$2^{2^5} + 1 = 641 \cdot 6700417$

$\mathrm{avg}(x, y) = (x \,\&\, y) + ((x \oplus y) \overset{u}{\gg} 1)$

$2^{2^6} + 1 = 274177 \cdot 67280421310721$

$x - y = x + \bar{y} + 1$

$\lfloor a \rfloor + \lfloor b \rfloor \le \lfloor a + b \rfloor \le \lfloor a \rfloor + \lfloor b \rfloor + 1$

$\mathrm{pop}(x) = -\sum_{i=0}^{31} (x \overset{rot}{\ll} i)$

George Boole
1815 - 1864

$\lfloor \sqrt{11111111} \rfloor = 1111$

$(x \ne 0) = (x \,|\, {-x}) \overset{u}{\gg} 31$

$\mathrm{mux}(x, y, m) = ((x \oplus y) \,\&\, m) \oplus y$

$A(n, d) = A(n-1, d-1)$, d even

$-\bar{x} = x + 1$

Hacker's Delight

$\frac{1}{3} = 0.01010101\ldots$

Second Edition

$1111^2 = 11100001$

$n = -2^{31} b_{31} + 2^{30} b_{30} + 2^{29} b_{29} + \ldots + 2^0 b_0$

$\lceil x \rceil = -\lfloor -x \rfloor$

$f(x, y, z) = g(x, y) \oplus z h(x, y)$

Num factors of 2 in $x = \log_2(x \,\&\, (-x)),\ x \ne 0$

$\mathrm{rjust}(x) = x \overset{u}{\div} (x \,\&\, -x),\ x \ne 0$

$p_n = 1 + \sum_{m=1}^{2^n} \left\lfloor \sqrt[n]{n} \left\lfloor \sum_{x=1}^{m} \left\lfloor \cos^2 \pi \frac{(x-1)! + 1}{x} \right\rfloor \right\rfloor^{-1/n} \right\rfloor$

$x \oplus y = (x \,|\, y) - (x \,\&\, y)$

$x + y = (x \,|\, y) + (x \,\&\, y)$

Henry S. Warren, Jr.

Hacker's Delight

The second multiplication can be avoided, because the product is equal to the first product shifted right six positions. The last mask is equal to the second mask shifted right eight positions. With these simplifications, this requires 12 basic RISC instructions, including one *multiply* and one *remainder*. The *remainder* operation must be unsigned, and it cannot be changed to a *multiply* and *shift*.

The reader who studies these marvels will be able to devise similar code for other bit-permuting operations. As a simple (and artificial) example, suppose it is desired to extract every other bit from an 8-bit quantity and compress the four bits to the right. That is, the desired transformation is

```
0000 0000 0000 0000 0000 0000 abcd efgh ==>
0000 0000 0000 0000 0000 0000 0000 bdfh
```

This can be computed as follows:

$$t \leftarrow (x * \mathbf{0x01010101})\ \&\ \mathbf{0x40100401}$$

$$(t * \mathbf{0x08040201}) \overset{u}{\gg} 27$$

On most machines, the most practical way to do all these operations is by indexing into a table of 1-byte (or 9-bit) integers.

Incrementing a Reversed Integer

The Fast Fourier Transform (FFT) algorithm employs an integer i and its bit reversal rev(i) in a loop in which i is incremented by 1 [PuBr]. Straightforward coding would increment i and then compute rev(i) on each loop iteration. For small integers, computing rev(i) by table lookup is fast and practical. For large integers, however, table lookup is not practical and, as we have seen, computing rev(i) requires some 29 instructions.

If table lookup cannot be used, it is more efficient to maintain i in both normal and bit-reversed forms, incrementing them both on each loop iteration. This raises the question of how best to increment an integer that is in a register in reversed form. To illustrate, on a 4-bit machine we wish to successively step through the values (in hexadecimal)

```
0, 8, 4, C, 2, A, 6, E, 1, 9, 5, D, 3, B, 7, F.
```

In the FFT algorithm, i and its reversal are both some specific number of bits in length, almost certainly less than 32, and they are both right-justified in the register. However, we assume here that i is a 32-bit integer. After adding 1 to the reversed 32-bit integer, a *shift right* of the appropriate number of bits will make the result usable by the FFT algorithm (both i and rev(i) are used to index an array in memory).

The straightforward way to increment a reversed integer is to scan from the left for the first 0-bit, set it to 1, and set all bits to the left of it (if any) to 0's. One way to code this is

```
unsigned x, m;

m = 0x80000000;
x = x ^ m;
if ((int)x >= 0) {
   do {
      m = m >> 1;
      x = x ^ m;
   } while (x < m);
}
```

This executes in three basic RISC instructions if `x` begins with a 0-bit, and four additional instructions for each loop iteration. Because `x` begins with a 0-bit half the time, with 10 (binary) one-fourth of the time, and so on, the average number of instructions executed is approximately

$$\begin{aligned}
&3\cdot\frac{1}{2}+7\cdot\frac{1}{4}+11\cdot\frac{1}{8}+15\cdot\frac{1}{16}+\dots\\
&=4\cdot\frac{1}{2}+8\cdot\frac{1}{4}+12\cdot\frac{1}{8}+16\cdot\frac{1}{16}+\dots-1\\
&=4\left(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\dots\right)-1\\
&=7.
\end{aligned}$$

In the second line we added and subtracted 1, with the first 1 in the form $1/2+1/4+1/8+1/16+\dots$. This makes the series similar to the one analyzed on page 113. The number of instructions executed in the worst case, however, is quite large (131).

If *number of leading zeros* is available, adding 1 to a reversed integer can be done as follows:

First execute: $s \leftarrow \text{nlz}(\neg x)$

and then either: $x \leftarrow x \oplus (\mathbf{0x80000000} \overset{s}{\gg} s)$

or: $x \leftarrow ((x \ll s) + \mathbf{0x80000000}) \overset{u}{\gg} s$

Either method requires five full RISC instructions and, to properly wrap around from **0xFFFFFFFF** to **0**, requires that the shifts be modulo 64. (These formulas fail in this respect on the Intel x86 machines, because the shifts are modulo 32.)

The rather puzzling one-liner below [Möbi] increments a reversed integer in six basic RISC instructions. It is free of branches and loads but includes an integer division operation. It works for integers of length up to that of the word size of the machine, less 1.

$$\boldsymbol{revi} \leftarrow \boldsymbol{revi} \oplus \left(\boldsymbol{m} - \frac{\boldsymbol{m}}{(\boldsymbol{i} \oplus (\boldsymbol{i}+\mathbf{1})) + \mathbf{1}}\right)$$

To use this, both the non-reversed integer ***i*** and its reversal ***revi*** must be available. The variable ***m*** is the modulus; if we are dealing with n-bit integers, then $\boldsymbol{m} = 2^n$. Applying the formula gives the next value of the reversed integer. The non-reversed integer ***i*** would be incremented separately. The reversed integer is incremented "in place"; that is, it is not shifted to the high-order end of the register, as in the two preceding methods.

A variation is

$$\boldsymbol{revi} \leftarrow \boldsymbol{revi} \oplus \left(\boldsymbol{m} - \frac{\boldsymbol{m}/\mathbf{2}}{\neg\boldsymbol{i} \mathbin{\&} (\boldsymbol{i}+\mathbf{1})}\right), \qquad (1)$$

which executes in five instructions if the machine has *and not*, and if ***m*** is a constant so that the calculation of $\boldsymbol{m}/\mathbf{2}$ does not count. It works for integers of length up to that of the word size of the machine. (For full word-size integers, use 0 for the first occurrence of ***m*** in the formula, and 2^{n-1} for $\boldsymbol{m}/\mathbf{2}$.)

7–2 Shuffling Bits

Another important permutation of the bits of a word is the "perfect shuffle" operation, which has applications in cryptography. There are two varieties, called the "outer" and "inner" perfect shuffles. They both interleave the bits in the two halves of a word in a manner similar to a perfect shuffle of a deck of 32 cards, but they differ in which card is allowed to fall first. In the outer perfect shuffle, the outer (end) bits remain in the outer positions, and in the inner perfect shuffle, bit 15 moves to the left end of the word (position 31). If the 32-bit word is (where each letter denotes a single bit)

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP,
```

then after the outer perfect shuffle it is

```
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP,
```

and after the inner perfect shuffle it is

```
AaBb CcDd EeFf GgHh IiJj KkLl MmNn OoPp.
```

Assume the word size W is a power of 2. Then the outer perfect shuffle operation can be accomplished with basic RISC instructions in $\log_2(W/2)$ steps, where each step swaps the second and third quartiles of successively smaller pieces [GLS1]. That is, a 32-bit word is transformed as follows:

```
abcd efgh ijkl mnop ABCD EFGH IJKL MNOP
abcd efgh ABCD EFGH ijkl mnop IJKL MNOP
abcd ABCD efgh EFGH ijkl IJKL mnop MNOP
abAB cdCD efEF ghGH ijIJ klKL mnMN opOP
aAbB cCdD eEfF gGhH iIjJ kKlL mMnN oOpP
```

Straightforward code for this is

```
x = (x & 0x0000FF00) << 8 | (x >> 8) & 0x0000FF00 | x & 0xFF0000FF;
x = (x & 0x00F000F0) << 4 | (x >> 4) & 0x00F000F0 | x & 0xF00FF00F;
x = (x & 0x0C0C0C0C) << 2 | (x >> 2) & 0x0C0C0C0C | x & 0xC3C3C3C3;
x = (x & 0x22222222) << 1 | (x >> 1) & 0x22222222 | x & 0x99999999;
```

which requires 42 basic RISC instructions. This can be reduced to 30 instructions, although at an increase from 17 to 21 cycles on a machine with unlimited instruction-level parallelism, by using the *exclusive or* method of exchanging two fields of a register (described on page 47). All quantities are unsigned:

```
t = (x ^ (x >> 8)) & 0x0000FF00;  x = x ^ t ^ (t << 8);
t = (x ^ (x >> 4)) & 0x00F000F0;  x = x ^ t ^ (t << 4);
t = (x ^ (x >> 2)) & 0x0C0C0C0C;  x = x ^ t ^ (t << 2);
t = (x ^ (x >> 1)) & 0x22222222;  x = x ^ t ^ (t << 1);
```

The inverse operation, the outer unshuffle, is easily accomplished by performing the swaps in reverse order:

```
t = (x ^ (x >> 1)) & 0x22222222;  x = x ^ t ^ (t << 1);
t = (x ^ (x >> 2)) & 0x0C0C0C0C;  x = x ^ t ^ (t << 2);
t = (x ^ (x >> 4)) & 0x00F000F0;  x = x ^ t ^ (t << 4);
t = (x ^ (x >> 8)) & 0x0000FF00;  x = x ^ t ^ (t << 8);
```

Using only the last two steps of either of the above two shuffle sequences shuffles the bits of each byte separately. Using only the last three steps shuffles the bits of each halfword separately, and so on. Similar remarks apply to unshuffling, except by using the *first* two or three steps.

To get the inner perfect shuffle, prepend to these sequences a step to swap the left and right halves of the register:

```
x = (x >> 16) | (x << 16);
```

(or use a *rotate* of 16 bit positions). The unshuffle sequence can be similarly modified by *appending* this line of code.

Altering the transformation to swap the *first* and *fourth* quartiles of successively smaller pieces produces the bit reversal of the inner perfect shuffle of 2^n bits for odd n, and the bit reversal of the outer perfect shuffle for even n.

Perhaps worth mentioning is the special case in which the left half of the word `x` is all 0. In other words, we want to move the bits in the right half of `x` to every other bit position—that is, to transform the 32-bit word

```
0000 0000 0000 0000 ABCD EFGH IJKL MNOP
```

to

```
0A0B 0C0D 0E0F 0G0H 0I0J 0K0L 0M0N 0O0P.
```

The outer perfect shuffle code can be simplified to do this task in 22 basic RISC instructions. The code below, however, does it in only 19, at no cost in execution time on a machine with unlimited instruction-level parallelism (12 cycles with either method). This code does not require that the left half of word `x` be initially cleared.

```
x = ((x & 0xFF00) << 8) | (x & 0x00FF);
x = ((x << 4) | x) & 0x0F0F0F0F;
x = ((x << 2) | x) & 0x33333333;
x = ((x << 1) | x) & 0x55555555;
```

Similarly, for the inverse of this “half shuffle” operation (a special case of *compress*; see page 150), the outer perfect unshuffle code can be simplified to do the task in 26 or 29 basic RISC instructions, depending on whether or not an initial *and* operation is required to clear the bits in the odd positions. The code below, however, does it in only 18 or 21 basic RISC instructions, and with less execution time on a machine with unlimited instruction-level parallelism (12 or 15 cycles).

```
x = x & 0x55555555;          // (If required.)
x = ((x >> 1) | x) & 0x33333333;
x = ((x >> 2) | x) & 0x0F0F0F0F;
x = ((x >> 4) | x) & 0x00FF00FF;
x = ((x >> 8) | x) & 0x0000FFFF;
```

7–3 Transposing a Bit Matrix

The transpose of a matrix A is a matrix whose columns are the rows of A and whose rows are the columns of A. Here we consider the problem of computing the transpose of a bit matrix whose elements are single bits that are packed eight per byte, with rows and columns beginning on byte boundaries. This seemingly simple transformation is surprisingly costly in instructions executed.

On most machines it would be very slow to load and store individual bits, mainly due to the code that would be required to extract and (worse yet) to store individual bits. A better method is to partition the matrix into 8×8 submatrices, load each 8×8 submatrix into registers, compute the transpose of the submatrix in registers, and then store the 8×8 result in the appropriate place in the target matrix. Figure 7–5 illustrates the transposition of a bit matrix of size 16×3 bytes. A, B, …, F are submatrices of size 8×8 bits. A^T, B^T, … denote the transpose of submatrices A, B, ….

For the purposes of transposing an 8×8 submatrix, it doesn't matter whether the bit matrix is stored in row-major or column-major order; the operations are the same in either event. Assume for discussion that it's in row-major order. Then the first byte of the matrix contains the top row of A, the next byte contains the top row of B, and so on. If L denotes the address of the first byte (top row) of a submatrix, then successive rows of the submatrix are at locations $L + n$, $L + 2n$, …, $L + 7n$.

For this problem we will depart from the usual assumption of a 32-bit machine and assume the machine has 64-bit general registers. The algorithms are simpler and more easily understood in this way, and it is not difficult to convert them for execution on a 32-bit machine. In fact, a compiler that supports 64-bit integer operations on a 32-bit machine will do the work for you (although probably not as effectively as you can do by hand).

The overall scheme is to load a submatrix with eight *load byte* instructions and pack the bytes left-to-right into a 64-bit register. Then the transpose of the register's contents is computed. Finally, the result is stored in the target area with eight *store byte* instructions.

The transposition of an 8×8 bit matrix is illustrated here, where each character represents a single bit.

```
0123 4567          08go wEMU
89ab cdef          19hp xFNV
ghij klmn          2aiq yGOW
opqr stuv    ⇒     3bjr zHPX
wxyz ABCD          4cks AIQY
EFGH IJKL          5dlt BJRZ
MNOP QRST          6emu CKS$
UVWX YZ$.          7fnv DLT.
```

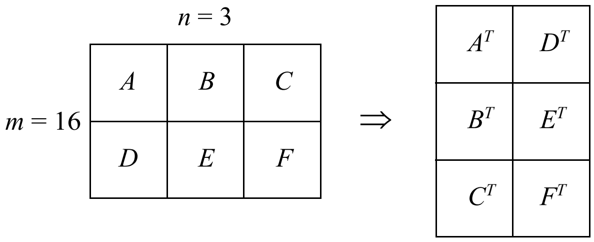

FIGURE 7–5. Transposing a 16×24-bit matrix.