The FPC Double-Precision Floating-Point Compression Algorithm and its Implementation

Martin Burtscher
Center for Grid and Distributed Computing
The University of Texas at Austin
burtscher@ices.utexas.edu

Paruj Ratanaworabhan
Computer Systems Laboratory
Cornell University
paruj@csl.cornell.edu

1. Preamble

This document provides detailed information about the design and our C implementation of the FPC compression algorithm. It is meant as an addendum to and contains excerpts from two other publications about FPC [1], [2]. Please refer to these publications for additional information such as performance results and related work.

2. Algorithm

FPC compresses linear sequences of IEEE 754 double-precision floating-point values by sequentially predicting each value, xoring the true value with the predicted value, and leading-zero compressing the result. As illustrated in Figure 1, it uses variants of an *fcm* and a *dfcm* value predictor to predict the doubles. Both predictors are effectively hash tables. The more accurate of the two predictions, i.e., the one that shares more common most significant bits with the true value, is xored with the true value. Xor turns identical bits into zeros. Hence, if the predicted and the true value are close, the xor result has many leading zeros. FPC then counts the number of leading zero bytes, encodes the count in a three-bit value, and concatenates it with a single bit that specifies which of the two predictions was used. The resulting four-bit code and the nonzero residual bytes are written to the output. The latter are emitted verbatim without any encoding.

FPC outputs the compressed data in blocks. Each block starts with a header that specifies how many doubles the block encodes and how long it is (in bytes). The header is followed by the four-bit codes, which in turn are followed by the residual bytes. Keeping the four-bit codes and the residual bytes separate instead of interleaving them makes FPC faster and potentially simplifies post-processing of the output (e.g., adding another compression stage).

To maintain byte granularity, which is more efficient than bit granularity, a pair of doubles is always processed together and the corresponding two four-bit codes are packed into a byte. In case an odd number of doubles needs to be compressed, a spurious double is encoded at the end. This spurious value is later eliminated using the count information from the header.

Decompression works by reading the current four-bit code, decoding the three-bit field, reading the specified number of residual bytes, and zero-extending them to a full 64-bit number. Based on the one-bit field, it xors this number with either the 64-bit *fcm* or *dfcm* prediction to recreate the original double. This lossless reconstruction is possible because xor is reversible.

For performance reasons, FPC interprets all doubles as 64-bit integers and uses only integer arithmetic. Since there can be between zero and eight leading zero bytes, i.e., nine possibilities, not all of them can be encoded with a three-bit value. We decided not to support a leading zero count of four because it occurs only rarely. Consequently, all xor results with four leading zero

bytes are treated like values with only three leading zero bytes and the fourth zero byte is emitted as part of the residual.

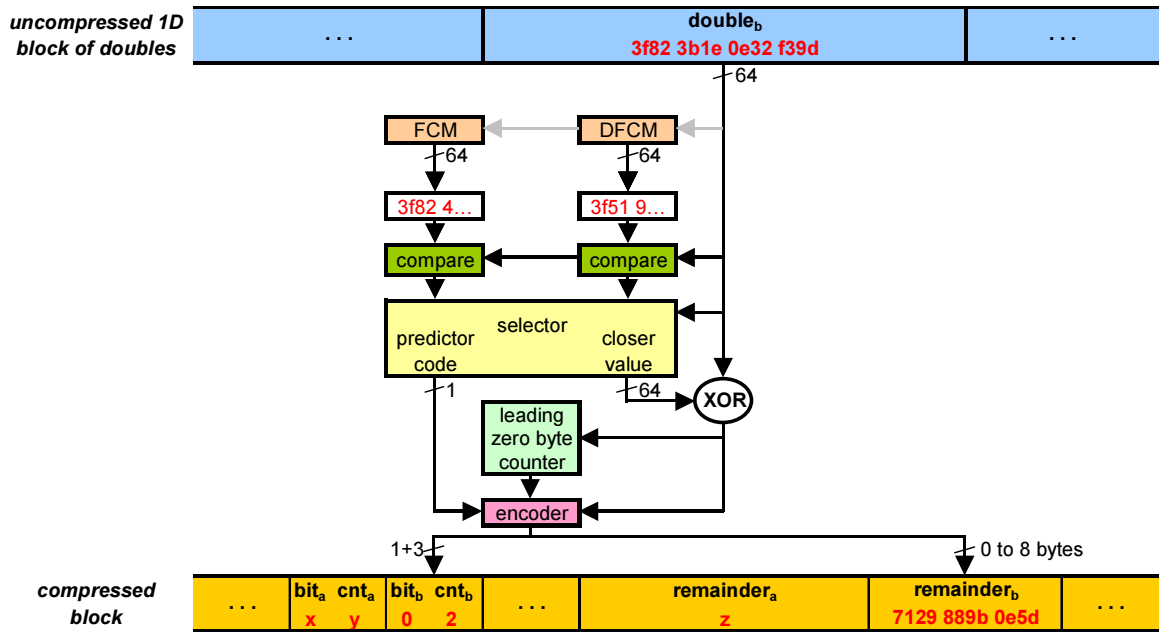

Figure 1: FPC compression algorithm overview

Before compression and decompression, both predictor tables are initialized with zeros. After each prediction, they are updated with the true double value to ensure that they generate the same sequence of predictions during compression as they do during decompression. The following pseudo code demonstrates the operation of the *fcm* predictor. The `table_size` has to be a power of two. `fcm` is the hash table.

```
unsigned long long true_value, fcm_prediction, fcm_hash, fcm[table_size];
...
fcm_prediction = fcm[fcm_hash];  // prediction: read hash table entry
fcm[fcm_hash] = true_value;      // update: write hash table entry
fcm_hash = ((fcm_hash << 6) ^ (true_value >> 48)) & (table_size - 1);
```

Right shifting `true_value` (i.e., the current double expressed as a 64-bit integer) by 48 bits eliminates the often random mantissa bits. The remaining 16 bits are xored with the previous hash value to produce the new hash. However, the previous hash is first shifted by six bits to the left to gradually phase out bits from older values. The hash value (`fcm_hash`) therefore represents the sequence of most recently encountered doubles, and the hash table stores the double that follows this sequence. Hence, making an *fcm* prediction is tantamount to performing a table lookup to determine which value followed the last time a similar sequence of previous doubles was seen.

The *dfcm* predictor operates in the same way. However, it predicts integer differences between consecutive values rather than absolute values, and the shift amounts in the hash function are different.

```
unsigned long long last_value, dfcm_prediction, dfcm_hash, dfcm[table_size];
...
dfcm_prediction = dfcm[dfcm_hash] + last_value;
dfcm[dfcm_hash] = true_value - last_value;
dfcm_hash = ((dfcm_hash << 2) ^ ((true_value - last_value) >> 40)) &
  (table_size - 1);
last_value = true_value;
```

Note that the two predictors were designed to complement each other. Neither one of them performs particularly well when used in isolation. The complete C source code is provided in the Appendix of this paper. A downloadable version of the code and a brief description of how to compile and use it, as well as sample datasets, are available at the following URL.

http://www.csl.cornell.edu/~burtscher/research/FPC/

3. Design

FPC's primary objective is to maximize the throughput while still delivering a competitive compression ratio. Therefore, FPC does not include features that improve the compression ratio at a significant cost of speed. For example, we deemed extracting and handling the sign, exponent and mantissa separately to be too slow for throughput-oriented compression. Likewise, we excluded variable-length encoding at bit granularity as well as bit reversal because of their inefficiency on modern CPUs. Furthermore, we replaced all floating-point arithmetic with integer arithmetic. Even though the former is more natural and sometimes results in better compression ratios, it is slower and, more importantly, may cause exceptions.

Our previous experience with fast lossless compressors demonstrated algorithms that predict the data using value predictors and leading-zero compress the residual to be very fast while offering a good compression ratio. Hence, we based FPC on this approach. We considered both subtraction and xoring for the residual generation. Since subtraction both with a two's complement and with a sign-magnitude representation yields a lower compression ratio and a lower processing speed, we abandoned subtraction and selected xor.

3.1 Predictor Parameter Selection

Value predictors have been researched extensively to predict the results of CPU machine instructions at runtime. These predictors are designed to make billions of predictions per second in hardware. As a consequence, they employ simple and fast prediction algorithms.

First, we had to determine which and how many (software) predictors to use. As one might expect, the more accurate prediction algorithms tend to be slower. Similarly, employing a larger number of predictors increases the probability of one of them being correct but lowers the throughput. We experimented with many combinations and configurations of four basic value predictors (a last value, a stride, a finite context method, and a differential finite context method predictor) as well as variations thereof (including a last *n* value and a stride 2-delta predictor).

Because a high processing speed was paramount in our design, we soon found two-predictor combinations to represent the best tradeoff for the following reasons. First, adding predictors increases the runtime linearly but quickly yields diminishing returns on the gained compression ratio. Therefore, only few predictors should be used. Second, to achieve high performance, we had to operate at least at byte granularity. Consequently, we were faced with three-bit codes to express the number of leading zero bytes of the residual between the predicted and the true val-

ue. That left five bits to select one of 32 predictors, which was far beyond the number of predictors we could reasonably employ. The only good alternative, which we ended up choosing, was to utilize one bit to pick between two predictors. Concatenating this bit with the three-bit leading zero count resulted in a four-bit field, which can be combined with the four-bit field of the next prediction to form a byte. (Four-predictor combinations together with two-bit codes for expressing the leading zero counts result in poor compression ratios.)

The next question was which two predictors to select. Initially, we evaluated single predictors with different configurations in isolation and paired up the best performers. However, this approach ended up combining predictors that largely made the same predictions. So we switched to evaluating predictor pairs rather than single predictors, i.e., we optimized the algorithm as a whole instead of its individual components. The result was a significant boost in compression ratio without loss in throughput. Note that the predictors making up the best pairs do not perform particularly well when used in isolation, but they complement each other nicely.

The two-predictor experiments revealed that we should combine an *fcm* predictor with a *dfcm* predictor. That left us with determining good parameters for these predictors. For speed reasons and to prevent overfitting to our test datasets, we opted to hardcode the parameters and use the same fixed set of parameters for all predictor sizes. To determine the best configuration, we evaluated the following combinations of table sizes and shift amounts in the two hash functions:

Number of table entries: 1024, 32768, 1048576
Left shift in *fcm* hash function: 1, 2, 3, 4, 5, 6, 7, 8
Right shift in *fcm* hash function: 8, 16, 24, 32, 40, 48, 56
Left shift in *dfcm* hash function: 1, 2, 3, 4, 5, 6, 7, 8
Right shift in *dfcm* hash function: 8, 16, 24, 32, 40, 48, 56

Next, we performed a local search to refine the best right-shift amounts. Unfortunately, no clear winners could be identified because different datasets prefer different configurations and large predictors work well with settings that are suboptimal for small predictors and vice versa. In the end, we settled for the parameters listed in the previous section, which perform reasonable in most cases and work well in the mid range of table sizes.

Because FPC runs at the same speed for all table sizes that fit into the L1 data cache (cf. Section 4.3) but compresses better with larger tables, there is little reason to use it with very small tables (e.g., less than half of the L1 data cache size). Hence, we were not overly concerned with our parameter choices resulting in poor compression at the low end. Nevertheless, for our datasets, the most important change to improve the compression ratio with small tables is to increase the right-shift amount in *dfcm* from forty to a value in the fifties. At the high end, better hash functions are obtained by lowering the left-shift amount in *fcm* to between two and four, increasing the *dfcm* left-shift amount to between four and eight, and lowering the *dfcm* right-shift amount to 32.

4. Implementation

We co-designed the FPC algorithm and its C implementation to achieve both a high throughput and a high compression ratio. The source code (cf. Appendix) includes the following features to help make it efficient.

- The compressor and decompressor are each written as a single static function, which allows the compiler to perform aggressive optimizations.
- There are no global variables that could restrict the optimization potential.
- All variables except for the two hash tables, the input buffer, and the output buffer are local scalars and are declared with the "register" keyword to inform the compiler that they should be register allocated and that they cannot be accessed through pointers.
- Even though the code compresses floating-point values, it exclusively uses integer variables and operations. Moreover, the code contains no multiplication or division operations. As a result, only fast, low-latency integer machine instructions are executed.
- When accessing hash table entries, the code first copies the entries into scalar variables and then operates exclusively on these copies. Moreover, the code is written in such a way that all uses immediately precede the next hash table read. Thus, the rest of the loop body hides (some of) the read latency associated with the hash table accesses. For the other non-scalar accesses, i.e., to the input and output buffers, the latency is not much of a concern. Because these two buffers are read and written sequentially, the CPU's prefetching and caching mechanisms are effective at hiding the access latency.
- The algorithm operates on a block of data at a time with a fixed block size. This speeds up the implementation because it allows the use of efficient block I/O calls to transfer the data.
- The compression and the decompression function each contain one critical loop that processes a block of data. These loops account for about 90% of the total runtime, making them the most performance critical code. Since the loops contain no function calls, the compiler does not have to worry about side effects, calling conventions, etc. when optimizing them.
- All IF statement bodies in the two loops exclusively assign constants to scalar variables or copy one scalar variable into another. This is important because it enables the compiler to use conditional move instructions instead of conditional branches. (Conditional branches tend to be slow in compressors because they are often input data dependent and can therefore be hard to predict.) As a result, all of the about 50 and 70 C statements in the two loop bodies can be compiled into a single basic block on architectures that support conditional moves. This makes instruction fetching fast (the loop end branch is highly predictable) and is great for code scheduling as it allows the compiler to hide latencies and to expose the ILP.

4.1 Compression Code

The line numbers in this and the next section refer to the code listing in the Appendix. Invoking FPC with one command-line parameter (the binary logarithm of the desired number of hash table entries) launches the compressor, which reads the uncompressed data from the standard input and writes the compressed data to the standard output. It works as follows.

First, the compressor writes one byte to the output to record the hash table size (lines 27-29). Then it performs initializations (lines 30-36), allocates the two hash tables and zeros them out (lines 37 and 39). Next it reads in the first block of data (line 42) and enters the outer loop (line 43), which performs more initializations (lines 44-46), runs the inner loop to compress the current block of data (lines 47-131), adds a header with size information (lines 135-140), writes out the compressed data (line 141), reads the next uncompressed block (line 143), and repeats the outer loop as long as there are more data to process. Note that variable *i* indicates which double of the current block is being processed. The variable *out* specifies the current byte index for writing the compressed data. It is initialized (line 45) to leave space at the beginning of the output buffer for the six-byte header and for as many four-bit fields as there are doubles in the block.

Each iteration of the inner loop compresses a pair of double-precision values. For efficiency reasons, the loop does not check whether the current block contains an odd number of doubles. Instead, the second double, which may be garbage, is compressed speculatively. Corrective actions are only taken after leaving the loop (lines 132-134). The inner loop works as follows.

First, it xors the first prediction with the true value (line 48), updates the first hash table (line 49), computes the next hash (line 50), and retrieves the next prediction (line 51). Second, the same steps are performed with the second prediction and hash table (lines 53-59). Third, the smaller of the two xor results is selected (lines 61-65), i.e., the one with more leading zero bits. Fourth, a three-bit value is computed that encodes the number of leading zero bytes (lines 66-80). Fifth, the entire 64-bit xor result is written to the output buffer (lines 82-85). One read, two writes, and a few shifts and logic operations are necessary to accomplish this action because of alignment reasons. Note, however, that always writing all eight bytes is substantially faster than skipping the leading zero bytes, which would necessitate conditional (IF, SWITCH, or WHILE) statements. Furthermore, these extra byte writes zero out the bytes up to the next eight-byte boundary in the output buffer, which is necessary for the correct operation of the following iteration. Note that this trick only works on little-endian machines. On big-endian machines, the array would have to be filled from high to low addresses to make the algorithm work. Sixth, the current write position is advanced by the actual number of "non-zero" bytes (line 87). This way, the leading zero bytes will be overwritten in the next iteration. Seventh, a four-bit number is formed (line 88) out of the one-bit value that specifies which of the two xor values was used and the three-bit value that encodes the number of leading zero bytes. Eighth, steps one through seven are repeated for the second double (lines 90-129). Ninth, the four-bit codes from the first and the second double are combined into a byte, which is written to the output buffer (line 130).

4.2 Decompression Code

Invoking FPC without a command-line parameter launches the decompressor, which reads the compressed data from the standard input and writes the decompressed data to the standard output. The following steps illustrate its operation.

First, the decompressor reads up to seven bytes from the input to obtain the hash table size and the header information of the first block (lines 157-161). Then it initializes variables (lines 163-167 and 173-178) and allocates the two hash tables and zeros them out (lines 168 and 170) before entering the outer loop (line 180). The outer loop reads a block of compressed data (line 181) and, if available, the header information of the next block. Then it runs the inner loop to decompress the block (lines 184-244), outputs the decompressed data (line 245), computes the size of the next block from the header information (lines 247-256), and repeats the outer loop until there are no more input data (line 258).

The inner loop decompresses one pair of double-precision values per iteration. Again, the loop body does not check whether the current block contains an odd number of compressed doubles. In case of an odd number, it simply decompresses a spurious double at the end, which will later be suppressed. The inner loop operates as follows.

First, it obtains the two four-bit codes for the current pair of doubles (line 185). Then it extracts the next eight-byte value (lines 187-194). Due to alignment reasons, this requires two memory accesses, an add, and a few shifts. Depending on the four-bit code, the appropriate number of leading zero bytes is inserted (lines 196-197). It is faster to always read eight bytes and mask out the unnecessary bytes than reading a variable number of bytes. The four-bit code also informs the decompressor about which of the two hash table values the eight-byte value has

to be xored with to regenerate the original double (lines 200-202). This decompressed value is then written to the output buffer (line 214) and used to update the two hash tables and to compute the new hashes (lines 204-205 and 208-210). Then the next predictions are obtained (lines 206 and 211). Finally, the above steps are repeated to decompress the second double (lines 216-243).

4.3 Performance Notes

The inner loops compress and decompress two doubles per iteration to maintain byte granularity in spite of the four-bit codes. As a consequence, sometimes an extra double is processed unnecessarily. However, this happens at most once at the end of the input and therefore represents a negligible overhead, which is more than compensated for by not having to check for an odd block size during each iteration.

Aside from memory accesses, integer subtraction and shifts are the most complicated operations in these loops. Other than the loop-end branch, most systems execute the compress and decompress loop without any control transferring instructions because the IF statements are converted into conditional move instructions, which makes them very efficient.

The fact that the two critical loop bodies are single basic blocks has an important implication. The exact same sequence of instructions is executed to compress (decompress) a block of n doubles regardless of the data values or their compressibility. The running time of these loops, which account for most of the total runtime, is therefore only dependent on the load latency, as all other instructions have fixed latencies. In other words, as long as the hash tables fit in the L1 data cache, the compression and the decompression time for a block of data are constant no matter what data are being processed and what compression ratio is being achieved. This rather unusual feature, which most other compression algorithms do not possess, is required in real-time environments.

5. Acknowledgements

This project is supported by the Department of Energy under Award DE-FG02-06ER25722.

6. References

[1] M. Burtscher and P. Ratanaworabhan. “FPC: A High-Speed Compressor for Double-Precision Floating-Point Data.” *IEEE Transactions on Computers*, to appear. 2008.

[2] M. Burtscher and P. Ratanaworabhan. “High Throughput Compression of Double-Precision Floating-Point Data.” *2007 Data Compression Conference*, pp. 293-302. March 2007.

7. Appendix

The C source code of FPC, listed below, is also available on-line at http://www.csl.cornell.edu/~burtscher/research/FPC/.

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

#define SIZE 32768

static const long long mask[8] =
{0x0000000000000000LL,
```

```
 9  0x00000000000000ffLL,
10  0x000000000000ffffLL,
11  0x0000000000ffffffLL,
12  0x00000fffffffffffLL,
13  0x0000ffffffffffffLL,
14  0x00ffffffffffffffLL,
15  0xffffffffffffffffLL};
16
17 static void Compress(long predsizem1)
18 {
19   register long i, out, intot, hash, dhash, code, bcode, ioc;
20   register long long val, lastval, stride, pred1, pred2, xor1, xor2;
21   register long long *fcm, *dfcm;
22   unsigned long long inbuf[SIZE + 1];
23   unsigned char outbuf[6 + (SIZE / 2) + (SIZE * 8) + 2];
24
25   assert(0 == ((long)outbuf & 0x7));
26
27   outbuf[0] = predsizem1;
28   ioc = fwrite(outbuf, 1, 1, stdout);
29   assert(1 == ioc);
30   predsizem1 = (1L << predsizem1) - 1;
31
32   hash = 0;
33   dhash = 0;
34   lastval = 0;
35   pred1 = 0;
36   pred2 = 0;
37   fcm = (long long *)calloc(predsizem1 + 1, 8);
38   assert(NULL != fcm);
39   dfcm = (long long *)calloc(predsizem1 + 1, 8);
40   assert(NULL != dfcm);
41
42   intot = fread(inbuf, 8, SIZE, stdin);
43   while (0 < intot) {
44     val = inbuf[0];
45     out = 6 + ((intot + 1) >> 1);
46     *((long long *)&outbuf[(out >> 3) << 3]) = 0;
47     for (i = 0; i < intot; i += 2) {
48       xor1 = val ^ pred1;
49       fcm[hash] = val;
50       hash = ((hash << 6) ^ ((unsigned long long)val >> 48)) & predsizem1;
51       pred1 = fcm[hash];
52
53       stride = val - lastval;
54       xor2 = val ^ (lastval + pred2);
55       lastval = val;
56       val = inbuf[i + 1];
57       dfcm[dhash] = stride;
58       dhash = ((dhash << 2) ^ ((unsigned long long)stride >> 40)) & predsizem1;
59       pred2 = dfcm[dhash];
60
61       code = 0;
62       if ((unsigned long long)xor1 > (unsigned long long)xor2) {
63         code = 0x80;
64         xor1 = xor2;
65       }
66       bcode = 7;                 // 8 bytes
67       if (0 == (xor1 >> 56))
68         bcode = 6;               // 7 bytes
69       if (0 == (xor1 >> 48))
70         bcode = 5;               // 6 bytes
71       if (0 == (xor1 >> 40))
```

```
 72        bcode = 4;              // 5 bytes
 73      if (0 == (xor1 >> 24))
 74        bcode = 3;              // 3 bytes
 75      if (0 == (xor1 >> 16))
 76        bcode = 2;              // 2 bytes
 77      if (0 == (xor1 >> 8))
 78        bcode = 1;              // 1 byte
 79      if (0 == xor1)
 80        bcode = 0;              // 0 bytes
 81
 82      *((long long *)&outbuf[(out >> 3) << 3]) |= xor1 << ((out & 0x7) << 3);
 83      if (0 == (out & 0x7))
 84        xor1 = 0;
 85      *((long long *)&outbuf[((out >> 3) << 3) + 8]) = (unsigned long long)xor1 >>
         (64 - ((out & 0x7) << 3));
 86
 87      out += bcode + (bcode >> 2);
 88      code |= bcode << 4;
 89
 90      xor1 = val ^ pred1;
 91      fcm[hash] = val;
 92      hash = ((hash << 6) ^ ((unsigned long long)val >> 48)) & predsizem1;
 93      pred1 = fcm[hash];
 94
 95      stride = val - lastval;
 96      xor2 = val ^ (lastval + pred2);
 97      lastval = val;
 98      val = inbuf[i + 2];
 99      dfcm[dhash] = stride;
100      dhash = ((dhash << 2) ^ ((unsigned long long)stride >> 40)) & predsizem1;
101      pred2 = dfcm[dhash];
102
103      bcode = code | 0x8;
104      if ((unsigned long long)xor1 > (unsigned long long)xor2) {
105        code = bcode;
106        xor1 = xor2;
107      }
108      bcode = 7;              // 8 bytes
109      if (0 == (xor1 >> 56))
110        bcode = 6;            // 7 bytes
111      if (0 == (xor1 >> 48))
112        bcode = 5;            // 6 bytes
113      if (0 == (xor1 >> 40))
114        bcode = 4;            // 5 bytes
115      if (0 == (xor1 >> 24))
116        bcode = 3;            // 3 bytes
117      if (0 == (xor1 >> 16))
118        bcode = 2;            // 2 bytes
119      if (0 == (xor1 >> 8))
120        bcode = 1;            // 1 byte
121      if (0 == xor1)
122        bcode = 0;            // 0 bytes
123
124      *((long long *)&outbuf[(out >> 3) << 3]) |= xor1 << ((out & 0x7) << 3);
125      if (0 == (out & 0x7))
126        xor1 = 0;
127      *((long long *)&outbuf[((out >> 3) << 3) + 8]) = (unsigned long long)xor1 >>
         (64 - ((out & 0x7) << 3));
128
129      out += bcode + (bcode >> 2);
130      outbuf[6 + (i >> 1)] = code | bcode;
131    }
132    if (0 != (intot & 1)) {
```

```
133       out -= bcode + (bcode >> 2);
134     }
135     outbuf[0] = intot;
136     outbuf[1] = intot >> 8;
137     outbuf[2] = intot >> 16;
138     outbuf[3] = out;
139     outbuf[4] = out >> 8;
140     outbuf[5] = out >> 16;
141     ioc = fwrite(outbuf, 1, out, stdout);
142     assert(ioc == out);
143     intot = fread(inbuf, 8, SIZE, stdin);
144   }
145 }
146
147 static void Decompress()
148 {
149   register long i, in, intot, hash, dhash, code, bcode, predsizem1, end, tmp, ioc;
150   register long long val, lastval, stride, pred1, pred2, next;
151   register long long *fcm, *dfcm;
152   long long outbuf[SIZE];
153   unsigned char inbuf[(SIZE / 2) + (SIZE * 8) + 6 + 2];
154
155   assert(0 == ((long)inbuf & 0x7));
156
157   ioc = fread(inbuf, 1, 7, stdin);
158   if (1 != ioc) {
159     assert(7 == ioc);
160     predsizem1 = inbuf[0];
161     predsizem1 = (1L << predsizem1) - 1;
162
163     hash = 0;
164     dhash = 0;
165     lastval = 0;
166     pred1 = 0;
167     pred2 = 0;
168     fcm = (long long *)calloc(predsizem1 + 1, 8);
169     assert(NULL != fcm);
170     dfcm = (long long *)calloc(predsizem1 + 1, 8);
171     assert(NULL != dfcm);
172
173     intot = inbuf[3];
174     intot = (intot << 8) | inbuf[2];
175     intot = (intot << 8) | inbuf[1];
176     in = inbuf[6];
177     in = (in << 8) | inbuf[5];
178     in = (in << 8) | inbuf[4];
179     assert(SIZE >= intot);
180     do {
181       end = fread(inbuf, 1, in, stdin);
182       assert((end + 6) >= in);
183       in = (intot + 1) >> 1;
184       for (i = 0; i < intot; i += 2) {
185         code = inbuf[i >> 1];
186
187         val = *((long long *)&inbuf[(in >> 3) << 3]);
188         next = *((long long *)&inbuf[((in >> 3) << 3) + 8]);
189         tmp = (in & 0x7) << 3;
190         val = (unsigned long long)val >> tmp;
191         next <<= 64 - tmp;
192         if (0 == tmp)
193           next = 0;
194         val |= next;
195
```

```
196         bcode = (code >> 4) & 0x7;
197         val &= mask[bcode];
198         in += bcode + (bcode >> 2);
199 
200         if (0 != (code & 0x80))
201           pred1 = pred2;
202         val ^= pred1;
203 
204         fcm[hash] = val;
205         hash = ((hash << 6) ^ ((unsigned long long)val >> 48)) & predsizem1;
206         pred1 = fcm[hash];
207 
208         stride = val - lastval;
209         dfcm[dhash] = stride;
210         dhash = ((dhash << 2) ^ ((unsigned long long)stride >> 40)) & predsizem1;
211         pred2 = val + dfcm[dhash];
212         lastval = val;
213 
214         outbuf[i] = val;
215 
216         val = *((long long *)&inbuf[(in >> 3) << 3]);
217         next = *((long long *)&inbuf[((in >> 3) << 3) + 8]);
218         tmp = (in & 0x7) << 3;
219         val = (unsigned long long)val >> tmp;
220         next <<= 64 - tmp;
221         if (0 == tmp)
222           next = 0;
223         val |= next;
224 
225         bcode = code & 0x7;
226         val &= mask[bcode];
227         in += bcode + (bcode >> 2);
228 
229         if (0 != (code & 0x8))
230           pred1 = pred2;
231         val ^= pred1;
232 
233         fcm[hash] = val;
234         hash = ((hash << 6) ^ ((unsigned long long)val >> 48)) & predsizem1;
235         pred1 = fcm[hash];
236 
237         stride = val - lastval;
238         dfcm[dhash] = stride;
239         dhash = ((dhash << 2) ^ ((unsigned long long)stride >> 40)) & predsizem1;
240         pred2 = val + dfcm[dhash];
241         lastval = val;
242 
243         outbuf[i + 1] = val;
244       }
245       ioc = fwrite(outbuf, 8, intot, stdout);
246       assert(ioc == intot);
247       intot = 0;
248       if ((end - 6) >= in) {
249         intot = inbuf[in + 2];
250         intot = (intot << 8) | inbuf[in + 1];
251         intot = (intot << 8) | inbuf[in];
252         end = inbuf[in + 5];
253         end = (end << 8) | inbuf[in + 4];
254         end = (end << 8) | inbuf[in + 3];
255         in = end;
256       }
257       assert(SIZE >= intot);
258     } while (0 < intot);
```

```
259   }
260 }
261
262 int main(int argc, char *argv[])
263 {
264   long val, ioc;
265
266   assert(4 <= sizeof(long));
267   assert(8 == sizeof(long long));
268   assert(0 < SIZE);
269   assert(0 == (SIZE & 0xf));
270   val = 1;
271   assert(1 == *((char *)&val));
272
273   if (argc > 1) {
274     val = -1;
275     val = atol(argv[1]);
276     assert(0 <= val);
277     Compress(val);
278     ioc = fread(&val, 1, 1, stdin);
279     assert(0 == ioc);
280   } else {
281     Decompress();
282   }
283
284   return 0;
285 }
```