEE 754 standard.

The floating point representation described earlier in this section is identical to the IEEE 754 standard except for the number of bits in the exponent field and in the significand. FIGURE 3.39 shows the two formats for the standard. The single-precision format has an 8-bit cell for the exponent using excess 127 representation (except for denormalized numbers, which use excess 126) and 23 bits for the significand. It corresponds to C type `float`. The double-precision format has an 11-bit cell for the exponent using excess 1023 representation (except for denormalized numbers, which use excess 1022) and a 52-bit cell for the significand. It corresponds to C type double.

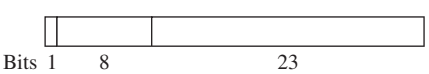
The single-precision format has the following bit values. Positive infinity is

0 1111 1111 000 0000 0000 0000 0000 0000

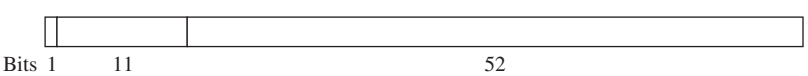
The hexadecimal abbreviation for the full 32-bit pattern arranges the bits into groups of four as

0111 1111 1000 0000 0000 0000 0000 0000

FIGURE 3.39
The IEEE 754 floating-point standard.



(a) Single precision.



(b) Double precision.

which is written 7F80 0000 (hex). The largest positive value is

0 1111 1110 111 1111 1111 1111 1111 1111

or 7F7F FFFF (hex). It is exactly $2^{128} - 2^{104}$, which is approximately 2^{128} or 3.4×10^{38}. The smallest positive normalized number is

0 0000 0001 000 0000 0000 0000 0000 0000

or 0080 0000 (hex). It is exactly 2^{-126}, which is approximately 1.2×10^{-38}. The smallest positive denormalized number is

0 0000 0000 000 0000 0000 0000 0000 0001

or 0000 0001 (hex). It is exactly 2^{-149}, which is approximately 1.4×10^{-45}.

Example 3.48 What is the hexadecimal representation of −47.25 in single-precision floating point? The integer 47 (dec) = 101111 (bin), and the fraction 0.25 (dec) = 0.01 (bin). So, 47.25 (dec) = 101111.01 = 1.0111101×2^5. The number is negative, so the first bit is 1. The exponent 5 is converted to excess 127 by adding 5 + 127 = 132 (dec) = 1000 0100 (excess 127). The significand stores the bits to the right of the binary point, 0111101. So, the bit pattern is

1 1000 0100 011 1101 0000 0000 0000 0000

which is C23D 0000 (hex). ∎

Example 3.49 What is the number, as written in binary scientific notation, whose hexadecimal representation is 3CC8 0000? The bit pattern is

0 0111 1001 100 1000 0000 0000 0000 0000

The sign bit is zero, so the number is positive. The exponent is 0111 1001 (excess 127) = 121 (unsigned) = 121 − 127 = −6 (dec). From the significand, the bits to the right of the binary point are 1001. The hidden bit is 1, so the number is 1.1001×2^{-6}. ∎

Example 3.50 What is the number, as written in binary scientific notation, whose hexadecimal representation is 0050 0000? The bit pattern is

0 0000 0000 101 0000 0000 0000 0000 0000

The sign bit is 0, so the number is positive. The exponent field is all 0's, so the number is denormalized. The exponent is 0000 0000 (excess 126) = 0 (unsigned) = 0 − 126 = −126 (dec). The hidden bit is 0 instead of 1, so the number is 0.101×2^{-126}. ∎

The double-precision format has both wider range and greater precision because of the larger exponent and significand fields. The largest double value is approximately 2^{1023}, or 1.8×10^{308}. The smallest positive normalized number is approximately 2.2×10^{-308}, and the smallest denormalized number is approximately 4.9×10^{-324}.

Figure 3.37, which shows the denormalized special values, applies to IEEE 754 values with a few modifications. For single precision, the exponent field has eight bits. Thus, the three sequences in the top figure correspond to multipliers of 2^{-127}, 2^{-126}, and 2^{-125}. Because the significand has 23 bits, each of the three sequences has $2^{23} = 8{,}388{,}608$ values instead of 16 values. It is still the case that the spacing between successive values in each sequence is double the spacing between successive values in the preceding sequence.

For double precision, the exponent field has 11 bits. So, the three sequences in the top figure correspond to multipliers of 2^{-1023}, 2^{-1022}, and 2^{-1021}. Because the significand has 52 bits, each of the three sequences has $2^{52} = 4{,}503{,}599{,}627{,}370{,}496$ values instead of 16 values. With denormalization, each of the 4.5 quadrillion values in the left group are spread out evenly as they approach +0.0 from the right.

3.6 Models

A model is a simplified representation of some physical system. Workers in every scientific discipline, including computer science, construct models and investigate their properties. Consider some models of the solar system that astronomers have constructed and investigated.

Aristotle, who lived in Greece about 350 BC, proposed a model in which the earth was at the center of the universe. Surrounding the earth were 55 celestial spheres. The sun, moon, planets, and stars were each carried around the heavens on one of these spheres.

How well did this model match reality? It was successful in explaining the appearance of the sky, which looks like the top half of a sphere. It was also successful in explaining the approximate motion of the planets. Aristotle's model was accepted as accurate for hundreds of years.

Then in 1543 the Polish astronomer Copernicus published *De Revolutionibus*. In it he modeled the solar system with the sun at the center. The planets revolved around the sun in circles. This model was a better approximation to the physical system than the earth-centered model.

In the latter part of the 16th century, the Danish astronomer Tycho Brahe made a series of precise astronomical observations that showed a discrepancy in Copernicus's model. Then in 1609 Johannes Kepler proposed a model in which the earth and all the planets revolved around the sun not in circles, but