2

Information Representation

One definition of a computer is a black box that manipulates information. First, information is entered into the computer. Then some **form** of processing is applied to the input information. Finally. the result is output to the user. In order to make any sort of evaluation of the computer or the proposed manipulation, some knowledge is required of **the** methods used for information storage and transfer. The purpose of this chapter is to examine the methods used for representing information. This includes not only numeric information, but also textual information, address representation,error coding information, boolean values, and status information. Each of these types of information is useful, and each type will be used by the computer at the appropriate time for a specific function. First, let us examine number representation, both integer and floating point, to determine the capabilities and limitations of available types of number systems. In addition, we will examine some of the difficulties **introduced** by numeric manipulation. Then we will move on to representation of status information, boolean information, and addresses. Finally we will consider the problems associated with integrating all of these types of information into the same system.

2.1. Integer Number Systems: Bounded Usefulness

Representation of information within a computer, and in most communication methods associated with computers, relies on the concept of a "bit." We will consider a bit to be a variable capable of assuming one of two distinct values. For numbers, these values are considered ones and zeros. Other interpretations are possible: true and false, asserted and **unasserted,** and so on. Collections of bits form numbers; each bit position doubles the possible representations of the system. Thus. the number of bits available for representation determines the number of representable values. For N bits, there are 2^N possible representations. Table 2.1 summarizes the number of representable values for popular computer sizes.

Table 2.1. Number of Representable Values.

Number of Bits	Number of Representable Values	Machines. Uses
4	16	4004, control
8	256	8080, 6800, control, communication
16	65.536	PDP11, 8086, 32020
32	4.29×10^9	IBM 370, 68020. VAX11/780
48	1.41×10^{14}	Unisys
64	1.84×10^{19}	Cray, IEEE (dp)

The number of bits used in a particular format identifies the total number of representable values, but does not directly specify the range of those values. The assumptions made about the representations actually identify the range and usefulness of the system. The simplest assumption is to let the binary numbers represent unsigned integers. If this is the case. then the range of representable numbers is from 0 to $2^N - 1$. These numbers are equally spaced. with a value of 1 between each representation. The system is a positional system, in every respect like the base 10 system with which we are familiar. Each bit position k has associated with it a value of 2^k, and the value represented by the collection of bits is represented by:

$$V_{\text{UNSIGNED INTEGER}} = \sum_{i=0}^{N-1} b_i \times 2^i$$

where b_i is the one or zero in position i. Thus, in unsigned binary the pattern 101101 means $1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45_{10}$.

While the unsigned integer representation is simple and easily manipulated, negative numbers cannot be represented. Hence, other integer systems are more often used for information representation. Perhaps the most widely utilized system is the two's complement system. Here, the 2^N representable values range from $-(2^{N-1})$ to $2^{N-1} - 1$. To negate a value, the value is subtracted from 2^N. Table 2.2 gives a few of the 256 values of an 8-bit two's complement system. This representation has also a positional nature, and the value of a panicular representation is given by:

$$V_{\text{TWO'S COMPLEMENT}} = -b_{N-1} \times 2^{N-1} + \sum_{i=0}^{N-2} b_i \times 2^i$$

Thus, in two's complement representation, the pattern 101101 means $-1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = -19$. One thing to note here is that, even though the most significant bit is not defined as a sign bit, it can be considered such. The reason for this is that, if the most significant bit is set. then the value will be negative, since the most significant bit carries more weight than all of the other bits combined.

Example 2.1: Finding values in two's complement number system: What is the bit representation of 87_{10} in an 8-bit two's complement number system?

There are a variety of algorithms for converting between bases; it is not our intention to promote one or another. And since this number is a positive number within the representable values of the system. the various

Table 2.2. 8-Bit Two's Complement Representations.

Bit Pattern	*Value*	*Note*
01111111	127	Largest representable value.
01111110	126	
01111101	125	
...	...	
...	...	
00000010	2	Note that leading zero indicates
00000001	1	positive number.
00000000	0	Unique representation of zero.
11111111	–1	Minus one is always all ones.
11111110	–2	Note that leading one indicates
11111101	–3	negative number.
...	...	
...	...	
10000010	–126	
10000001	–127	
10000000	–128	Smallest (most negative) representable value.

bit positions can each be checked to ascertain that the desired bit pattern is 01010111.

What is the bit representation of -76_{10} in an 8-bit two's complement number system?

Again, the solution begins by finding the bit pattern for 76_{10}, which is 01001100. To negate this, the number is then subtracted from 2^8:

$$\begin{array}{r} 1\ 00000000 \\ -\ 01001100 \\ \hline 10110100 \end{array}$$

It is not necessary to do these calculations in binary:

$$\begin{array}{r} 256 \\ -\ 76 \\ \hline 180 \end{array}$$

The representation 10110100 is equivalent (in unsigned binary) to 180. Also note that negative numbers are negated to positive numbers in exactly the same way:

What is the representation of the negative of 11010110_2?

Base two:

$$\begin{array}{r} 1\ 00000000 \\ -\ 11010110 \\ \hline 00101010 \end{array}$$

Base 10:

$$\begin{array}{r} 256 \\ -\ 214 \\ \hline 42 \end{array}$$

The answer 42 converts to 00101010, as above.

The last portion of the example demonstrates the method utilized by many people to arrive at the correct bit representation for negating a two's complement

number: complement all of the bits and add 1, which is the same as complement and increment. This also demonstrates the method for subtracting one number from another: the number to be subtracted is complemented and fed into one input of an adder, the other number forms the other adder input, and the carry in of the adder is asserted. The result is that a complement and increment have been performed on the number to be negated, and the result out of the adder will be the desired value.

One of the extremely attractive features of the two's complement system is its circular nature. This is graphically demonstrated in Figure 2.1 for a 4-bit two's complement system. The numbers are arranged around a circle from 1000 to 0111. As can be seen from the figure, progressing from one point to the next, or from one number to the next is accomplished by simply increasing or decreasing the values by one. When this happens at the 0111 to 1000 border, the number changes from a positive to a negative value. The net result is a discontinuity in the desired numeric sequence. The name given to this discontinuity is an overflow — we have exceeded our ability to represent information in the number system. The same thing will happen if you specify successively more negative numbers: decrementing 1001 to 1000 works fine, but decrementing 1000 results in 0111, which is a positive number. We have again crossed the discontinuity boundary, and exceeded our ability to represent information in the number system. When an arithmetic operation causes this to occur, many computers will respond by setting an "overflow bit." This bit can be included as one of the several bits making up the status word of a processor; these bits will be further described in the next chapter. In addition, the benefit of the circular nature of the two's complement system will be further discussed after consideration of a fractional representation of information.

The numbers to this point have been described as integers, which is the correct interpretation only if we make the proper assumptions concerning the placement of the radix point of the system. Unless otherwise stated, we naturally assume that the radix point is located directly to the right of the least significant bit. With this assumption the patterns do indeed represent integers, and all of the

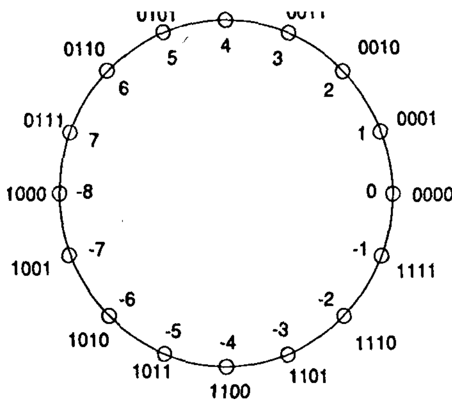

Figure 2.1. Graphical Representation of the Circular Nature of Two's Complement Numbers.

statements made concerning integer manipulation apply. However, if we assume that the radix point is located at some point other than to the right of the least significant bit position, then the range and granularity of the representable values changes.

The placement of the radix point (we are used to base 10, where it is the decimal point) is a matter of definition; no piece of hardware is installed on circuit boards to identify the location of the radix point. The radix point is established to satisfy the needs for which the processing element is utilized. If the information to be represented contains fractional values, then assumption of a radix point establishes a fixed point system that is so adjusted that it can cover the necessary range. Addition and subtraction operations for this type of a system are exactly the same as integer operations. However, for multiplication and division, care must be taken to assure that the radix point is in the correct place after an operation, and that the correct bits are saved. For example, multiplication of two 16-bit numbers, regardless of the placement of a radix point, results in a 32-bit number. However, if the result of the multiplication of two 16-bit numbers is to be stored in a 16-bit number, then there are limits to the size of the number. In the case of a fixed point system, the correct bits must be selected so that the assumptions made about the radix point of the multiplier and multiplicand are also true for the result.

A fixed point, noninteger system is also a positional system, just like the positional systems already described. The only difference is that the position of the radix point introduces a new factor into the equation. Let p represent the location of the radix point; this is the number of bit positions to the left of the least significant digit (bit) where the assumed radix point is found. Thus, the value of p for an integer system would be zero. Then the equation for the value of a two's complement fixed point number would be given by:

$$V_{\text{FIXED POINT}} = -b_{N-1} \times 2^{N-p-1} + \sum_{i=0}^{N-2} b_i \times 2^{i-p}$$

This gives the user the flexibility to choose a system that will fulfill the needs of a specific project. That is, a designer can determine the smallest value required to meet the needs of the system, and select a number system accordingly. Therefore, one of the characteristic values of a number system that will help determine its usefulness is the difference (in absolute value) between adjacent numeric representations. We will call this difference Ar. Note that Δr for all integer systems is 1; Δr for fixed point systems will be 2^{-p}.

A simple example of this is a 5-digit decimal number system for representation of monetary values. If p is equal to **0,** then the system can represent values from $0 to $99999, and Δr has a value of **$1.** Thus, any value less than a dollar cannot be represented in the system. If the system designer needs to represent cents as well as dollars, then p can be assumed to have a value of 2. The five digits can then represent values from $0.00 to $999.99, and Δr has a value of $0.01. Both systems have the same number of representable values, but the range and the Δr differ with the use of the system and the assumed value for p.

Example 2.2: Fixed point number system: Consider a fixed point 16-bit two's complement system with a value of p equal to 8. What is the smallest representable number? What is the largest representable number? What is the Ar? We know that addition and subtraction will not have any effect on

the position of the radix point, but what is the correct procedure for selecting the bits to retain after a multiplication?

The smallest representable number can be defined to be either the smallest absolute value that can be represented or the most negative number. The smallest absolute value for this system is zero; the next smallest absolute value is represented by the bit pattem 0000000.00000001. This has the value of 2^{-8}, which is just 3.9 x 10^{-3}. The most negative representable number has the bit pattern 10000000.00000000; this has the value of -2^7, which is –128. The Δr for this system is the same as the smallest representable value, 2^{-8}. To ascertain the effect that multiplication has upon the radix point, notice what happens when we multiply two of these numbers: the least significant bit will represent 2^{-16}, while the most significant bit will be 2^{14}. Thus, to get a properly aligned value when the process is over, the 31-bit result must be right shifted 8 bits, and then the next 16 bits form the desired result. Note that this dictates that the multiplication of the two input numbers has a maximum value if the number of bits saved as a result is to be the same as the number of bits used for the inputs. If the number of bits required to represent the value of the result exceeds 16, then an overflow has occurred.

A fixed point system is often used in applications like digital signal processing (DSP), where the values are scaled as they enter the system, and the intermediate values are sufficiently represented by the number of bits in the system. The Fujitsu 8764 DSP chip uses a 16-bit value with p equal to 14, while some components, such as the AMD29517, are set up for a 16-bit system with p equal to 15. If p is equal to 15 then all representable values (except for –1.0) have an absolute value less than one, and the system scales easily. These systems utilize integer arithmetic units, which are faster and require fewer devices than their floating point counterparts.

Some applications, such as the digital signal processing applications mentioned above, are able to take full advantage of the circular nature of the two's complement number system. These applications have a characteristic, inherent in the application itself, that will permit the number system to cross the overflow boundary without causing a disruption in the overall flow of instructions and data. For example, one of the frequently used digital signal processing algorithms is a finite impulse response (FIR) filter function. It can be shown that certain FIR filters will always result in a value within the ability of the number system to represent information; hence, intermediate overflows in the addition process can be ignored.

The two's complement number system is the most widely used integer system in machines, but it is not the only one. Another method used in some machines is the one's complement system. Mathematically, the one's complement of an N-bit number with value V is defined as $2^N - 1 - V$. The $2^N - 1$ portion of the equation is merely N one's, and subtracting V from a pattern of all 1's results in zeros where the ones were, and ones where the zeros were. Hence, the negative of a number is formed by complementing all of the bits in the number. Therefore, to negate any number, all that is required is to invert every bit position, which is a very fast operation. The range covered by this method is from $2^{N-2} - 1$ to $-(2^{N-2} - 1)$, which is just one different from the range of the two's complement number system. However, some "features" of this system limit its usefulness in digital systems. Unlike the other systems discussed above, this system

does not follow a positional notation methodology. The bits have different significance depending on the sign of the number. Also, this system has two legal representations for the number zero, both of which must be checked by any operation that tests for zero. Finally, treatment of the cany in this system is different from other systems, because of its "end-around" feature. The proof of this feature will be left as an exercise, but the effect can be seen from the following example:

Example 2.3: One's complement arithmetic: Consider ***a*** 6-bit one's complement system. Represent 15. –15. 13. and –13 in this system. Then **perform** the following additions: 15 **+** 13, 15 **+ (–13),** 13 **+ (–13),** and 13 **+** (–15).

The numbers are derived in a simple fashion:

Decimal Value	*One's Complement*	*Comment*
15	001111	Positive numbers same as two's complement.
– 15	110000	Complement bits to negate.
13	001101	Positive numbers same as two's complement.
– 13	110010	Complement bits to negate.
Now for the additions:		
15	001111	This proceeds just like the two's
+ 13	+ 001101	complement version.
28	0 011100	No cany out: number is correct, and the result is as we expect.
15	001111	The addition is done in the normal
+ –13	+ 110010	fashion, but
2	1 000001	the result of one is incorrect; however, the presence of a carry says we
	+ 000001	should add that as a 1 in the LSB
	000010	which gives the expected result.
13	001101	This time we will add a positive number
+ –13	+ 110010	to its negative (which is just complement)
0	111111	and end up with all ones — a valid zero.
13	001101	Here the positive number is smaller
+ –15	+ 110000	than the negative number, so result
–2	0 111101	is negative; no cany — the value is correct.

Note that, in all of the above cases, the carry out can be added to the intermediate result (hence the name of end-around carry) to produce the correct final result.

The one's complement number system can be used in many of the same ways that other systems can be used, but care must be. taken to operate within the constraints that it imposes.

Another system utilized to represent numbers is the excess system. Here an excess is purposely added to the value to be represented, and the resulting bit pattern is stored or used as required. One of the most prevalent uses of excess codes is to store exponents in floating point numbers. If we let S represent the value that

will be stored or otherwise utilized. V the true value of the number, and E the excess, then the relationship between them is defined as:

$$S = V + E$$

In operations utilizing this type of representation care must be taken to be sure that the result is within the desired range. That is, if two numbers are added together, the following will happen:

$$S1 + S2 = (V_1 + E) + (V_2 + E)$$

$$= (V_1 + V)_2 + 2 \times E$$

To obtain the correct result $[(V_1 + V_2) + E]$, a value equal to E must be removed from the calculated result. In some systems, where E is a power of 2, this is a simple operation. However, in other systems the operation can become more complicated.

Example 2.4: Number representation in excess codes: What is the representation of $+37_{10}$ in an 8-bit excess *128* code? What is the representation of -23_{10} in an 8-bit excess 128 code? What is the sum of the two numbers, in the 8-bit excess *128* code?

An 8-bit unsigned number can represent values between 0 and *255.* The excess representation can then represent values from –128 to +127.

+ 128	10000000	This is the excess.
+ 37	00100101	The value to be represented.
165	10100101	The representation of 37_{10} in excess 128 code.
+ 128	10000000	This is the excess.
– 23	00010111	The value to be represented.
105	01101001	The representation of -23_{10} in excess **128** code.
165	10100101	This is +**37** in excess **128.**
+ 105	+ 01101001	This is **–23** in excess 128.
270	1 00001110	Note the carry out in this operation. **270** is too big to represent in **8** bits; to correct for the 2 **x** E that is in this sum, subtract **128.**
– 128	– 10000000	In binary, is this add or subtract?
142	10001110	This is the representation of **14,** the correct result. in excess **128.**

Another use for the excess code is in representing decimal numbers. A 4 bit integer representation can assume values between 0 and *15.* If we limit ourselves to decimal numbers, the desired values are 0 to 9. These are represented in excess *3* by the numbers *3* to *12.* One of the beneficial effects of this type of representation is that, when two numbers are added together by a 4-bit binary adder, if the addition of those decimal values would have resulted in a carry out, then there will be a carry out of the binary adder. Note that if D_1 and D_2 *are* decimal numbers represented by $V_1 = D_1 + 3$ and $V_2 = D_2 + 3$, then $V_1 + V_2 = D_1 + D_2 + 6$. Then if the sum of D_1 and D_2 would result in a value greater than 9, which would cause a carry in a decimal adder, $V_1 + V_2$ would cause a carry in

a binary adder. Note also that the resulting value $(D_1 + D_2 + 6)$ must have 3 removed from it before it is the valid representation in excess 3 for the resulting number, assuming that no carry out resulted. If a carry out did result, then the 4-bit representation is actually the correct value, since the excess is 6, and there are 6 unused representations in the 4-bit scheme. This code can be very useful for systems that work with 4-bit quantities.

Example 2.5: BCD excess 3 system: Consider *a* system that works with 3-digit decimal numbers, and it stores the digits in excess 3 format. What is the representation of 573? What is the representation of 142? Add the two numbers, and give the correct result in excess 3 format.

The numbers are handled on a digit-by-digit basis, with the excess being included with each digit:

Decimal	*Binary*	
+3 3 3	0011 0011 0011	This is the excess.
5 7 3	0101 0111 0011	And the number to be represented.
8 10 6	1000 1010 0110	The excess 3 representation.
+3 3 3	0011 0011 0011	This is the excess.
1 4 2	0001 0100 0010	This is the number to be represented.
4 7 5	0100 0111 0101	The excess 3 representation.
573	1000 1010 0110	Do the addition in decimal and in
+ 142	0100 0111 0101	binary. Correct as needed to make output correct.
715	1100 1 0001 1011	
		Carry out of second set of 4 bits indicates that the most significant digit should be incremented by one. Also indicates that this value is correct as it stands (since 2 × E = 6 and the carry out indicates that the number overflowed into the next digit) so we need to add 3. Therefore, the MSD needs to be incremented by one and decremented by 3; the middle digit needs to have 3 added; and the LSD needs to be decremented by 3.
	− 0010 + 0011 − 0011	
	1010 0100 1000	Which is the correct excess 3 representation for 715_{10}.

The excess representation for decimal numbers does have some useful characteristics, but usually this information is represented in the more natural binary coded decimal format (BCD). This code is listed in Table 2.3. Here, the numbers 0–9 are represented by the equivalent binary representations 0000 to 1001. Thus, 4 bits are used for each decimal number. To represent all of the decimal numbers from 0 to 99_{10}, 8 bits would be required. The smallest number (zero) would be represented as 00000000; the largest would be 10011001. But we already know that with 8 bits we should be able to represent 256 values; why are we limited to 100 values with BCD? The other representations (1010 to 1111) are not used with BCD, which does not fully utilize all of the representable values. Thus, a BCD system is capable of $10^{N/4}$ different representations.

Table 2.3. Binary Coded Decimal (RCD) Representations.

Bit Pattern	*Value*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	Not valid
1011	Not valid
1100	Not valid
1101	Not valid
1110	Not valid
1111	Not valid

The above representations point out some very important characteristics of the representation of information with bit patterns:

- For a representation of N bits, 2^N different values can be represented. This is true whether the information represented is numerical in nature, an address, or any other information, such as an instruction.
- The meaning associated with a bit pattern depends on the assumptions made.
- The assumptions about bit meaning will impact on the design of the hardware that manipulates the bits. Note that a one's complement adder is a different piece of hardware than a two's complement adder.
- The Δr for each representation is the same for the entire range of representable values.
- The assumption of a radix point will allow representation of fractional values. Note that the assumption of a radix point does not impact on the addition process; however, multiplication and division must account for shifts in the radix point due to an increased number of bits in the result.
- The choice of a coding method is based on available hardware, the desired range of values, and other system goals.

Regardless of the mechanisms chosen for information representation, the resulting collection of bits must be represented and communicated. Internal to a machine, the data is just that: a collection of bits in a register, in a memory location, or on a bus. But how humans see and remember or communicate this information is not usually in bit pattern format. We generally group bits together and utilize a different base to represent them; the most common systems are octal and hexadecimal. These representations will be utilized as appropriate throughout this book.

Example 2.6: Alternate representations for bit patterns: Represent -157_{10} and $+25{,}477_{10}$ in binary, octal, and hexadecimal. Do this for the 16-bit two's complement representation and a 16-bit excess 32,768 representation.

Two's Complement

+157	= 0000000010011101	The binary representation. to 16 bits.
−157	= 1111111101100011	To negate, complement and increment.
	= 1 111 111 101 100 011	Group in groups of 3 for octal.
	= 177543	
	= 1111 1111 0110 0011	Group in groups of 4 for hexadecimal.
	= FF63	
25.477	= 0110001110000101	the binary representation. to 16 bits.
	= 0 110 001 110 000 101	Group in groups of 3 for octal.
	= 061605	
	= 0110 0011 1000 0101	Group in groups of 4 for hexadecimal.
	= 6385	

Excess 32,768

32,611	= 0111111101100011	Excess code for −157.
	= 0 111 111 101 100 011	Group for octal.
	= 077543	Compare with two's complement.
	= 0111 1111 0110 0011	Group for hexadecimal.
	= 7F63	Compare with two's complement.
58,245	= 1110001110000101	Excess code for 25477.
	= 1 110 001 110 000 101	Group for octal.
	= 161605	Compare with two's complement.
	= 1110 0011 1000 0101	Group for hexadecimal.
	= E385	Compare with two's complement.

2.2. Fioating Point Number Systems: Coding for Range

The previous section pointed out the fact that for an N-bit number, there are 2^N different representable values. If we assume an integer interpretation to the bit pattern, then we have a numerical range of 2^N. Throughout this range Δr is equal to one. The coding mechanism (two's complement, one's complement, excess code, etc.) identifies the low point and the high point of that range. If we assume a radix point within the word, then the range is smaller; however, now we have the ability to represent fractional values. Many problems require the ability to represent information of a much greater or smaller magnitude than possible with fixed point systems, and for these problems we need a different type of information representation system. We are familiar with the use of scientific notation to represent large numbers, such as Avogadro's number (6.022×10^{23}), or small numbers, such as the mass of a proton (1.673×10^{-24} g). This same scheme is used to represent large and small numbers in computers, and has the name of a floating point number system (FPNS). This type of number system does not expand the quantity of representable values; rather, it modifies the way in which the 2^N values is interpreted.

To specify a floating point number, seven different pieces of information are necessary: base of the system, sign, magnitude, and base of the mantissa, and the sign, magnitude, and base of the exponent. We will first look at scientific notation, which is used to identify these pieces, and then examine methods used in computers to do the same things. The numbers in scientific notation above have the following format:

$$\text{(Sign) Mantissa} \times \text{Base}^{\text{EXPONENT}}$$

The "base" in the above equation is the radix of the system. For "normal" scientific numbers this radix is 10, because that is the base of the number system with which we are most familiar. Most computers do arithmetic in a binary fashion, so this choice would not be advantageous for a computer. The radix of the system is a constant that is decided at the time the system is defined, and it has a direct bearing on the range of values that the system can represent, as we shall see. The value used for the radix is not stored in the computer, but forms part of the definition of the number system. We will denote the radix of the system as r_b.

The radix of the system also applies to the mantissa. The mantissa is used to identify the significant digits of a value. In practice, we may use mantissas with few digits or many digits. In a machine representation, the number of digits used for mantissa representation is the same for all numbers (of the same type, i.e., single precision or double precision). One of the characteristics of the floating point number is the number of digits used to represent the mantissa. This number will be identified simply as m. Thus, for a specific floating point number system, each mantissa will consist of m base r_b digits. Let us designate the value of a mantissa as V_M. In the consideration of the range of the system, we will need to know the maximum and minimum allowable values for the mantissa, which we will designate as $V_{M_{MAX}}$ and $V_{M_{MIN}}$.

The location of the value of a floating point number on the real number line will be determined by the exponent. If the exponent is a large positive number, then the value of the floating point number is very large. If the exponent is zero then the value of the floating point number is just the value of the mantissa. If the exponent is a large negative number, then the value of the floating point number is very small. Determining the value of the exponent requires information concerning the sign, the radix. and the number of digits in the representation. We will let the radix of the exponent be designated by r_e. Like the radix of the system, the choice of r_e is made at design time, and is part of the number definition. For the scientific number examples above, $r_b = r_e = 10$, but in most computers, $r_e = 2$.

The number of digits in the exponent specifies the maximum size of the exponent, which, in conjunction with the radix of the system, identifies the range of the number system. We will designate this number (the number of digits in the exponent) with the letter e. Note that the exponent will contain e base r_e digits, and that, like the mantissa, r_e and e are decisions made at the time the number system is defined.

The sign of the exponent also needs to be identified. For our scientific examples, this was directly identified by the presence or absence of the minus sign. It is possible to do the same for exponents stored in floating point numbers in computers: identify a bit that is a sign bit for the exponent, and let the exponent be stored in sign-magnitude format. However, most computers use not this method, but rather a coding technique to represent positive and negative values. The method most often used is the excess code technique, although other methods could be used as well. We will examine the reason for excess codes in the exponent a little later. Whatever the coding scheme chosen, each of the allowable representations for the exponent results in a unique value for the exponent. Let the value of the exponent be represented as V_E. As with the mantissa, we will

need to know the maximum and minimum representable values of the exponent. We will designate these as $V_{E_{MAX}}$ and $V_{E_{MIN}}$.

The sign of the number itself must also be known. In the scientific representation above it is identified explicitly by a sign. This sign-magnitude mechanism is also the most prevalent mechanism for the storage of floating point numbers in computers. However, this information may be coded into the number by any of the coding schemes which allow for positive and negative representations.

The final piece of information we need is the placement of the radix point. In scientific notation, we explicitly designate the location of the radix point. However, in the machine we will need to make provisions for identifying the location of the radix point, or make appropriate assumptions about the system at design time. As with the fractional systems discussed above, let p designate the location of the radix point with respect to the least significant digit. This p will be used in determining the value of the mantissa, since, for an unsigned mantissa.

$$V_M = \sum_{i=0}^{N-1} d_i \times r_b^{i-p}$$

where the mantissa is composed of N r_b digits labeled d_{N-1} to d_0. So, the value! of the floating point number. V_{FPN}, is given by

$$V_{FPN} = (-1)^{SIGN} V_M \times r_b^{V_E}$$

The location of the radix point of the mantissa is directly connected to the value of the exponent. Consider the following representations for the number $32{,}768_{10}$.

$$3.2768 \times 10^4 = 32.768 \times 10^3 = 3276.8 \times 10^1$$

Each of the representations is a correct number in scientific notation, and the location of the decimal point is reflected in the value of the exponent. If the location of the radix point within the word is allowed to vary from number to number, then provisions must be made to record p and use that information in all of the calculations. This could be confusing and cumbersome, so in most systems an assumption is made concerning the location of the radix point (that is, the value of p) to minimize the amount of stored information and to make the arithmetic easier. The process of representing all of the numbers such that the mantissas all have the same value for p is called **normalization.** This process also identifies the allowable mantissa values. The assumption that we will make for our examples is that p = M, and that the **leftmost** digit of the mantissa is nonzero. This means that the mantissa is a fraction that can have values between $1/r_b$ and almost **1.** Specifically, the maximum value is $V_{M_{MAX}} = 0.d_m d_m d_m \ldots$ to the length of the mantissa, where $d_M = r_b - 1$; this number is very close to one ($1 - r_b^{-N}$, for N digits). And the minimum mantissa value is $V_{M_{MIN}} = 0.100 \ldots = 1/r_b$. The value of this number varies with each r_b. Thus, the only legal values for the mantissas vary from $V_{M_{MIN}}$ to $V_{M_{MAX}}$, and the numbers represented by the FPNS must be obtained by combining a member of this set of mantissas and one of the available exponents. This leads to the following observations concerning nonzero values in a normalized floating point number system:

$$\text{Maximum representable value} = V_{FPN_{MAX}} = V_{M_{MAX}} \times r_b^{V_{E_{MAX}}}$$

$$\text{Minimum representable value} = V_{FPN_{MIN}} = V_{M_{MIN}} \times r_b^{V_{E_{MIN}}}$$

$$\text{Number of legal mantissas} = NLM_{FPN}$$

$$= (r_b - 1) \times r_b^{m-1}$$

$$\text{Number of representable values} = NRV_{FPN}$$

$$= \text{Number of legal mantissas}$$

$$\times \text{ Number of legal exponents}$$

These values help to identify the characteristics of a floating point number system, and are useful to determine if the system can be used in a specific application.

Example 2.7: Characteristics of a FPNS: Consider a normalized floating point number system which has $r_b = 10$, $r_e = 10$, $m = 3$, $e = 2$, both exponent and number itself stored in sign/magnitude format. What is the largest representable fraction? What is the smallest representable fraction? What is the largest representable exponent? What is the smallest representable exponent? What is the largest representable number? What is the smallest representable positive nonzero number? How many nonzero numbers can be represented in this system?

From the equations given above,

$$V_{M_{MAX}} = 0.999 = 1.000 - 10^{-3}$$

$$V_{M_{MIN}} = 0.100$$

$$V_{E_{MAX}} = 99$$

$$V_{E_{MIN}} = -99$$

Note that this is the most negative exponent. The smallest exponent in absolute value is 0, but that exponent does not lead to the smallest representable numbers.

$$V_{FPN_{MAX}} = 0.999 \times 10^{99}$$

$$V_{FPN_{MIN}} = 0.100 \times 10^{-99}$$

$$NLM_{FPN} = 9 \times 10 \times 10$$

$$= 900$$

$$NRV_{FPN} = 2 \times 900 \times 199$$

$$= 358{,}200$$

There are 199 representable exponents: 99 greater than zero, 99 less **than** zero, and zero. At this point we will make an observation concerning the number system, but leave the discussion of the problem until a later section. The number system can represent very large values, and very small values. The question is, can the value 1.00 be added to **10,000?** The representation **of 1.00** is 0.100×10^1. **The** representation **of 10,000** is 0.100×10^5. **But** there is no legal representation for 10,001. This points out the fact that the Ar **is** different for each V_E. For the legal representation of 1.00, the Ar is 1/100. For the representation of 10,000 the Δr is **100.**

The above example provides a very interesting illustration. With five digits and two sign bits 358,200 different values can be represented with a **normalized** floating point system. If we assume that one of the sign bits can be interpreted as a number with a value of 0 or 1, then an integer number system with the same quantity of symbols **(i.e.,** five base 10 digits, a base 2 digit, and a sign) would be capable of representing 399,998 values, more than the floating point system. But these numbers vary between –199,999 and 199,999. (Note that we are not counting the representation of zero in either case.)

The range of the system and the number of representable fractions are both affected by the choice of the base of the system. The example given above used base 10, but computers don't usually provide that capability. With the base 2 arithmetic capabilities of the machines, one would **assume** that the most natural base for floating point numbers would be two. However, grouping the bits into other base values, such as base 4 or base 8, can expand the range of the system. To demonstrate this let us compare two different normalized floating point number systems, simplified enough that we can enumerate all of the legal values in the systems.

Both of these systems are representable in six bits. The first system is enumerated in Table 2.4. This table identifies how each bit is used in the number system. Four bits are used in the mantissa, and two bits for the exponent. With the $r_b = 2$ for this system, $m = 4$. Also, $r_e = 2$, and $e = 2$. Missing from the number system are negative values, both for the exponent and for the number itself. Nevertheless, the system demonstrates some important points. First of all, the first bit in the mantissa gives no information. In our definition of a normalized FPNS, the first digit to the right of the radix point must be nonzero, and for a base 2 system, the only digit left is a one. Hence, this digit adds no information to the system. Second, the Δr changes for each value of the exponent: when the exponent is 0, the Ar is 1/16; when the exponent is **1,** the Ar is **1/8.** The Δr doubles for successive exponent values. The third observation is that the 32 values representable in this system is only half of the $2^6 = 64$ legal combinations of six bits. Thus, while an integer system would represent 64 equally spaced values (0 to 63). the system demonstrated in Table 2.4 represents 32 nonequally spaced values from $1/2$ to $7\,1/2$.

In contrast to the system of Table 2.4 is another 6-bit normalized floating point system shown in Table 2.5. This system is constructed so that the bits are grouped into base 4 digits; thus, the permissible values for the first digit are **1,** 2, and 3, all of which take two bits to represent. However, note that the digit 1_4 has a leading 0 ($1_4 = 01_2$). This increases the number of allowable mantissas from eight to twelve. Thus, the first bit of the mantissa in this representation is ***not*** redundant, as it was for the previous system. The Δr for this system varies by a factor of four for subsequent exponent values. Note that the Δr for $V_E = 0$ is the

Table 2.4. 6-Bit Normalized Floating Point System, Base 2.
$r_b = 2,\ r_e = 2,\ m = 4,\ e = 2$

				$V_E \rightarrow$	00	01	10	11
				$2^{V_E} \rightarrow$	1	2	4	8
V_M base 2				V_M	$V_M \times 2^{V_E}$			
1	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	1	2	4
1	0	0	1	$\frac{9}{16}$	$\frac{9}{16}$	$1\frac{1}{8}$	$2\frac{1}{4}$	$4\frac{1}{2}$
1	0	1	0	$\frac{5}{8}$	$\frac{5}{8}$	$1\frac{1}{4}$	$2\frac{1}{2}$	5
1	0	1	1	$\frac{11}{16}$	$\frac{11}{16}$	$1\frac{3}{8}$	$2\frac{3}{4}$	$5\frac{1}{2}$
1	1	0	0	$\frac{3}{4}$	$\frac{3}{4}$	$1\frac{1}{2}$	3	6
1	1	0	1	$\frac{13}{16}$	$\frac{13}{16}$	$1\frac{5}{8}$	$3\frac{1}{4}$	$6\frac{1}{2}$
1	1	1	0	$\frac{7}{8}$	$\frac{7}{8}$	$1\frac{3}{4}$	$3\frac{1}{2}$	7
1	1	1	1	$\frac{15}{16}$	$\frac{15}{16}$	$1\frac{7}{8}$	$3\frac{3}{4}$	$7\frac{1}{2}$

Smallest fraction = $0.1000_2 = \frac{1}{2}$

Largest fraction = $0.1111_2 = \frac{15}{16}$

Smallest number = $0.1000_2 \times 2^0 = \frac{1}{2}$

Largest number = $0.1111_2 \times 2^3 = 7\frac{1}{2}$

Number of fractions = $1 \times 2 \times 2 \times 2 = 8$

Number of values = $8 \times 4 = 32$

Adapted from David J. Cooke, *The Structure of Computers and Computations,* Tables 3.2 and 3.3 (1978), p. 203.

same in both systems. Finally, the 64 element capability for a six-bit system is more closely approached by the 48 values representable in this system than the previous system. Note that these values range from 1/4 to 60 — a much greater range than the base 2 system. However, also note that this system does not have the capability to represent many of the numbers represented in the base 2 system, such as $1\frac{1}{8}$.

These examples (6-bit normalized floating point number systems) underline the fact that not all floating point number systems are created equal. One N-bit floating point representation with its set of values for r_b, m, r_e, e, and so on, will have different characteristics from another N-bit floating point system. The designer is left with the task of selecting a representation which will fit the required combination of needs.

The method of storing numbers in a machine underlines the differences and similarities in computer floating point number systems. The information stored (or sent, or manipulated, or ...) is the sign, the exponent, and the mantissa. This information is usually grouped in that way: the sign of the number is the most significant bit, followed by the exponent, and then the mantissa. This is

Table 2.5. 6-Bit Normalized Floating Point System. Base 4.

$r_b = 4,\ r_e = 2,\ m = 2,\ e = 2$

		$V_E \rightarrow$	00 (0)	01 (1)	10 (2)	11 (3)
		$2^{V_E} \rightarrow$ (4^{V_E})	1 (4^0)	4 (4^1)	16 (4^2)	64 (4^3)
V_M base 4		V_M	$V_M \times 2^{V_E}$ ($V_M \times 4^{V_E}$)			
1	0	$\frac{1}{4}$	$\frac{1}{4}$	1	4	16
1	1	$\frac{5}{16}$	$\frac{5}{16}$	$1\frac{1}{4}$	5	20
1	2	$\frac{3}{8}$	$\frac{3}{8}$	$1\frac{1}{2}$	6	24
1	3	$\frac{7}{16}$	$\frac{7}{16}$	$1\frac{3}{4}$	7	28
2	0	$\frac{1}{2}$	$\frac{1}{2}$	2	8	32
2	1	$\frac{9}{16}$	$\frac{9}{16}$	$2\frac{1}{4}$	9	36
2	2	$\frac{5}{8}$	$\frac{5}{8}$	$2\frac{1}{2}$	10	40
2	3	$\frac{11}{16}$	$\frac{11}{16}$	$2\frac{3}{4}$	11	44
3	0	$\frac{3}{4}$	$\frac{3}{4}$	3	12	48
3	1	$\frac{13}{16}$	$\frac{13}{16}$	$3\frac{1}{4}$	13	52
3	2	$\frac{7}{8}$	$\frac{7}{8}$	$3\frac{1}{2}$	14	56
3	3	$\frac{15}{16}$	$\frac{15}{16}$	$3\frac{3}{4}$	15	60

Smallest fraction = 0.10_4 = $\frac{1}{}$

Largest fraction = 0.33_4 = $\frac{15}{16}$

Smallest number = 0.10, $\times 4^0$ = $

Largest number = $0.33_4 \times 4^3 = 60$

Number of fractions = 3 x 4 = 12

Number of values = 12 x 4 = 48

Adapted from David J. Cooke, The *Structure of Computers and Computations*, Tables 3.2 and 3.3 (1978). p. 203.

graphically depicted in Figure **2.2.** The information that never changes is not stored. Examples of this nonstored information are the radix of the system and the radix of the exponent; not so obvious examples of this are the location of the radix point and the coding method for storing the exponent. All of these are decided at design time, and remain constant for the life of the data. Another piece of constant information is the leading "1" for normalized base 2 mantissas. There is no reason to store this bit, and so the usual way to store a normalized base **2** mantissa is shown in Figure **2.2.** Only the bits that change are stored, so the most significant bit of the mantissa is said to be "hidden" behind the exponent. Some manufacturers refer to this as a "hidden bit" technique. The net result is to double the number of representable mantissas.

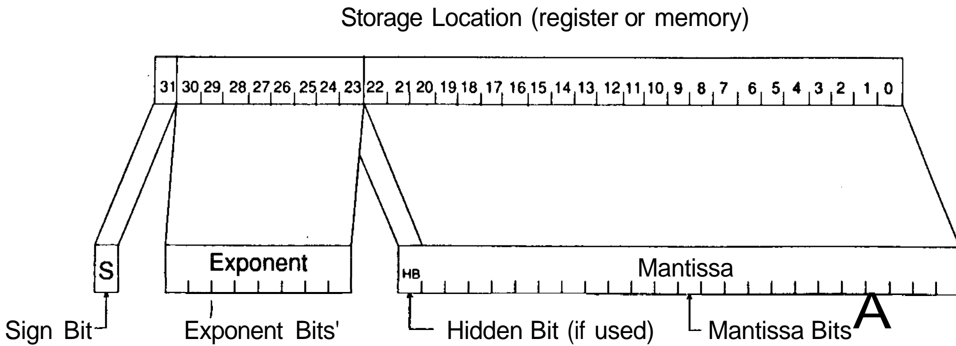

Figure 2.2. Normalized Floating Point Number Storage.

Some interesting observations can be made by considering the use of the "hidden bit" technique with the system depicted in Table 2.4. If this technique were adopted, then the number of legal mantissas would double, as would the representable values. Thus, all 64 combinations of bits would form correct floating point numbers. Note, however, the following: $V_{M_{MIN}}$ remains the same at 1/2, and $V_{M_{MAX}}$ is 31/32. The new representable values available by this technique are between each of the old values; the overall range is increased only from $7\frac{1}{2}$ to $7\frac{3}{4}$. So although the number of values has doubled, the range of representable numbers is basically the same. Finally, none of the values representable is zero; the smallest number is 000000, which turns out to be (with the hidden bit coming into play), $0.1000_2 \times 2^0 = 1/2$.

This raises the question: how is the number zero represented? We will examine this more closely in the examples to follow, but the general technique is to make an assumption concerning the exponent. A common method is to use an excess 2^{e-1} code for exponent representation. As the binary representations of this code vary from $2^e - 1$ to 1, the exponents vary from $2^{e-1} - 1$ to $-(2^{e-1} - 1)$. If the exponent bits are all zero, then the number is assumed to be zero, regardless of the values of the bits located in the mantissa field.

Example 2.8: Characteristics of a base 2 FPNS: Determine the characteristics, as defined by the above equations, of the DEC 32-bit normalized floating point number system.

This system has an r_b of 2, an r_e of 2, $m = 24$ with hidden bit, $p = 24$, $e = 8$, the exponent is stored in excess 128 code, and the number is stored in sign-magnitude form (mantissa is considered positive). So, from the above equations:

$$V_{M_{MIN}} = 0.1000..._2 = 1/2$$

$$V_{M_{MAX}} = 0.1111..._2 = 0.999999940395 = 1.0 - 2^{-24}$$

$$V_{FPN_{MIN}} = 0.1000..._2 \times 2^{-127} = 2.9387 \times 10^{-39}$$

$$V_{FPN_{MAX}} = 0.1111..._2 \times 2^{+127} = 1.7014 \times 10^{38}$$

$$NLM_{FPN} = 2^{23} = 8,388,608$$

$$NRV_{FPN} = 2^{23} \times (2^8 - 1) = 2.139 \times 10^9$$

Discussion: This system uses 99.6% of the available bit patterns for legal normalized floating point values. However, if the computations are very small or very large, then the system will not be able to provide the dynamic range needed for the calculations. One of the questions sometimes asked about a floating point system is how many significant digits are available. This is a subjective measure, since the amount of precision available is a function of the number only, not of the process producing the number. That is, for any V_E there are $2^{23} = 8.4 \times 10^6$ different values. A number in this system represented in base 10 scientific notation would require six digits; we say that there are six significant figures. even though a measurement represented by those figures may only be accurate to three places. DEC also provides a double precision format to increase the amount of precision with which calculations can be made. The double precision system is identical ($r_b = r_e = 2$, $e = 8$, exponent in excess 128 format, sign-magnitude representation for the number) except that the number of bits in the mantissa is extended from 24 to 56. Thus, the values cover the same sections of the real number line, but there are 2^{32} more values to represent the information to a greater precision. Thus for a given V_E there are 3.6×10^{16} values, or about 16 digits of significance.

The DEC format is prevalent simply because of the large number of DEC machines in use today. However, there are other systems available in both 32- and 64-bit formats. Another prevalent system is the IBM floating point system, which is analyzed in the next example.

Example 2.9: Characteristics of a base 16 FPNS: Determine the characteristics of the IBM 32-bit normalized floating point number system.

This system has an r_b of 16, an r_e of 2, $m = 6$ hexadecimal digits, $p = 6$, $e = 7$, the exponent is stored in excess 64 code, number is stored in sign-magnitude form (mantissa is considered positive).

$$V_{M_{MIN}} = 0.100000_{16} = 1/16$$

$$V_{M_{MAX}} = 0.FFFFFF_{16} = 0.999999940395 = 1.0 - 16^{-6}$$

$$V_{FPN_{MIN}} = 0.100000_{16} \times 16^{-63} = 8.636 \times 10^{-78}$$

$$V_{FPN_{MAX}} = 0.FFFFFF_{16} \times 16^{+63} = 7.237 \times 10^{75}$$

$$NLM_{FPN} = 15 \times 16^5 = 15,728,640$$

$$NRV_{FPN} = 15 \times 16^5 \times (2^7 - 1) = 1.9975 \times 10^9$$

This system has a far greater range than the DEC system, but actually has 7% fewer representable values. Nevertheless in certain applications, this is a reasonable system, and is chosen by some system designers. The IBM system has a double precision format which, like the DEC format, does not

extend the range of the system, but includes 16^8 more values in the same basic area of the real number line.

One of the disappointing features of a normalized floating point system is that there is a large discontinuity when the values of the numbers approach zero. This is depicted graphically in Figure 2.3, which shows the smallest representable values of the DEC floating point system. The problem is, with the stipulation that the first digit to the right of the radix point be nonzero, the first representable value away from zero is disproportionately large. This is one of the problems addressed by the IEEE floating point system, which is the object of the next example.

Example 2.10: Characteristics of IEEE FPNS: Determine the characteristics, as defined by the above equations, of the IEEE 32-bit and 64-bit normalized floating point number systems.

First, the 32-bit system: This system has an r_b of 2, an r_e of 2, m = 24 with hidden bit, but here p = 23; e = 8, the exponent is stored in excess 127 code, and the number is stored in sign-magnitude form (mantissa is considered positive). The effect of p being 23 while m is 24 is that rather than range from 1/2 to almost 1, as in the DEC system, these mantissas range from 1 to almost 2. Another difference is that the exponent V_E = 255 is special. That is, when the exponent is 255, special values are possible, such as infinity. So, the system has the following characteristics for normalized numbers:

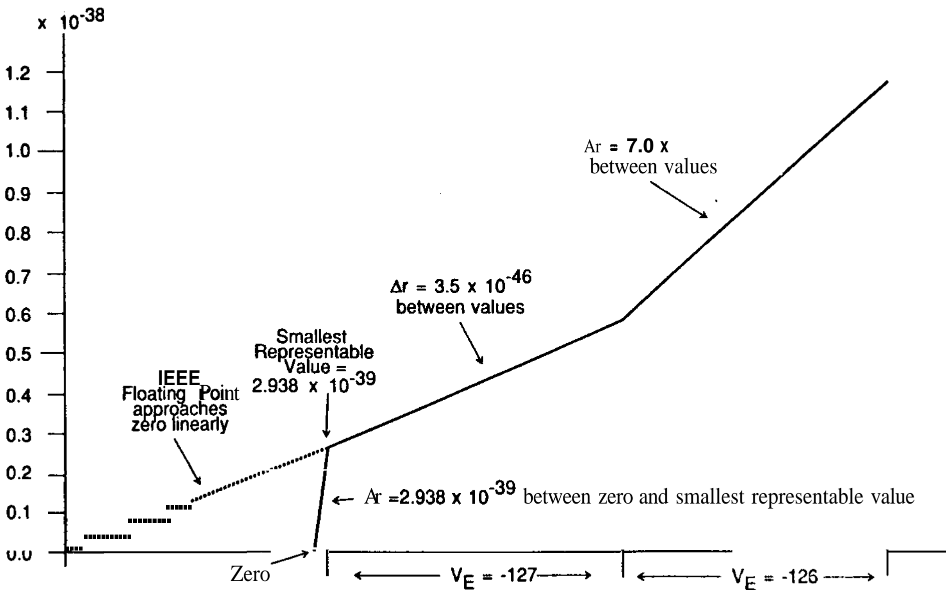

Figure 2.3. Values of the DEC Normalized Floating Point System Near Zero.

$$V_{M_{MIN}} = 1.000..._2 = 1$$

$$V_{M_{MAX}} = 1.111..._2 = 1.99999988 = 2.0 - 2^{-23}$$

$$V_{FPN_{MIN}} = 1.000..._2 \times 2^{-126} = 1.1755 \times 10^{-38}$$

$$V_{FPN_{MAX}} = 1.111..._2 \times 2^{+128} = 3.4028 \times 10^{38}$$

$$NLM_{FPN} = 2^{23} = 8{,}388{,}608$$

$$NRV_{FPN} = 2^{23} \times (2^8 - 2) = 2.131 \times 10^9$$

Discussion: Notice that the largest representable number in this system is twice as large as the $V_{FPN_{MAX}}$ of the DEC system. The reason for this is that the $V_{E_{MAX}}$ of both systems is the same (127), but the $V_{M_{MAX}}$ of the IEEE system is ~2, while the $V_{M_{MAX}}$ of the DEC system is ~1. At the other end of the normalized numbers, the smallest value ($V_{FPN_{MIN}}$) representable by the IEEE system is four times as large as the $V_{FPN_{MIN}}$ of the DEC system. One of the reasons the system is organized the way it is concerns the representation of zero, and the use of unnormalized numbers. The previous representations assumed that the number was zero if the exponent bits were all zero. The IEEE floating point system has a provision that lets the number become unnormalized as it approaches zero. That is, when the exponent bits are all zero, then the bits in the mantissa field continue to be significant, and the exponent remains at a –126. This allows the number to approach zero in a linear fashion, as opposed to the discontinuity depicted in Figure 2.3. There are two representations for zero: when all bits (excluding sign bit) are zero, the value of the number is zero. The use of unnormalized representations extends the range of the system down to 10^{-45}. The number of significant digits here is basically the same as the systems already discussed.

When V_E is 255, the system allows for representation of some specific information:

V_F	Sign	Meaning of Representation
$\neq 0$	1,0	Not a number (NaN)
0	0	$+\infty$
0	1	$-\infty$

The formal number system specification should be consulted for a complete explanation of the definition and use of the system. As shown above, the single precision system is very similar in many respects with the DEC system. However, the IEEE system changes the number of bits in the exponent for the double precision representation.

For the 64-bit representation, $r_b = r_e = 2$ as before, but m = 53, p = 52, e = 11, and the exponent is stored in excess 1,023 format. With these modifications, the characteristics of the number system change somewhat. The $V_{M_{MIN}}$ is still 1.0, and the $V_{M_{MAX}}$ gets closer to two $(2 - 2^{-52})$. But:

$$V_{FPN_{MIN}} = 1.000..._2 \times 2^{-1022} = 2.225 \times 10^{-308}$$

$$V_{FPN_{MAX}} = 1.111..._2 \times 2^{+1023} = 1.798 \times 10^{308}$$

$$NLM_{FPN} = 2^{52} = 4.5 \times 10^{15}$$

$$NRV_{FPN} = 2^{52} \times (2^{11} - 2) = 9.214 \times 10^{18}$$

The double precision system provides 15 significant digits, and has a range much larger than either the DEC or IBM double precision formats.

The floating point number systems already mentioned are the ones that are most accessible for the majority of computer users. This is especially true in that the newer machines are utilizing chip sets that conform to the IEEE standard. This is true of the:

- 68020, which is the processor in many of the Sun computers.
- 80386, which is in the Sequent systems and many other computer systems.
- 32332, which is in the Encore systems and other computers.
- MIPS R2000, which is used in many engineering workstations.

In addition, many chip sets (see the AMD29C327, the ADP2100, etc.) are available for users to implement machines of their own design, utilizing the IEEE floating point format. And the DLC and IBM machines continue to be extremely prevalent throughout the computing community.

We will mention one more floating point system, that utilized by the Cray machines. This 64-bit format is utilized a great deal for scientific computing.

Example 2.11: Cray FPNS: Consider the Cray 64-bit floating point format. This system has an r_b of 2, an r_e of 2, m = 48, p = 48, e = 15, number is stored in sign-magnitude form (mantissa is considered positive). The exponent is stored in excess 16,384 format. With an exponent so large, Cray does not use the full range, but rather uses the uppermost (and also, the most negative) portions to identify underflow and overflow. This information is then stored in the number itself. The effect is to have a maximum positive exponent of 8,191, and a maximum negative exponent of -8,192. This gives rise to the following set of numbers:

$$V_{FPN_{MIN}} = 0.1000..._2 \times 2^{-8192} = 4.584 \times 10^{-2467}$$

$$V_{FPN_{MAX}} = 0.1111..._2 \times 2^{+8191} = 5.4537 \times 10^{2465}$$

$$NLM_{FPN} = 2^{48} = 2.815 \times 10^{14}$$

$$NRV_{FPN} = 2^{48} \times (2^{14} - 1) = 4.6114 \times 10^{18}$$

This system has an extremely large range, and carries about 14 significant figures. The effect is to have a number system capable of extremely large and extremely small numbers, and sufficient significance for almost all necessary computations. Compare this system, for example, with the 64-bit IEEE format, which does not have the extreme range, but does carry over 15 digits of significance.

The information in this section points out the fact that not all floating point number systems are created equal. Each of the designers of the various systems

has been influenced by a different set of real or perceived requirements in the choices made. Table 2.6 identifies a number of machines and the floating point choices made for them. We should add here that many manufacturers support floating point formats (as a special option) which go beyond those we have identified here. Some 128-bit and 256-bit formats allow calculations with extremely large and small numbers. However, the principles of data representation are the same, and an understanding of the principles discussed here will apply to the larger numbers.

As we have seen, a variety of floating point formats have been utilized in the design of different computer systems. One of the problems that arises is the exchange of data from one system to another: the bit patterns cannot be directly exchanged, even if the number of bits used in the representations is identical. However, if one is aware of the differences in the number systems, one can take the necessary steps to make sure that the numbers on one machine are correct on another. In any event, this should serve as a reminder that with N bits on a computer, 2^N different bit patterns are available. Floating point number systems allow the expression of large and small quantities, but do not expand the number of allowable representations.

2.3. Coding for Nonnumeric Information

The information utilized by computers in various tasks is not limited to numbers. Thus far, we have examined utilizing bit patterns to represent integer and floating point information. Other information must also be stored within the machine, or on electronic media such as tape or disks. This information may be instructions, text, addresses, status information, or other information needed by the machine. This information will be represented by bit patterns, just as the numbers were represented by bit patterns. And in a manner similar to the numeric data, the assumptions about the format of the information is made at design time. We will briefly examine several types of nonnumeric information in this section, including text, boolean, graphics symbols, and addresses.

Textual information has become one of the most often utilized forms of information for both storage and manipulation. This seems counterintuitive, since computers have historically been used to "compute," that is, doing calculations for a variety of applications. However, when one considers the fact that programs are input in text form, that compilers operate on strings of characters, and that answers are generally provided via some type of textual information, then the amount of character information begins to be appreciated. A more recent utilization for computers is in the office, where reports, letters, contracts, and other types of printed information are generated. In short, many applications must store, manipulate, and transfer textual information. How can this be accomplished?

One question in this regard is, what is the set of elements to be represented? Those interested in mathematical information would immediately respond with the characters needed to represent data: the digits (0–9), decimal point, plus, minus, and space. This gives a minimal character set with only 14 elements. However, there are severe limitations to the understandability of the results: no labels, no carriage returns or line feeds, and so on. So, at least add the alphabet (A–Z), punctuation, and formatting characters (comma, tab, carriage return, line feed, form feed, parenthesis). This gets the number of elements up to 46. We know that, in order to represent 46 different elements, we will need at least

Table 2.6. Floating Point Information Systems.

System	Word Size (# Bits)	r_b	Exponent # Bits	Exponent Code	Mantissa # Bits	Mantissa Repre.	Mantissa Code
Burroughs B6700/7700	48	8	7	SM	39	Int	SM
CDC 7600	60	2	11	Ex 1024	48	Int	1's C
DEC — single	32	2	8	Ex 128	24	Fra	SM
DEC—double	64	2	8	Ex 128	56	Fra	SM
Honeywell 8200	48	2 or 10	7	Ex 64	40 (base 2) 20 (base 10)	Fra Fra	SM
IBM — single	32	16	7	Ex 64	24	Fra	SM
IBM —double	64	16	7	Ex 64	56	Fra	SM
IEEE — single	32	2	8	Ex 127	24	Fra	SM
IEEE — double	64	2	11	Ex 1023	53	Fra	SM
Cray	64	2	15	Ex 16384	48	Fra	SM

Int = Integer representation

SM = Sign/magnitude

Fra = Fractional

1's C = One's complement

Ex = Excess code

$\lceil \log_2 46 \rceil$ = 6 bits. With 6 bits we would be able to represent 2^6 = 64 different bit patterns, or 64 different elements in the set. So we can represent most of the information that we need with 6-bit characters; however, note that this set is not large enough to include both upper- and lowercase letters. Character sets that are to represent both upper- and lowercase letters, control characters, punctuation marks, and other special characters must have at least 7 bits. The bit patterns can then be mapped to the characters or control information to be represented.

One of the early types of devices utilized to communicate with computers was the card reader. This mechanical marvel utilized a coding scheme to represent its various information. The information represented in the earliest machines included only uppercase letters, numbers, and special characters. To represent this information a code was developed for use with the card reader which was capable of this reduced set of characters. This 6-bit code, called the BCD code, should not be confused with the 4-bit representation mentioned earlier

in this chapter used to represent the digits 0–9. Later, the 6-bit **BCD** code was extended to include the lowercase characters and additional information needed in computer communications. This code (given in Appendix A) is known as **EBCDIC:** Extended Binary Coded Decimal Interchange Code. It was used in the **IBM** 3601370 and other IBM equipment, but is not in general use in computers. However, this information can be useful if one needs to decode data generated by an **EBCDIC** machine. An examination of the code reveals that not all of the $2^8 =$ 256 representations are used. However, all 8 of the bits are required to specify the various characters and control codes. It is also interesting to note that arithmetic using these codes may not always give the desired result. That is, if a routine were written to write out all of the standard letters (A–Z) in the alphabet, then one way to approach it would be to place the code for " A in a register, and increment it to get the code for "B", and so on. However, note the discontinuity at "I": the code for the letter "J" is not the next in numeric sequence from "I." This illustrates one of the reasons that the code was not widely received.

Another code, which has received almost universal acceptance for the representation of textual information, is the **ASCII** code: American Standard Code for Information Interchange (also in Appendix A). In contrast to the **EBCDIC** format, the **ASCII** code is a 7-bit representation, which limits it to 128 different values. The difficulty mentioned in connection with the **EBCDIC** format does not apply to the **ASCII** code: incrementing the representation of a letter gives the successive letter, except for "Z." This code is used in most terminals, printers, and other devices that deal with character information.

The normal method for handling this information is to place the bit pattern in an 8-bit field called a byte. The **EBCDIC** format would utilize all of the bits in a byte, while the **ASCII** code would "waste" one of the bits. These bytes form **8-**bit values, which are treated in exactly the same fashion as numbers. Thus, the hardware elements that operate on integers will also operate on characters. This allows one set of characters to be compared to a similar set of information, to be searched for specific patterns, or to be operated on by programs seeking statistical information. A spelling check program, for example, would identify a group of letters as a word, then compare that word against words that it knows are spelled correctly. If the program is unable to recognize the word, or construct it from a known word according to a set of rules, then the word is labeled as incorrect. and the operator is informed of this infraction. In all of these operations, the computer is operating on the bit patterns representing the characters, and the meaning of those characters becomes significant only to the humans at the end of the process.

The difficulty of expanding the set of representable elements can be overcome in a variety of ways. One obvious way would be to include the eighth bit of the **ASCII** code, doubling the available representations. Another method is exemplified by the character codes used in some 60-bit machines. Some machines have been built with 60-bit word lengths, a compromise between the needed accuracy and the expensive memory available at design time. It is not possible to equally divide the 60-bit word into either 7- or 8-bit quantities. So the system designers implemented a 6-bit system, which limited the number of available characters to 64. This system works well as long as the information output is in the specified characters, which consist of the numbers, the uppercase letters, common punctuation, and special characters. To represent the lowercase characters, a **two-character** sequence is used. The first character is an "escape" character, which informs the system that the desired character was not in the standard set, but rather in an alternate set. And the pattern identifying that character in the

alternate set is found in the next 6-bit field. The effect of this method of information representation is to use 6 bits to represent some characters (preferably the most often used), and 12 bits to represent other characters. This method works well for data that is basically numbers and uppercase characters. But for text. such as correspondence or reports, this method is cumbersome and wasteful of bits.

The same type of arrangements can be utilized to enhance the number of representable elements for "standard" character sets. For example, nonstandard characters, such as Greek characters or special purpose characters (≠, ∞, ±, **x, ...**) can be represented in this fashion. These characters become even more important as graphics-orienteddevices become more prevalent.

Regardless of the coding scheme chosen, the computer deals with characters in the same fashion as it does with other data; the arithmetic is performed in the same way, and conditions are tested in much the same fashion. The result of a test is an example of another type of information: boolean. In general, the term "boolean" refers to information that can assume one of two possible values. For status information, this seems intuitively obvious: is the result of the arithmetic operation positive or negative? Is there an overflow or not? This type of information requires only a single bit to represent. In fact, a status register in a machine is nothing more than a collection of this type of single bits. Depending on the instruction set of the machine, these bits may or may not be individually setable/clearable/testable. We will examine this issue more closely when we discuss instruction sets.

Many languages also allow this type of variable to be declared. In general. the language will utilize an entire word to represent this information, which wastes a lot of bits. The smallest unit that could represent this under language control would be the smallest addressable unit of the computer. In most computers, the smallest addressable unit is the byte, but some large, mathematical type machines have a smallest addressable unit of a word (32 or 64 bits). In any case, compilers can be called upon to generate sequences of instructions that will allow storing of boolean information in individual bits of a word. This is a tradeoff between the use of time and the use of memory. Assigning a boolean variable to the smallest addressable unit of the machine will be faster than the alternative, which is to have boolean information limited to. individual bits within a word. However, this method uses more memory. The alternative, using individual bits for storage of boolean information, requires a smaller amount of storage for the boolean information, but also requires more instructions to interact with this information.

In general, information is stored in locations within the memory of the machine, and those locations are identified by addresses. The addresses, which themselves form information that can be manipulated and utilized as needed, are simply numbers that can be considered integers. The number of bits in the address determines the number of uniquely identifiable items: N bits specifies one of 2^N unique items. The address is utilized by the machine to "point" to an item; hence, the use of the address in this manner gives rise to the term "pointer." Pointers are very useful to create within a machine an instantiation of an abstraction, such as a tree structure or queue.

Bit patterns, then. can be used to represent different types of information: numbers (integer, floating point, fixed point. ...), characters, symbols, addresses, and so on. The meaning attached to the bit pattern is a function of when it is used and where it is found.

2.4. Coding for Errors — Detection and Correction

Information can be represented in a variety of ways, and in the previous sections we examined some of the different coding techniques. We know that with N bits we can represent 2^N different values, or addresses, or instructions, or ... . To represent anything else requires more bits, and the number of additional bits needed is determined by the amount of information to be supplied. If we add one more bit to an N-bit representation, then the number of representable elements doubles, from 2^N to 2^{N+1}. So how are these additional elements identified and treated? That is, what rules exist to effectively utilize the additional information present? Let us examine rules (codes) for adding sufficient information to detect the presence of errors. And then we will examine some rules (codes) for adding enough information not only to detect that an error occurred, but also to correct that error.

Perhaps the simplest method of adding error detection to data is to include a parity bit. Here, the N bits of information is augmented by an additional bit, which doubles the representable patterns. However, this additional bit is so constructed that half of these patterns will not be legal representations. Hence, it is possible to detect the presence of a single bit that is incorrect. The fault can be a "stuck-at" fault, in which the incorrect bit is "stuck" at that incorrect value, or it can be transitory in nature. In either case, if only one bit is incorrect, the manner of constructing the correct code words enables us to detect that an error has occurred. The construction rule is to choose the value for the additional bit so that the number of "one" bits is odd (or even). Figure 2.4 gives a circuit that will create the proper signal for 8 bits. This circuit is available in integrated circuit form as a **'280.** Note that the expansion of the exclusive-OR tree by one bit would enable checking the parity across 9 bits, to identify if it is odd (or even). (The **'280** is an exclusive-OR tree for nine bits.)

This type of error detection is useful wherever errors have a reasonable probability of occurrence. In general they are used in serial transmissions (terminal lines, etc.), in parallel data transmission systems (buses), or in memory systems. However, some conditions will invalidate the effectiveness of the use of a parity check. That is, if the assumptions of the fault model are exceeded or not applicable, then the effectiveness of the method is moot. For example, in one of the errors observed in bus systems the data is read (incorrectly) as all zero's. This would be a valid even parity condition, and so a system built to check for even parity would not detectthe presence of an error. Likewise, for a 16-bit bus with

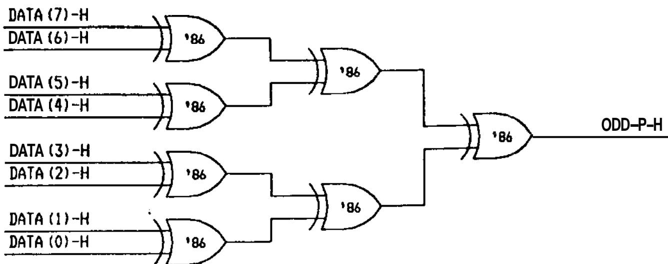

Figure 2.4. Generation of Parity Bits.

an additional bit for parity, the total number of lines is 17. And the similar error of reading all the lines as one's would be correct for a system using odd parity. One method suggested for this situation is to use two parity bits for the 16-bit bus, one for each byte. The parity sense of half of the bus would be set to odd, the other half to even. Then if all zero's were read, half of the bus would complain. And if the condition of all one's were read, then the other half of the bus would complain. In either case, the fact that an error had occurred would be correctly identified, even though the assumptions of the fault model had been violated.

Example 2.12: Parity defectionand generation: Construct a circuit providing a bidirectional data path that is 8 bits of data plus parity. That is, one side of the path is a byte-wide source/destination of information, and the other side is a tri-state data bus that includes a parity bit.

The solution of this problem is to expand the circuit given in Figure 2.4 to include the generate/decode capability. An example of such a circuit is given in Figure 2.5. This figure shows that the data path is treated in the same way that a data path might be if no parity capability were required: the data is fed through a bidirectional tri-state transceiver ('245). So the only bit that needs to be dealt with is the parity bit (PARITY-H). When the direction line (IN-H) identifies incoming data, the parity line is enabled into the parity circuit to check consistency. If the parity sense is incorrect, then the error line (ERROR-L) is asserted. When the direction line identifies that this module provides information to the bus, then the outgoing parity generator is enabled, and the parity line is driven in the same way (by different physical circuits) that the bus lines are driven. Figure 2.5 shows the parity circuits as being separate, but they need not be, and cleverness in the design will match system requirements with an appropriate circuit. Some integrated circuits will do this function, such as the '286, a symbol of which is shown in Figure 2.6.

If we want be able to identify the location of an error, then more information must be added to the system than can be added by a single bit. A sufficiently

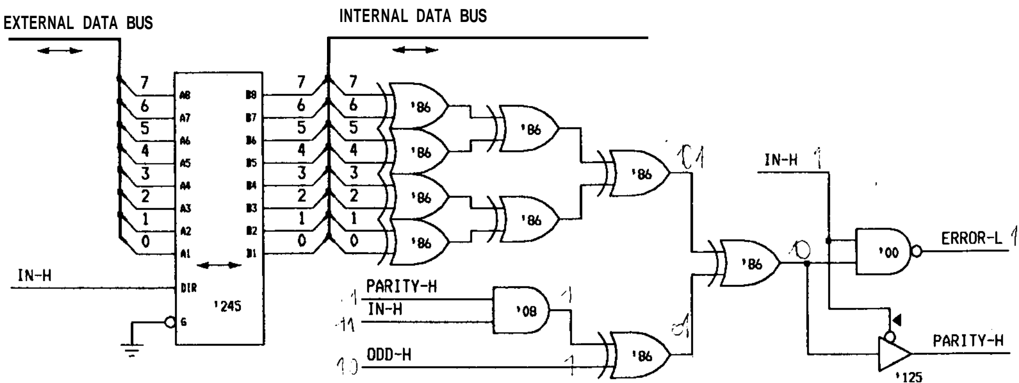

Figure 2.5. Byte-Wide Data Path with Bidirectional Parity Bit.

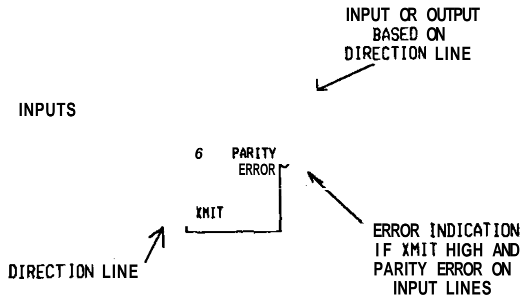

Figure 2.6. **Functional Diagram for '286 Parity Checker/Generator.**

large number of bits must be added to the data to not only identify the fact that one of the bits is in error, but also identify the faulty bit. Again, the fault model can be a stuck-at or a transient fault. But, for our discussion, we are limiting the errors to a single fault within the word. One class of codes that allows this type of information to be encoded into the extra bits is the set of Hamming codes. Many methods can be used to construct a code of this type. We will examine one method, but, once the principles are understood, the exact implementation and design choices can be driven by whatever constraints are imposed by the system. That is. the code could be chosen so that a minimal number of gates are required to identify errors, or the code could be chosen in an attempt to optimally position the 2^N valid code words in the total of 2^{N+p} choices, where p code bits are added to N data bits.

First, let us describe one method for construction of a code to identify the location of an error. To illustrate this method, we will utilize a system with 4 data bits and 3 code bits, or 7 bits in all. We will arrange these bits as shown in Figure 2.7, with the data bits labeled D_3, D_2, D_1, and D_0, and the code bits labeled C_2, C_1, and C_0. Note that the code bits physically occupy the positions corresponding to their binary weight. Thus, C_0 is in the $2^0 = 1$ position; C_1 is in the $2^1 = 2$ position; and C_2 is in the $2^2 = 4$ position. The remaining positions are occupied by the 4 data bits. Also shown in the figure is the fact that the code bits are constructed in such a way that parity is preserved across a subset of the bits of

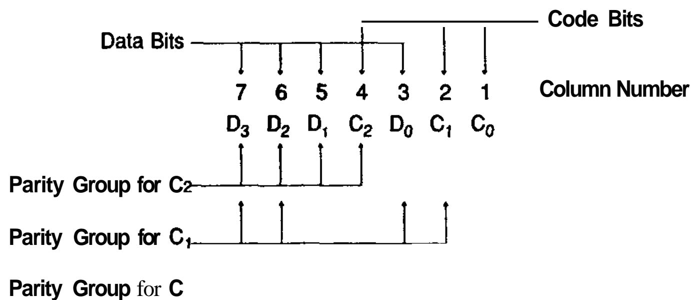

Figure 2.7. **Construction of a Hamming Code for 7 Bits.**

the entire word. The subsets are so constructed that when a single bit is in error, a unique pattern is identified by the code bits. Note that there is a single code bit in each subset; let the subset associated with code bit C_k be called set k. In this example, subset k contains all of the bits that have the 2^k bit set in the binary representation of bit position. The following table describes the situation for this system:

Set	*Bit Positions*	*Bit Names*
0	1, 3, 5, 7	C_0, D_0, D_1, D_3
1	2, 3, 6, 7	C_1, D_0, D_2, D_3
2	4, 5, 6, 7	C_2, D_1, D_2, D_3

Since three code bits are associated with this method, there. must be three parity circuits to generate the three parity bits when the word is written — and three parity circuits to check the parity when the word is read. The ordering of these bits is such that they form a 3-bit word (set 2, set 1, set 0), which will identify a column. In Figure 2.8, the number 0101 is used as an example. The bits are placed in their proper position in the word, the code bits are generated assuming odd parity sense, and the result is presented as 0100110. If there are no errors, the output of the parity checkers for the subsets would be 000, which identifies a nonexistent bit position in our system. If the pattern 0100010 is detected, then the parity checks of the subsets identifies some errors. Set 2 is correct, set 1 is

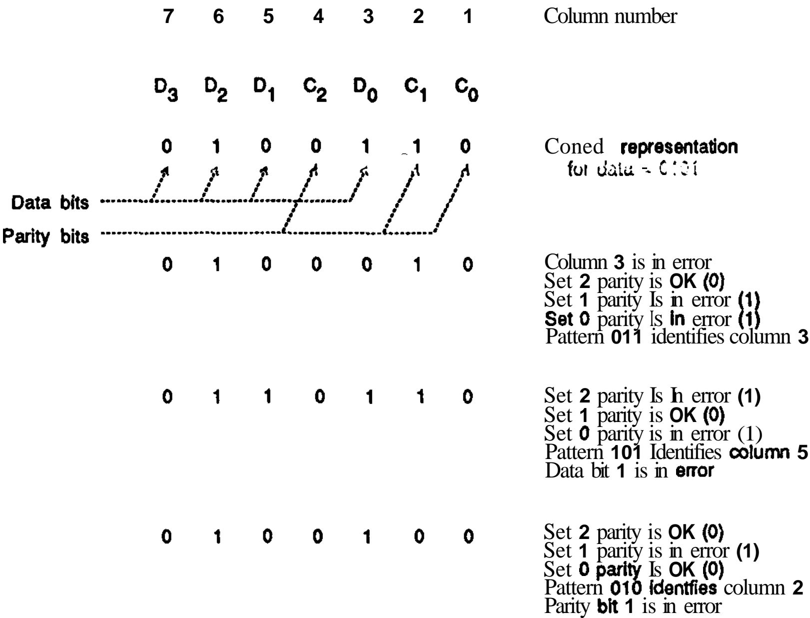

Figure 2.8. Hamming Code Examples for Data = 0101.

incorrect. and set 0 is incorrect, which results in an error syndrome of 011. This identifies the fact that the bit in position 3 is incorrect, and to make the word correct all that needs to happen is to invert the bit in position 3, D_0. Figure 2.9 indicates how some parity checkers, a decoder, and some exclusive-OR gates could be connected to perform this function.

The method described above was constructed to have the property previously mentioned: any single error will produce a unique bit pattern at the output of the parity check stages. Also, the placement of the code bits was done in such a fashion that decoding the location of the error from the pattern which the error created could be accomplished with a standard decoder IC. Other coding schemes are possible, as long as each single error causes a unique response, and a decoder system can be constructed to identify the location of the error from that response.

This method can effectively utilize $2^N - 1$ bits, N bits to create the data dependent code, and up to $2^N - 1 - N$ bits for data. For small N, $2^N - 1 - N$ data bits is also small; thus. for a small number of data bits the overhead is large. However, as the number of data bits becomes larger. then the overhead is reduced. For example, a system with 64 data bits would require 7 code bits, or about a 10% overhead. One of the problems incurred in using an error correcting code in a memory system is the fact that many machines are byte addressable. That is, even though the system memory may be organized in 32- or 64-bit elements for the error correction capability, the system must be able to modify only part of the data bits in a 32- or 64-bit word. This requires a read/modify/write capability, so that the other parts of the data word remain correct, and the code bits are

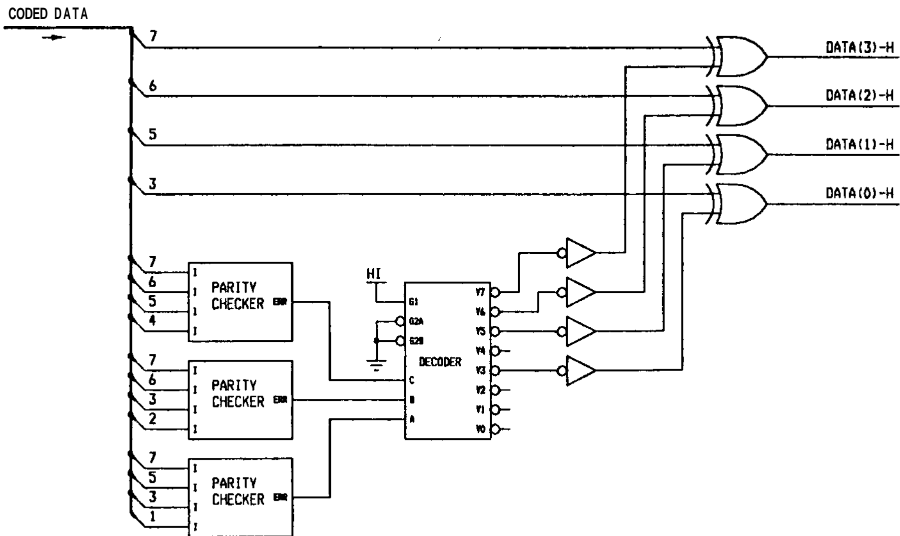

Figure 2.9. Correction Circuit for 7-Bit Hamming Code System..

appropriately set. In any case, this type of code will properly provide capability for single error correction (SEC).

This method is not sufficient to also provide double error detection (DED). Notice that, if two errors were to occur in the example just given, then an incorrect bit position would be identified, and the results would be wrong. Double error detection can be added to this method by including a single parity bit across the entire word, data bits and code bits alike. This bit would be created after the code bits had already been identified. So, the decoding system needs to take into account parity errors detected by the code bits. C_2, C_1, C_0, and occurrences of parity errors detected by the double **error** bit. These **are** then handled in the following fashion:

Parity Condition		
Double Bit	***Code Bits***	***Comment***
Correct	Correct	No error detected; normal condition.
Incorrect	Incorrect	Single error; **location** of error identified by binary weighting of code bits.
Incorrect	Correct	Single bit error; double bit is incorrect.
Correct	Incorrect	Double bit error; two bits in error — not correctable.

Example 2.13: Hamming Code for 8 bits: Consider a code for 8 data bits constructed after the pattern described above. In this code. what is **the** correct representation for 01011100? Also, describe the infomiation available in the patterns.

This code will require 8 data bits (D_7 – D_0), 4 code bits (C_3 – C_0), and a double error bit (DEB). Following the pattern above, **these** will be arranged as follows:

(DEB —Double Error Bit; BW — Binary Weight)

13	**12**	11	10	9	8	7	**6**	**5**	4	3	2	1	Columnnumber
DEB	D_7	D_6	D_5	D_4	C_3	D_3	D_2	D_1	C_2	D_0	C,	C_0	Content of bit position
	I	I	1	I	1	0	0	0	0	0	0	0	**BW** of Column Number; 8-bit
	I	0	0	0	0	1	I	I	1	0	0	0	BW **of Column** Number; 4-bit
	0	I	1	0	0	I	1	0	0	I	1	0	BW **of Column** Number; 2-bit
	0	1	0	1	0	I	0	1	0	1	0	1	**BW of Column Number;** 1-bit

From the above information. set 3 consists of C_3, D_4, D_5, D_6, and D_7. Set 2 consists of C_2, D_1, D_2, D_3, and D_7. Set 1 consists of C_1, D_0, D_2, D_3, D_5, and D_6. Set 0 consists of C_0, D_0, D_1, D_3, D_4, and D_6. With this information, the desired pattern can be created:

	0	1	0	1		1	1	0		0			Placement of **data bits**
1	0	1	0	1	1	1	I	0	1	0	0	0	Code bits added to word

2.5. Information Representation— A Matter of Bits

We have discussed a number of different methods of representing information. A collection of bits will be interpreted by the computer in any of a number of ways. depending on the instruction being executed, the number systems adopted by the

designers, and the coding schemes employed. It is not sufficient to know the pattern of ones and zeroes; we must know the rules concerning the interpretation of those bits. The **rules** for interpretation of the information are established at design time, and will be effective throughout the life of the system. These rules will enable the following pattern to be correctly interpreted:

01001110011100100000000000000000

If this is a DEC floating point number, it has the value of 2.537×10^8. If this is a IEEE floating point number, it has the value of 1.015×10^9. If it is an integer, it has the value 2.106×10^{10}. If this is a 68000 instruction, the computer should respond by performing a stop if the system is in the supervisor state; otherwise it will trap. If it is to be part of an ASCII character string, then it will provide the characters "Nr <nul> <nul>". In any case, the spectrum of possibilities of information content is limited in quantity, since N bits allows only for 2^N representable values. However, the interpretation of those values is influenced by the circumstance in which the value is found. The system designers make the choices that will allow representation of the information in a sensible and coherent fashion.

Information representation requires that both the supplier and the user of the patterns agree on the significance of the arrangements of digits (bits). The current technology represents information within a computing system in the form of bits, and those bits can be organized in many ways. However, the use of standard representations promotes systematic interpretation of the information. As we have seen,

- With N bits for the representation of information. 2^N different things can be represented.
- The coding of the N bits in an integer form allows representation of 2^N numerical values, all separated from their neighbors by one $(\Delta r = 1)$.
- Integer representations can assume different coding schemes, such as ones complement, twos complement, excess codes, and the like, each of which has its own unique set of characteristics. The coding scheme for a number is a choice made at the definition/design stage of a computer system, and the choice is made in such a way that the system will behave in a predictable and appropriate fashion.
- Most computer systems use two's complement representations for integer values.
- Coding of information in a floating point format allows the range of the representable numbers to increase dramatically. This allows computer users to remove themselves from the scaling aspects of the data manipulation.
- The magnitude of the representable values in a floating point number system changes with each exponent. And the distance between representable values (Δr) doubles each time the exponent increases in value by one.
- Not all floating point number systems are created equal. They have different capabilities for storing information and different ranges, which effect their applicability for user problems. The choice of the radix, the placement of the radix point, and the coding schemes all influence the values that can be represented by the system.

- Bit patterns can be used to represent other types of information besides numbers. Characters, instructions, addresses, and status information are just a few of the kinds of information also represented as a collection of bits.
- By creating rules concerning the legal patterns of bits, sufficient information can be included in a pattern of bits to identify the fact that an error occurred, and find the location of the error. This reduces the total number of correct values represented by a number of bits, but can be very useful in identifying problems in data transfers.

2.6. Problems

2.1 If the technology were available for a wire or "bit" to represent three values rather than two, what would the result be? That is, consider a system with n tertiary bits, as opposed to n binary bits. Each "bit" in this system would be capable of representing the values 0, 1, or 2. How many different values could be represented with 8 tertiary bits? 16 tertiary bits? 32 tertiary bits? What is the general formula for the total number of values available in the tertiary system?

2.2 Examples were given in the chapter to demonstrate the mechanism involved in adding one's complement arithmetic, i.e., the end-around carry. Prove that the end-around carry works and is needed.

2.3 What does the bit pattern 10010101 represent in the following systems: unsigned binary, 8-bit two's complement,, 8-bit one's complement, 8-bit BCD (2 digits), two 4-bit excess 3 coded base 10 digits.

2.4 Represent +95 and –95 as 8-bit one's complement and 8-bit two's complement numbers.

2.5 Express the following base 10 numbers in a 4-bit-per-digit excess 3 code: 45932 and 51373. Add the numbers together. Express the result in the same code.

2.6 Consider a 12 bit integer number system which is an excess 1,023 system. What is the smallest representable value? The largest representable value'! What is the representation of zero?

2.7 Consider a 12-bit fixed point two's complement number system with $p=7$. What is the smallest representable (positive) value? What is the largest representable value? What is the most negative value? What is Ar for this system?

2.8 Consider an 8-bit fixed point two's complement number system. Give the equation for the value of a number. Multiply two such equations together to give the result of a multiplication. Give an algorithm for selecting the proper 8 bits from all of the bits available after a multiplication.

2.9 A base 10, 5-digit, sign-magnitude system has a value of p equal to 3. What is the largest representable number? What is the smallest representable (positive) number? What is the most negative representable number? What is Δr for this system?

2.10 Consider a 16-bit floating point number stored in the following format:

s eeeee fffffffff

The "s" represents the sign of the number. The "eeeee" is the exponent. stored in excess 12. The "fff..." is the 'mantissa, which is a base 2 fraction, stored with the hidden bit technique. Give the characteristics ($V_{FPN_{MAX}}$, etc.) of this number system. What is Δr for this system when the value of the exponent is zero?

2.11 A floating point number system has the basic format given in 2.10, but the mantissa is a base 4 fraction, so that the hidden bit technique is not viable. Give the characteristics for this system. What is Δr for this system when the value of the exponent is zero?

2.12 Consider a floating point number system with the following characteristics: **normalized,** radix of the system is 4, radix of the exponent is 2, 12 bits total, with 4 bits in the exponent (e = 4), exponent stored in excess 8 format, 8 bits in the mantissa (m = 4, since base 4 number), mantissa stored in fractional form.

a. What is the smallest representable nonzero value?

b. What is the largest representable value?

c. What is the decimal equivalent of Δr when the exponent pattern is 0111?

d. What is the value, base 10, of the following pattern: 110001001011?

e. What is the pattern for the number $2\frac{3}{16}$?

f. What is the resulting pattern from adding the following positive numbers: 011001101011 and 100010110110? Use rounding for the result.

2.13 Given the following floating point format

s exp man

where the "s" is a 1-bit sign, "exp" is the 4-bit exponent field (exponent stored in excess 4 format), and "man" is the fractional mantissa, base 8, 6 bits wide. The format is for a normalized number system. Give the;

a. largest fraction.

b. smallest fraction.

c. largest number.

d. smallest number.

f. what number is represented by 00110101000?

g. represent the number 5/16 in this format.

2.14 A Hamming code has been created with the following pattern:

7	6	5	4	3	2	1	Column Number
D3	D2	D1	P2	D0	P1	P0	Data, parity designators

The code is constructed as discussed in the text. The parity sense is odd. Given that information, answer the following:

0	1	0	0	1	1	0	Part A
1	0	0	0	0	1	1	Part B
1	1	0	0	1	1	1	Part C
0	1	0	1	0	0	0	Part D

a. Is the representation correct? If not, is it correctable? To what?

b. Is the representation correct? If not, is it correctable? To what?

c. Is the representation correct? If not, is it correctable? To what?

d. What is the number represented?

e. Represent the number 6 in this code.

2.15 Consider the following floating point system:

a b c

a = sign of mantissa
b = 4-bit exponent in excess 8 code, radix = 4
c = 7-bit normalized mantissa

a. What number is represented by 010111001000?

b. What number is represented by 101111100000?

c. Represent 4 9 in this code.

d. Represent 114 in this code.

2.16 An error detectinglcorrectingcode is constructed as described in the chapter, with the following format (parity sense is odd parity; PA is parity across the entire word):

Col →	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Name →	PA	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	P_3	D_3	D_2	D_1	P_2	D_0	P_1	P_0
A	0	1	0	1	0	1	1	1	1	0	1	0	0	1	1	1
B	1	1	0	0	1	0	1	1	1	0	0	0	0	1	1	0
C	0	0	1	0	1	0	0	0	1	1	1	1	0	0	1	0
D	0	1	1	1	0	1	0	1	0	0	0	0	0	1	1	0
E																

For the first four numbers: if there is a single error, identify the bit in error; if there is a double error, indicate this result. For the final part, create the correct code for the decimal number 653.

2.17 Give the bit pattern for the following numbers in the 32-bit DEC, IEEE, and IBM floating point formats: **12,** 127, **2.5, 768.**

2.18 For the number systems listed in Table 2.6, find the minimum and maximum positive nonzero representable values.

2.19 Construct a 16-bit SECDED code using the technique demonstrated in Section 2.4. In this code, represent –512 and 183.

2.20 Design a combinational circuit that will correct single errors in a 7-bit Hamming **coded** word. The inputs thus are (all H asserted):

$$D_3 \quad D_2 \quad D_1 \quad P_2 \quad D_0 \quad P_1 \quad P_0$$

Odd parity is used. Outputs are H asserted also. Use any basic gates you choose, including **EXORs,** but make sure you maintain polarized mnemonics, incompatibility triangles, and so on. Explain any logic that is not intuitively obvious to the casual observer.

2.7. References and Readings

[Bart85] Bartee, T. C.. *Digital Computer Fundamentals, 6th edition.* New York: McGraw Hill Book Company, 1985.

[Boot84] Booth. T. L., *Introduction to Computer Engineering: Hardware and Software Design.* New York: John Wiley & Sons, 1984.

[Bree89] Breeding, K. I.. *Digital Design Fundamentals.* Englewood Cliffs, NJ: Prentice Hall. 1989.

[Cody84] Cody. W. I. et al.. "A Proposed Radix-and Word-Length-Independent Standard for Floating-Point Arithmetic," *IEEE Micro.* Vol. 4, No. 4, August 1984, pp. 86–100.

[Flet80] Fletcher. W. I.. *An Engineering Approach to Digital Design.* Englewood Cliffs, NJ: Prentice Hall. 1980.

[IEEE85] Institute of Electrical and Electronic Engineers. *Binary Floating Point Arithmetic.* IEEE Standard 754-1985. New York: IEEE, 1985.

[Knut73] Knuth. D. E., *The Art of Computer Programming: Volume 1. Fundamental Algorithm.* Reading. MA: Addiwn-Wesley. 1973.

[Knut69] Knuth, D. E.. *The Art of Computer Programming: Volume 2, Seminumerical Alogrithms.* Reading. MA: Addison-Wesley, 1969.

[Kuck78] Kucs. D. I.. *The Structure of Computers and Computations.* New York: John Wiley & Sons, 1978.

[Lang82] Langdon. G. G.. Jr., *Computer Design.* San Jose, CA: Computeach Press Inc, 1982.

[LiCo83] Lin. S., and D. I. Costello, Ir.. *Error Control Coding, Fundamentals and Applications.* Englewood Cliffs, NJ: Prentice Hall, 1983.

[Mano79] Mano, M. M., *Digital Logic and Computer Design.* Englewood Cliffs. NJ: Prentice Hall. 1979.

[Mano88] Mano, M. M., *Computer Engineering: Hardware Design.* Englewood Cliffs. NJ: Prentice Hall. 1988.

[PeWe72] Peterson. W. Wesley. and E. I. Weldon, Jr., *Error-Correcting Codes.* 2nd Edition. Cambridge. MA: MIT Press, 1972.

[RaFu89] Rao. T. R. N., and E. Fujiwara, *Error-Control Coding for Computer Systems.* Englewood Cliffs, NJ: Prentice Hall. 1989.

[Schn85] Schneider. G. M.. *The Principles of Computer organization:* New York: John Wiley & Sons, 1985.

[Wilk87] Wilkinson, B., *Digital System Design.* Englewood Cliffs, NJ: Prentice Hall International, 1987.