```
    point at extreme right of mantissa.
    if (*ibeta == 2 && !i) --(*maxexp);
    if (i > 20) --(*maxexp);
    if (a != y) *maxexp -= 2;
    *xmax=one-(*epsneg);
    if ((*xmax)*one != *xmax) *xmax=one-beta*(*epsneg);
    *xmax /= (*xmin*beta*beta*beta);
    i=(*maxexp)+(*minexp)+3;
    for (j=1;j<=i;j++) {
        if (*ibeta == 2) *xmax += *xmax;
        else *xmax *= beta;
    }
}
```

Some typical values returned by machar are given in the table, above. IEEE-compliant machines referred to in the table include most UNIX workstations (SUN, DEC, MIPS), and Apple Macintosh IIs. IBM PCs with floating co-processors are generally IEEE-compliant, except that some compilers underflow intermediate results ungracefully, yielding irnd $= 2$ rather than 5. Notice, as in the case of a VAX (fourth column), that representations with a "phantom" leading 1 bit in the mantissa achieve a smaller eps for the same wordlength, but cannot underflow gracefully.

CITED REFERENCES AND FURTHER READING:

Goldberg, D. 1991, *ACM Computing Surveys*, vol. 23, pp. 5–48.

Cody, W.J. 1988, *ACM Transactions on Mathematical Software*, vol. 14, pp. 303–311. [1]

Malcolm, M.A. 1972, *Communications of the ACM*, vol. 15, pp. 949–951. [2]

IEEE Standard for Binary Floating-Point Numbers, ANSI/IEEE Std 754–1985 (New York: IEEE, 1985). [3]

20.2 Gray Codes

A Gray code is a function $G(i)$ of the integers i, that for each integer $N \geq 0$ is one-to-one for $0 \leq i \leq 2^N - 1$, and that has the following remarkable property: The binary representation of $G(i)$ and $G(i+1)$ differ in *exactly one bit*. An example of a Gray code (in fact, the most commonly used one) is the sequence 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, and 1000, for $i = 0, \ldots, 15$. The algorithm for generating this code is simply to form the bitwise exclusive-or (XOR) of i with $i/2$ (integer part). Think about how the carries work when you add one to a number in binary, and you will be able to see why this works. You will also see that $G(i)$ and $G(i+1)$ differ in the bit position of the rightmost zero bit of i (prefixing a leading zero if necessary).

The spelling is "Gray," not "gray": The codes are named after one Frank Gray, who first patented the idea for use in shaft encoders. A shaft encoder is a wheel with concentric coded stripes each of which is "read" by a fixed conducting brush. The idea is to generate a binary code describing the angle of the wheel. The obvious, but wrong, way to build a shaft encoder is to have one stripe (the innermost, say) conducting on half the wheel, but insulating on the other half; the next stripe is conducting in quadrants 1 and 3; the next stripe is conducting in octants 1, 3, 5,

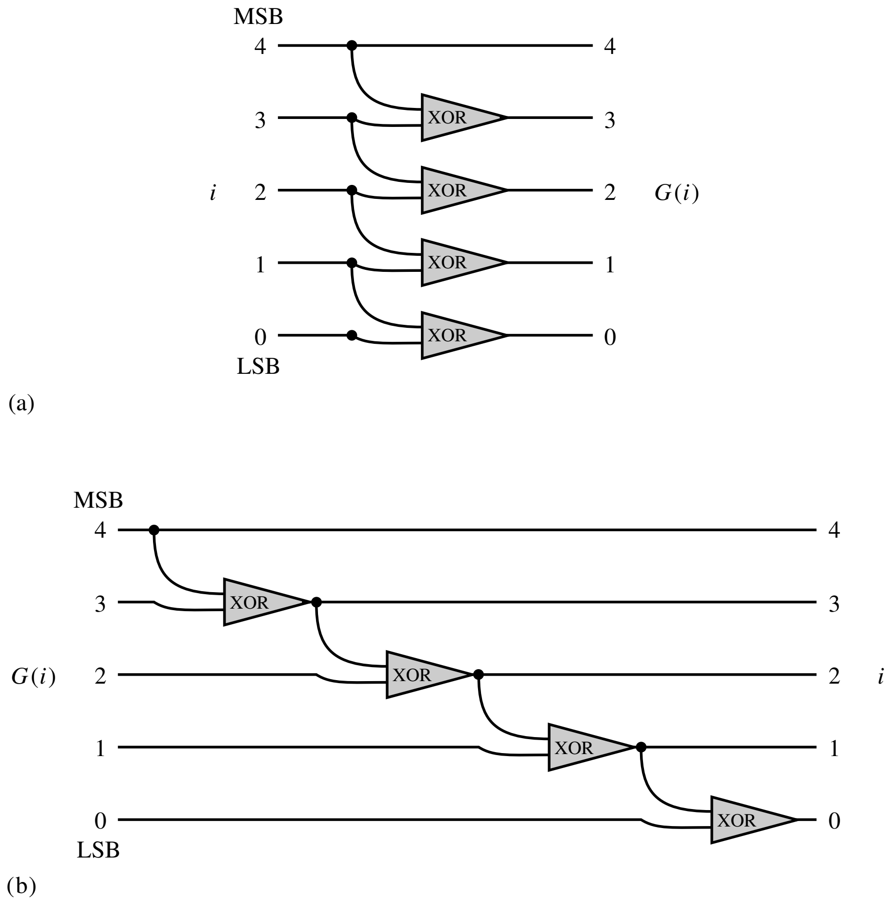

Figure 20.2.1. Single-bit operations for calculating the Gray code $G(i)$ from i (a), or the inverse (b). LSB and MSB indicate the least and most significant bits, respectively. XOR denotes exclusive-or.

and 7; and so on. The brushes together then read a direct binary code for the position of the wheel.

The reason this method is bad, is that there is no way to guarantee that all the brushes will make or break contact *exactly* simultaneously as the wheel turns. Going from position 7 (0111) to 8 (1000), one might pass spuriously and transiently through 6 (0110), 14 (1110), and 10 (1010), as the different brushes make or break contact. Use of a Gray code on the encoding stripes guarantees that there is no transient state between 7 (0100 in the sequence above) and 8 (1100).

Of course we then need circuitry, or algorithmics, to translate from $G(i)$ to i. Figure 20.2.1 (b) shows how this is done by a cascade of XOR gates. The idea is that each output bit should be the XOR of all more significant input bits. To do N bits of Gray code inversion requires $N-1$ steps (or gate delays) in the circuit. (Nevertheless, this is typically very fast in circuitry.) In a register with word-wide binary operations, we don't have to do N consecutive operations, but only $\ln_2 N$. The trick is to use the associativity of XOR and group the operations hierarchically. This involves sequential right-shifts by $1, 2, 4, 8, \dots$ bits until the wordlength is exhausted. Here is a piece of code for doing both $G(i)$ and its inverse.

```
unsigned long igray(unsigned long n, int is)
For zero or positive values of is, return the Gray code of n; if is is negative, return the inverse
Gray code of n.
{
    int ish;
    unsigned long ans,idiv;

    if (is >= 0)                    This is the easy direction!
        return n ^ (n >> 1);
    ish=1;                          This is the more complicated direction: In hierarchical
    ans=n;                              stages, starting with a one-bit right shift, cause each
    for (;;) {                          bit to be XORed with all more significant bits.
        ans ^= (idiv=ans >> ish);
        if (idiv <= 1 || ish == 16) return ans;
        ish <<= 1;                  Double the amount of shift on the next cycle.
    }
}
```

In numerical work, Gray codes can be useful when you need to do some task that depends intimately on the bits of i, looping over many values of i. Then, if there are economies in repeating the task for values differing by only one bit, it makes sense to do things in Gray code order rather than consecutive order. We saw an example of this in §7.7, for the generation of quasi-random sequences.

CITED REFERENCES AND FURTHER READING:

Horowitz, P., and Hill, W. 1989, *The Art of Electronics*, 2nd ed. (New York: Cambridge University Press), §8.02.

Knuth, D.E. *Combinatorial Algorithms*, vol. 4 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §7.2.1. [Unpublished. Will it be always so?]

20.3 Cyclic Redundancy and Other Checksums

When you send a sequence of bits from point A to point B, you want to know that it will arrive without error. A common form of insurance is the “parity bit,” attached to 7-bit ASCII characters to put them into 8-bit format. The parity bit is chosen so as to make the total number of one-bits (versus zero-bits) either always even (“even parity”) or always odd (“odd parity”). Any *single bit* error in a character will thereby be detected. When errors are sufficiently rare, and do not occur closely bunched in time, use of parity provides sufficient error detection.

Unfortunately, in real situations, a single noise “event” is likely to disrupt more than one bit. Since the parity bit has two possible values (0 and 1), it gives, on average, only a 50% chance of detecting an erroneous character with more than one wrong bit. That probability, 50%, is not nearly good enough for most applications. Most communications protocols [1] use a multibit generalization of the parity bit called a “cyclic redundancy check” or CRC. In typical applications the CRC is 16 bits long (two bytes or two characters), so that the chance of a random error going undetected is 1 in $2^{16} = 65536$. Moreover, M-bit CRCs have the mathematical