ISSN: 2993-8643

Research Article

Engineering: Open Access

Implementation of Floating-Point Arithmetic Coding Using x86-64 AVX-256 Assembly Language

Mike H.B. Gray*

Department of Chemistry, Occidental College, Los Angeles, CA 90041.

*Corresponding Author
Mike H.B. Gray, Department of Chemistry, Occidental College, Los Angeles, CA 90041.

Submitted: 2024, Jun 27; Accepted: 2024, Jul 05; Published: 2024, Jul 15

Citation: Mike H.B. Gray. (2024). Implementation of Floating-Point Arithmetic Coding Using x86-64 AVX-256 Assembly Language. *Eng OA, 2*(3), 01-18.

Abstract

Bit manipulations, especially those executed on multiple strings in parallel, e.g., on Intel® processors equipped with Advanced Vector Extensions (AVX), can be a powerful way to speed up unoptimized high-level sequentially executed code. A case in point is made for floating-point arithmetic coding (FPAC), implemented herein as a non-adaptive, lossless data compression algorithm using x86 AVX-256 stand-alone assembly language under 64-bit MASM assembler in Visual Studio 2022. Apart from writing and reading bit strings to and from file, FPAC can become fully vectorized and be improved in performance (relative to unoptimized integer versions) by orders of magnitude by blocking short sequences of symbols and bypassing interval renormalization. For an alphabet size, up to 53—the limiting case made for 0.474MB Protein Data Bank entry 4HHB, referred to as oxygen transport file (OTF)—it can also strongly outperform a commercially available, unoptimized C++ Huffman encoder by over a factor of 10 and beat the decoder by roughly a factor of 2. Disadvantageous but necessary to this prescription of vectorizations is an additional compressed storage requirement of the length of the codeword (this binary integer is to encode a block of 5 symbols) in addition to the codeword itself; for size-5 blocks, this compromises the compression efficiency as follows: the average number of bits per symbol required to compress the input message is demonstrated to lie in the interval [H(S) + b, H(S) + 0.4 + b], where b = 1.0 for single-precision floating-point arithmetic coding (SPFPAC), b = 1.2 for a slower but more practical double-precision counterpart (DPFPAC), and H(S) is the Shannon entropy of symbol frequencies.

Keywords: Advanced Vector Extensions (AVX), Assembly Language, Arithmetic Coding, Bit Manipulations, Data Compression

The Idea of Arithmetic Coding

We have an alphabet $S = \{s_1{:}s_m\}$ of m symbols that occur with repetition in a text file containing N total occurrences. The method is non-adaptive, and so the integer-valued symbol frequencies $f_1{:}f_m$ are required and obtained by a preliminary file scan. Consecutive sequences of symbols are represented by intervals of floating-point (FP) values between 0 and 1. Starting with the full range [0, 1), symbols are added to the sequence and the interval becomes narrower until we run out of FP precision required for the representation; in an attempt to prevent this from occurring, the sequence of collected symbols is truncated at a selected magic number, say 5, called the block size. The interval is reset to [0, 1) and the process starts over.

Having determined the interval between 0 and 1 corresponding to block $w = s_{i1}\dots s_{i5}$, the encoder chooses a single number t in the interval, called a tag, and represents w by a truncation of the binary representation of t. Just enough bits should be selected so that the decoder, when faced with this approximate representation of t, can recover the original sequence. Moreover, when w contains more frequent symbols, t (i.e., its truncation) should require fewer bits; similarly, when w contains less frequent symbols, t should require more bits. In this way, compression can be achieved by removing redundancies.

For optimal decoder operation, the frequencies are sorted in non-increasing order: $F_1 \geq \dots \geq F_m$ (these are relative frequencies: $F_k = f_k/N$). Define cumulative distribution function (CDF) as follows: $C_0 = 0$ and $C_{k+1} = C_k + F_{k+1}$ ($k = 0{:}\ m - 1$). When the next block is encountered and before its first symbol is encoded the entire interval is [0, 1), having length (i.e., range)

of L = 1. If [α, β) = [α, α+L) is the "current interval" after having encountered zero or more symbols and s_k is the next symbol encountered—index k in 1:m does not necessarily coincide with the iteration number in 1:5—then the next interval [α', β') is obtained by dividing [α, β) into a subinterval whose length is proportional to F_k as follows:

$$(*) \qquad \alpha' = \alpha + C_{k-1}L \qquad L' = LF_k$$

The formula for β is $\beta' = \beta + C_k L$ but we only store L as it involves less arithmetic. From the initialization [0, 1) and division formulas (*) we see that CDF values are nothing but the m + 1 possible interval endpoints for the first symbol encountered; and that the current length L is the iterated product of relative frequencies of symbols encountered so far, which indicates that blocks having more frequent symbols produce wider intervals. Inverse correlation of interval width to tag bit width will be seen when methods for binary tag generation are explored.

The process of subdivision is illustrated in Figure 1 with S = {a, b, c, d}, F_a = .4, F_b = .3, F_c = .2, and F_d = .1, whence C_a = .4, C_b = .7, C_c = .9, and C_d = 1.0. The sequence *badac* is encoded (only up to *bad* schematically) and taken from a larger anonymous message having given frequency distribution. If we stop at *bad*, the interval is [.508, .52); if we complete the 5-symbol block, the final interval becomes [.51136, .51232) having length L = .3 × .4 × .1 × .4 × .2 = .00096, the product of relative frequencies.

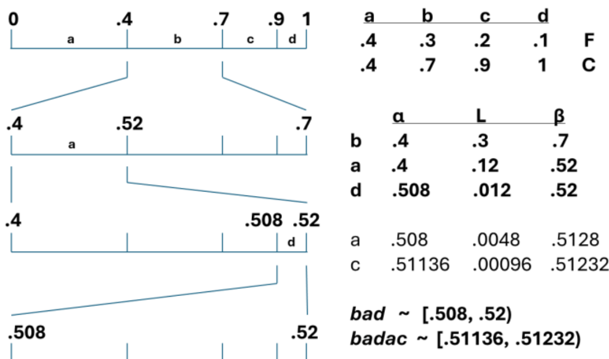


Figure 1: Interval Division

Each successive interval is a subinterval of the former, so if we provide decoder with the final interval [.51136, .51232), it can deduce the first symbol as *b*, since *b* is assigned to [.4, .7), which contains [.51136, .51232). The decoder can then divide [0, 1) into [.4, .7). The next symbol is decoded as *a*, since *ba* is assigned to [.4, .52), which is the only 2-symbol subinterval containing [.51136, .51232). Continuing like this, the decoder will correctly recover the sequence *badac*. But providing the decoder with two floating-point values, totaling 64 or 128 bits (for single or double precision) does not reduce the ASCII size of 40 bits per block, which explains why only a single number t (any value belonging to the final interval suffices) in binary format and hopefully fewer than 40 bits, is used for the compressed form of the interval.

Exploring Two Methods for Generating a Binary Tag

The first method explored is optimal from a standpoint of compression efficiency and involves calculating the *dyadic fraction with least denominator* (dfld)—see for a detailed treatment [1]. To summarize, the dfld t in (α, β]—note the interval is now open at left end—is the binary fraction of shortest bit length that belongs to (α, β]. To demonstrate its determination for (.51136, .51232] we first convert these endpoints (as decimal fractions) into binary fractions (they are continued fractions in this case), which gives:

$$.51136 = (0.10000010\ldots)_2$$
$$.51232 = (0.10000011\ldots)_2$$

These expansions are carried out until disagreement occurs; at this location we write bit 1 and truncate giving t = 0.10000011 = $2^{-1} + 128^{-1} + 256^{-1}$ = .51171875 = dfld in (.51136, .51232]. We can equally write t as a rational number, an odd number divided by a power of two, as t = $131/2^8$. Note that t requires 8 bits and no other number in said interval could have fewer. Moreover, as an interval widens, the location of disagreement in the endpoints heads toward the binary point, which establishes the inverse correlation of interval width to bit width and explains how compression occurs in context of FPAC. But the dfld tag method is prone to unwelcome precision errors. Consider this statistical model and block:

a	b	c	d	
.7	.1	.1	.1	F
.7	.8	.9	1	C

block ***dccbd***
α_{Ti83} = .98879
β_{Ti83} = .9888
$t_{Ti83} = 129603/2^{17}$
= .9887924194 . . .

Encoding this block using Texas Instruments™ calculator Ti83 (accurate to 14 digits of precision, with a home screen display of 10) gives the indicated endpoints and tag (note 129603 = 1FA43h requires 17 bits). If we perform this calculation using single-precision floating point (SPFP), known to give just under 7 reliable digits of precision, we might observe a C++ program (this test was done) producing β = .98880005. The dfld in (.98879, .98880005] is shown, using Ti83, to be t = $32401/2^{15}$ = .9888000488 . . ., which requires only 15 bits, but does not lie in (.98879, .9888], the correct interval in exact arithmetic. The decoder is now at risk of incorrectly decoding the block, even though α, β, and L are all within the SPFP limit of precision. The problem is that there could be another dyadic fraction of shorter bit width in an adjacent interval, as we have shown, dangerously close to an interval endpoint.

Experimenting with SPFPAC using dfld and observing errors of this nature led to its abandonment as a viable computational strategy. If, instead, we choose the interval midpoint and

truncate it to a certain prescribed number of bits, errors will come about predictably via the interval length dropping below a decided upon precision tolerance; not only will fewer result, but they can be detected and fixed.

The dfld and midpoint methods differ in how t is obtained and in how many bits of its binary representation are required so the decoder can uniquely recover its containing subintervals. As has been pointed out, the intervals assigned to individual symbols using CDF values (C_k)—see integer division formulas (*)—form a partition of [0, 1). In fact, the entire set { [α, β) } of all m^n possible intervals for blocks of fixed size n forms a partition of [0, 1) as well. Then any number in [α, β) serves as a unique identifier (i.e., tag) of [α, β). Sayood [2] shows by using the midpoint t = (α + β) / 2 and truncating its binary representation to $l(t) = 1 + \text{ceiling}(\log2(L^{-1}))$ many bits, that this truncated binary number is guaranteed to remain in its interval, as required for correct decoder operation. He further shows that the m^n tags generated in this manner form a prefix code (none of these binary words can be a prefix of the other). So FPAC implemented with blocking is, effectively, a compression by replacement scheme, although it is not necessary (as is the case for Huffman encoding [3]) to maintain a codebook of these prefix codes.

Efficiency of Arithmetic Encoding

Throughout this section (and the whole article for that matter), we have the well-rehearsed premise that $S = \{s_1 : s_m\}$ is an alphabet whose symbols have relative frequencies $F_1{:}F_m$ in a text message [4]. The entropy of this message, or sometimes stated as the entropy H(S) of these frequencies is defined as

$$H(S) = \sum_{k=1}^{m} F_k \log2\left(\frac{1}{F_k}\right)$$

It is well known that H(S) is a lower bound for the average number of bits per symbol (we write this average as <l(s)>) achievable in data compression by replacement schemes (e.g., Huffman and prefix codes) that involve binary codes. Sayood proves this lower bound for the midpoint method; the salient argument is that, in generating a prefix code, the average length <l(t)> of all m^n truncated tags cannot drop below n × H(S). Hankerson et al. [1] do not attempt a proof of the lower bound for the dfld method, citing theoretical formalities. Upper bounds for midpoint and dfld methods are shown in [1] and [2] to be vanishingly higher than H(S) as the block size becomes very large. This theorem clarifies:

Theorem-*The average number of bits per symbol <l(s)> obtained by arithmetic coding using dfld satisfies*

$$\langle l(s)\rangle \le H(S) + 1/n$$

whereas the less-efficient midpoint method satisfies

$$H(S) \le \langle l(s)\rangle \le H(S) + 2/n$$

Owing to an additional storage requirement discussed shortly, vectorized FPAC (of this article) adds either 1.0 or 1.2 to both the lower and upper bounds of the theorem.

Description of FPAC Encoder Algorithm

As discussed, the method is not adaptive, and a preliminary file scan and sorting of frequency array are required. Symbols associated with F and C are numbered 1:m, where the most frequent symbol is given index 1, the least frequent symbol index m. The input message (file) is represented by a character array called `ibuff`. The encoder operates in three phases: division, binary tag generation, and storage to compressed buffer. Pseudocode for processing a single block of size n (we have used n = 5 throughout) follows. Extensions to multiple blocks processed in parallel via x86 AVX are described later. Error detection is ignored temporarily.

Phase 1. Interval division and midpoint determination:

- Initialize α = 0; L = 1.0;
- Repeat n times: (divisions loop)

  Read the next symbol with index k from `ibuff` and perform the left-endpoint and interval length updates:

  $(\alpha = \alpha + C_{k-1}L)$ $(L = LF_k)$

  End Repeat
- β = α + L
- t = (α + β) / 2.0 (calculate midpoint t as FP)

Phase 2. Binary tag generation (this involves three steps):

- Convert FP representation of t into a left-justified bit string (E.G., $5/32 = (.00101)_2$ becomes 001010 . . . 0)
- Compute $l(t) = 2 + \text{floor}(\log2(L^{-1}))$
- Truncate bit string to l(t) many bits and right justify. This string, perhaps thought of as a binary integer, is called the *code*—E.G., the code for $(.00101)_2$ is 00101 if l(t) = 5, or just 001 if l(t) = 3.

Phase 3. Write the binary code and its length to compressed buffer.

The length l(t) is stored so the decoder can read the entire code segment and compute the tag FP representation (as t = code divided by $2^{l(t)}$) before the divisions loop begins. An alternative would be to implement incremental reception where the decoder processes one bit at a time—this is the hallmark of integer arithmetic coding—see WNC [5]. But this strategy, while having a very high compression efficiency, involves an unpredictable number of interval rescalings and so our hopes of processing multiple tags concurrently appear to be lost.

For *simulated file I/O,* the compressed buffer will consist of two arrays: `lengths` and `codes`. The former stores unsigned 8-bit integers, the latter 32-bit unsigned integers (for SPFP) or 64-bit for DPFP. In a later section concerning *contiguous file I/O*, the compressed buffer is modeled as a single array of 64-bit integers, the packed representation of `lengths` and `codes`. The code lengths will vary up to 31 bits for SPFPAC and up to 42 for DPFPAC (40 and up signals an error warning for DPFPAC; no error detection was implemented for SPFPAC). Storing these lengths requires 5 bits for SPFPAC and 6 for DPFPAC. Adding 5 bits to the code storage requirement changes the compression efficiency to give <l(s)> ≤ H(S) + 0.4 + 1 (where 1 =

5/5 adds one extra bit per symbol in a block of size 5) or in case we add 6 extra bits to a block of 5, then $<l(s)> \leq H(S) + 0.4 + 1.2$ (where 1.2 = 6/5 extra bits).

Description of FPAC Decoder

For the most part, the decoder mimics the encoder. The restored file is represented by a character array called `obuff`.

Phase 1. Load the code and its bit length l(t) from compressed buffer. Now compute $t = code/2^{l(t)}$ as FP value.

Phase 2. Decode block of n symbols:

- Initialize $\alpha = 0$; $L = 1.0$;
- Repeat n times: (divisions loop)

  Calculate scaled tag $t^* = (t - \alpha)/L$

  (Decode symbol procedure)

  Find index k in 1:m such that $C_{k-1} \leq t^* < C_k$ (this is equivalent to finding in which possible subinterval the unmodified tag t lies; the scaled tag method involves less arithmetic)

  Perform interval updates: $(\alpha = \alpha + C_{k-1}L)$ $(L = LF_k)$

  Write symbol s_k to `obuff`

  End Repeat

The *decode symbol* procedure in the divisions loop is the bottleneck of decoder and costs us m comparisons in the worst case, assuming it is executed sequentially. It is clear why F has been sorted in non-increasing order, since more frequent symbols have lower-valued indices and involve fewer comparisons. A binary interval search will help for larger alphabets and even more so when we perform the search to decode multiple symbols concurrently; this will lower the average number of comparisons needed to process each one. Another method involves performing a sequential search on multiple symbols concurrently (referred to as *vector sequential search* herein). These two methods are described in later sections and give the expected improvements over the one-by-one (i.e., one symbol at a time) sequential search.

Implementation Details

All programs were developed in Microsoft® Visual C++ and Macro Assembler (MASM), which are included with Visual Studio. Two separate files are required for the C++ (.cpp) and assembly language (.asm) code. All precompiled header files are built into C++. A C++ function `main`—see Supporting Information (SI)—contains code that calls assembly language functions for performing encoding and decoding. For instance, the function header for SPFPAC encoder, declared in C++ (.cpp) follows:

```
extern "C" void encode_spfp_avx256(uint8_t* ibuff,
int n_octets, float* F, float* C, uint8_t* lgths,
uint32_t* codes);
```

Before we schematically provide its stack frame (see Figure 2) the pertinent features of Visual C++ calling convention are underscored:

(Visual C++ Calling Convention)

- Parameters are always 8-byte values, even though only a fraction of this space may be needed.
- The first four parameters are passed in registers rather than on stack. Integer arguments are passed in RCX, RDX, R8, and R9; *shadow space* storage is automatically reserved for these four parameters on stack even though it might not be used. Floating-point arguments are passed in LO bits of XMM0, XMM1, XMM2, and XMM3.
- 8 bytes are required for each additional parameter if there are five or more parameters.

The calling convention has more intricacies (e.g., compliance with Microsoft ABI, alignment requirements of stack pointer RSP, and designation of registers as either volatile or nonvolatile); most of these are not relevant here as the application is stand-alone assembly language, having a limited external interface. One can consult Kusswurm [6] or Hyde [7] for details.

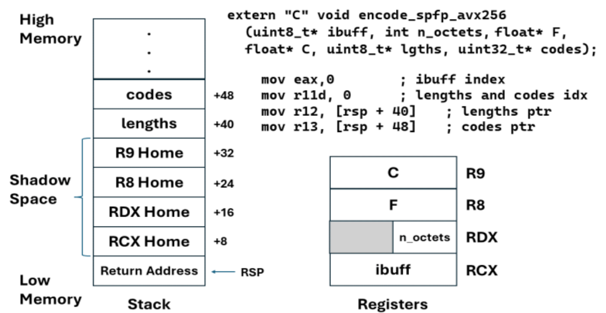


Figure 2: Stack Frame for encode_spfp_avx256

All parameters except `n_octets` are 8-byte pointers and take up the full 64-bit space. The `n_octets` parameter is a 32-bit integer, so it only needs 4 bytes (the other 4 are unused). Note `lengths` and `codes` are the fifth and sixth parameters and are located at byte offsets of +40 and +48 relative to RSP. These pointers were loaded into registers R12 and R13 for faster access, as indicated by the two `mov` instructions. The first four parameters are automatically loaded into registers RCX, RDX, R8, and R9. R11 is used to index both `lengths` and `codes`.

RCX is used as a fixed pointer to (the first byte of) `ibuff`; RAX is used to index the first block in 8 total (incremented by 1 byte per iteration of divisions loop; then by 35 bytes to the next octet, to complete the cycle). Offsets of 0, 5, 10, 15, 20, 25, 30, and 35 bytes relative to RCX+RAX are used to parallel load an octet of symbols—see Figure 3. To carry out this technique, said offsets with 5-byte spacing are loaded from 256-bit memory operand `ibuff_offsets` into YMM8 via instruction `vmovdqa`. This is an aligned move, which requires the memory operand to be 32-byte aligned. An unaligned move would work as well, as it could be done outside divisions loop and the performance penalty would be unnoticeable. Alignment is achieved by placing the declaration of memory operand in a 32-byte aligned memory segment

(not shown here); the simple `align` directive does not work for 32 bytes.

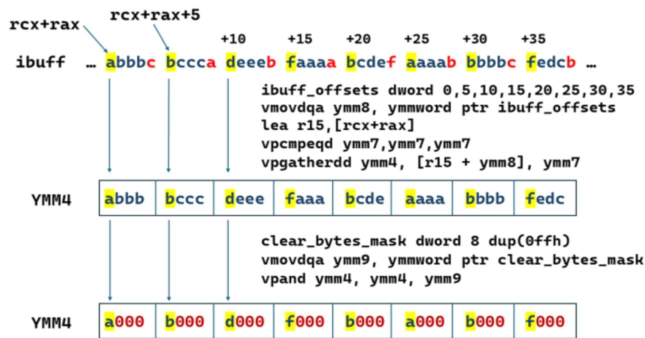


Figure 3: Parallel Loading an Octet of Symbols (SPFPAC)

The address of the first byte to be loaded (pointed to by RCX+RAX) is placed in R15 via `lea` (load effective address). The crux of parallel data transfer is instruction `vpgatherdd ymm4, [r15 + ymm8], ymm7`, which loads dword data into YMM4 using dword indices in YMM8; these indices act as offsets relative to R15 to access source operand `[r15 + ymm8]`. Register YMM 7 stores a gather mask; the HO bit of 8 dword lanes indicates whether data should be moved for that lane. In this case, the mask seems to be superfluous as all lanes are involved; instruction `vpcmpeqd ymm7,ymm7,ymm7` compares YMM7 to itself, which sets all 256 bits to 1. The reason for loading dword lanes of YMM4 with bytes from `ibuff` is that these bytes will act as indices themselves for accessing F and C arrays via further gather instructions, and these limit us to dword or qword indices (rather than our mere single byte requirement). Moreover, this instruction moves 8 groups of 4 bytes, so we need to clear out 3 unused bytes per group by AND'ing YMM4 with an appropriate mask loaded into YMM9.

The strategy for DPFPAC (which requires only 4 dword indices) is simpler; it loads the 4 dword lanes of an XMM register, one by one, using `vpinsrb`, which can act as an index source register for subsequent gather instructions. Although one-by-one insertion is not done, by definition, in parallel, it is arguably better than performing in only 4 lanes the analogous SPFPAC route, which requires clearing out wasted bytes and loading two separate masks. Both methods were employed for DPFPAC, but no performance difference was noticeable owing to the majority of work done in the divisions loop.

In order to gather values of C_{k-1} and F_k to perform updates for α and L, the character (byte) data must be converted to alphabet indices 1:m. For SPFPAC, the decision was to simply test lower case letters a, b, ... , l (only up to m=12), which correspond to ASCII hex codes 61h, 62h, ... , 6Ch. We subtract 61h to obtain CDF index k – 1, which is done by loading YMM6 with 8 copies of this constant and performing packed integer subtraction via `vpsubd ymm4, ymm4, ymm6`. There is no packed integer increment-by-one instruction to obtain k from k – 1, but 1 can be broadcast to 8 dword lanes and packed addition performed. A more elaborate conversion scheme is required for DPFPAC so a more diverse set of characters can be tested—this is described in a later section when we focus on DPFPAC.

With L stored in YMM1 and F_k gathered in YMM3, L is updated using packed SPFP multiplication `vmulps ymm1,ymm1,ymm3` where YMM1 acts as both source and destination operand. The update $\alpha' = \alpha + C_{k-1}L$ involves multiplication followed by addition. *Fused-multiply-add* (FMA) instruction in AVX-256 combines these operations into one. It performs packed FP multiplication followed by addition and uses a single rounding operation (the intermediate product is not rounded) making it more accurate than performing two steps separately. The rounding mode is determined by MXCSR control register, which has been set to *truncate* in the code listings. With α in YMM0, L in YMM1, and C_{k-1} in YMM2, the instruction `vfmadd231ps ymm0,ymm1,ymm2` performs the required update. The first two indices (23) in mnemonic indicate multiplicand and multiplier; the third index (1) indicates the source operand added to the product and is the overall destination, as well. Note that 231 corresponds to YMM1, YMM2, and YMM0, respectively.

Phase 2 (Convert FP to Left-Justified Bit String)

Before describing an implementation, a brief hiatus is taken to review IEEE 754 standard regarding FP numbers:

- Three fields are required: sign bit, exponent, and significand. The ordered representations for 32- and 64-bit types follow, where for our purpose the leading sign bit is usually 0 to indicate positive:

SPFP: (0) (8-bit biased exponent) (23-bit significand)

DPFP: (0) (11-bit biased exponent) (52-bit significand)

- To convert a binary fraction, say 0.1101 = 13/16 to SPFP, write it in normalized binary scientific as 1.101e-1, having a 3-bit significand of 101 (the leading 1 is omitted). Now add the bias value of 127 to the exponent (-1) to obtain the 8-bit biased exponent as 126 = $(01111110)_2$ = 7Eh. Then:

  $(0.1101)_2 = (0)(01111110)(101\ 0^{20})$ = 3F500000h as SPFP

  Enough 0's were added to significand to make up the entire 23-bit space. Converting to DPFP is the same process, but the bias is 1023 = $(01111111111)_2$ = 3FFh. Then $(0.1101)_2$ = 3FEA 0^{12} h as DPFP.

To recover the binary representation of a FP fractional value, we extract the significand and insert the leading 1 along with as many 0's that occur before it and after the binary point. First, we describe how this can be done for just one SPFP value, understood to be a fraction between 0 and 1, and stored in a 32-bit general-purpose register, say EAX. For left justification, it is desired to replace its HO bits with the binary representation of given fraction. E.G., 0.15625 = $(0.00101)_2$, so we start with EAX = 3E200000h (this is 0.15625 as SPFP). Upon completion, the HO bits will be 00101, that is, EAX = 28000000h.

The procedure follows:

- AND EAX with FF800000h and put result in EBX (EAX remains unchanged). Now shift right logical EBX by 23 bits so that the biased exponent is right-justified in EBX. Subtract EBX from 126 to give the number of leading zeros to appear in front of significand.

- Shift left EAX by 8 bits so the LO bit of exponent now occupies the HO position in EAX and is followed by significand. Insert the missing bit 1 at this HO location; this can be done by OR'ing EAX with mask 80000000h.

- Shift right logical EAX by the number of bits determined in the first step. Now the significand with leading 0's and the missing lead bit 1 is left-justified in EAX.

(Proof of correctness) Start with binary value $v = (0.1x_1 \dots x_{23})_2$, each $x_i = 0$ or 1, and note that v cannot represent a continued fraction or a dyadic fraction requiring more than 24 bits without rounding and losing accuracy. Then $v = (1.x_1 \dots x_{23}\ e\text{-}1)_2$, the biased exponent is 126, and so EAX = $0\ 0111\ 1110\ x_1 \dots x_{23}$, initially; applying step 1 indicates there are no leading 0's to be inserted. Completing the procedure gives EAX = $1x_1 \dots x_{23}\ 0^8$, as required. A proof by induction follows if we next consider $v = (0.01x_2 \dots x_{23})_2$, etc.

All these steps can be vectorized using AVX-256 instructions; we demonstrate for DPFP in Code Listing 1. Instead of subtracting from 126, we subtract from 1022; we shift right logical by 52 bits instead of 23; we shift left by 11 instead of 8; the exponent mask from step 1 is adjusted to $FFF0^{13}$h. The crucial instruction is `vpsrlvq ymm0, ymm0, ymm2` (variable shift right quadword) where qword lanes in YMM0 are independently shifted right by variable counts specified in the qword lanes of YMM2. In context, YMM2 stores the variable number of leading 0's to be inserted into HO bits of qwords in YMM0. YMM0 is both a source and destination of these shifts.

Code Listing 1. DPFP Conversion to Left-Justified Bit String

```
.data
seg1 segment readonly align(32)
y8 qword 4 dup(0fff0000000000000h)      ; exp mask
y9 qword 4 dup(1022)                    ; exp sub bias
y10 qword 4 dup(8000000000000000h)      ; insert lead bit
vals real8 0.5, 0.1, 0.125, 0.8125      ; vals to convert
seg1 ends

.code
asm_ PROC

vmovapd ymm0, ymmword ptr vals
vmovdqa ymm8, ymmword ptr y8
vmovdqa ymm9, ymmword ptr y9
vmovdqa ymm10, ymmword ptr y10

vpand ymm2, ymm0, ymm8     ; isolate exp at HO end
vpsrlq ymm2, ymm2, 52      ; shift exp into LO end
vpsubq ymm2, ymm9, ymm2    ; sub 1022 gives #leading 0's
vpsllq ymm0, ymm0, 11      ; shift out exp from ymm0
vpor ymm0, ymm0, ymm10     ; insert lead bit 1
vpsrlvq ymm0, ymm0, ymm2   ; insert leading 0's

; YMM0 = D000000000000000-2000000000000000-
;        1999999999999A00-8000000000000000

ret
asm_ ENDP
```

Log2 Calculation SPFP (x87 versus AVX-256)

Using x87 FPU to calculate $\log2(L^{-1})$ disrupts parallel operation and is a bottleneck to encoding. Even if it is abandoned, it is useful to compare it to a parallel binary interval search method, where the results are implementation and CPU dependent (time testing results are delayed until CPU identification is provided). See SI for the C++ `main` driving program for log2 calculations. The complete x86-x87 listing is found in Code Listing 2.

Code Listing 2. Log2 Calculation (x87 FPU)

```
.data
seg1 segment align(32)
L real4 8 dup(?)                    ; mem for log2 calc
log2_res dword 8 dup(?)             ; store fpu data
seg1 ends

.code
; extern "C" void log2_sequential_octets(int n_octets,
;     float* inputs, int32_t * outputs);

calc_log2 MACRO idx
fld1                               ; st0 = 1
fld1                               ; st1 = st0 = 1
fld [L + 4*idx]                    ; st0 = L st1 = st2 = 1
fdiv                               ; st0 = 1/L st1 = 1
fyl2x                              ; st0 = 1*log2(1/L)
fisttp [log2_res + 4*idx]          ; store floor(log2)
ENDM

log2_sequential_octets PROC

mov eax,0                          ; init i/o index
next_octet:                        ; begin loop

vmovdqa ymm1, ymmword ptr [rdx+rax*4]
vmovdqa ymmword ptr L, ymm1

calc_log2 0          ; sto floor(log2(1/L)) in log_res
calc_log2 1          ; sto in log_res + 4
calc_log2 2          ; sto in log_res + 8
calc_log2 3
calc_log2 4
calc_log2 5
calc_log2 6
calc_log2 7          ; sto in log_res + 28

vmovdqa ymm1, ymmword ptr [log2_res]
vmovdqa ymmword ptr [r8+rax*4], ymm1

add eax,8               ; incr i/o index
dec ecx                 ; decr loop counter
test ecx,ecx            ; test for zero
jnz next_octet          ; repeat if nonzero
ret
log2_sequential_octets ENDP
```

The `main` driving program creates a suitably large array `inputs` containing floating-point L-values between 2^{-8} and 2^{-18}, typical for the demand of accurate SPFPAC calculations; array `outputs` stores calculated values of $\text{floor}(\log2(L^{-1}))$. These two arrays are pointed to by RDX and R8 according to the header for `log2_sequential_octets`, which is called by C++. The first parameter `n_octets` counts the number of input values in groups of 8. The FPU will repeat the macro `calc_log2` 8 times sequentially without performing an expensive jump instruction in order to mimic AVX log2 calculation strategy. The crux of x87 log2 calculation is the usage of instruction `fyl2x`, which multiplies `st1`, the second element on stack, times log2 of the

top of stack `st0`. To simulate how SPFPAC operates, an octet of SPFP values is loaded into YMM1 via `vmovdqa ymm1, ymmword ptr [rdx+rax*4]`. The x87 FPU cannot directly interact with AVX registers, but we can store YMM1 in memory operand L before loading onto FPU stack. The instruction `fisttp` (integer store truncate pop) truncates `st0` to an integer by removing the fractional part, stores result in memory operand `log2_res` (which points to an octet of dwords), and pops the result off the stack. This 256-bit block of memory is eventually uploaded back to YMM1, and then stored in `outputs`.

In Code Listing 2, the x87 FPU calculates log2 using 80-bit extended double precision, and then all this effort is wasted by truncating it to integer. A faster strategy appears to involve calculating L^{-1} as a floating-point value, truncating it to integer, and then finding (via binary search) the interval $[2^k, 2^{k+1})$ containing this integer. Then k = floor(log2(floor(L^{-1}))). Moreover, floor(log(•)) = floor(log(floor(•))), regardless of whether the argument is truncated. The search requires at most floor (log2(p – 1)) integer comparisons, where p is the number of entries of a random-access power-two lookup table (p – 1 is the number of intervals formed by these powers). The truncated value of L^{-1} could coincide with an interval endpoint, and for a conventional search, the found index could be returned early. But a facile AVX strategy involves running each search, one performed independently per lane, the same number of iterations. The C++ function in Code Listing 3 does just this for a single search (Listing 4 provides an AVX extension).

Code Listing 3. Binary Interval Search (C++)

```
int floor_log2
(uint32_t pow2[], int log2p, int Low, int High, float x)
{
    uint32_t T = (uint32_t) 1 / x;
    for (int i = 0; i < log2p; ++i) {
        int Mid = (Low + High) / 2;
        if (T < pow2[Mid]) High = Mid - 1;
        else Low = Mid + 1;
    }
    if (T < pow2[Low]) Low = Low - 1;
    return Low;
}
```

For this search, we want to find interval $[2^k, 2^{k+1})$ containing truncated integer $T = x^{-1}$. After executing floor(log2(p – 1)) times, T is in either $[2^{Low}, 2^{Low+1})$ or $[2^{Low-1}, 2^{Low})$; in the former case, floor(log2(T)) = Low; in the latter, floor(log2(T)) = Low – 1. If the index could have been discovered in fewer than floor(log2(p – 1)) iterations (signaling an early return for a conventional search), the loop just executes a few more times with Low and High no longer changing.

Figure 4 demonstrates a proposed method to update four independent values of H (High) using vectorized Boolean operations (this would be a schematic for DPFPAC, where all values occupy qword lanes of YMM registers). The identifier P_x (notation borrowed from [2]) is an alias for the FP interval length L to prevent a temporary name clash with Low (L is written for Low in some of the comments; likewise, M is written for Mid on occasion). The values of pow2[Mid] are not shown but are understood. YMM14 stores the results of vector comparison ($1/P_x$ < pow2[Mid]); when true, a lane stores all 1's (as F^{16} in hex, but only FFFF is written in lacking space); otherwise, a lane stores all 0's. A value of *true* (all 1's) is -1 as a signed integer, so we can add this to Mid to obtain M – 1 as the update of H (but only when said comparison is true). In the other lanes we want the old value of H. So, the two registers YMM14 and YMM15 = NOT YMM14 can be used as masks to selectively pick the values of M that need to be decremented (via YMM7 = YMM4 AND YMM14) and the values of H that remain unchanged (via YMM13 = YMM3 AND YMM15). These values are then glued back together via the final OR operation to give the updated H.

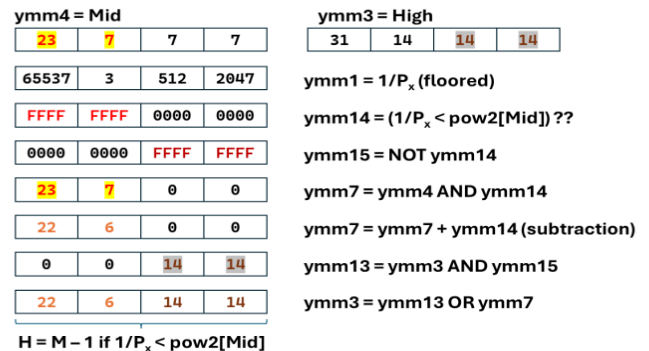


Figure 4: Updating H (AVX Binary Interval Search)

The update of L = M + 1 when comparison (1/Px < pow2[Mid]) is *false* is similar, but we need to subtract -1 from M to achieve addition. The complete x86 AVX binary interval search method to calculate floor(log2) for SPFP follows in Code Listing 4.

Code Listing 4. Log2 Calculation (AVX Binary Search)

```
.data
H_init dword 31                 ; initialize H to 31
log2_31 dword 4                 ; #reps of avxBS

seg2 segment align(32)
one_p0 real4 8 dup(1.0)
pow2 dword 1h,2h,4h,8h,10h,20h,40h,80h,
    100h,200h,400h,800h,
    1000h,2000h,4000h,8000h,
    10000h,20000h,40000h,80000h,
    100000h,200000h,400000h,800000h,
    1000000h,2000000h,4000000h,8000000h,
    10000000h,20000000h,40000000h,80000000h
seg2 ends

.code

; extern "C" void log2_avx_octets(int n_octets,
;    float* inputs, uint32_t * outputs);

log2_avx_octets PROC

mov eax,0
next_octet:
vmovdqa ymm1, ymmword ptr [rdx+rax*4]
vmovdqa ymm7, ymmword ptr [one_p0]

vdivps ymm1,ymm7,ymm1           ; y1 = 1/Px (SPFP)
vcvttps2dq ymm1,ymm1            ; take floor (truncate)

; y2 = Low  y3 = High  y4 = Mid  ebx = #reps of avxBS

vpxor ymm2,ymm2,ymm2                      ; Low = 0
```

```
vpbroadcastd ymm3, dword ptr [H_init]     ; High = 31
mov ebx, dword ptr [log2_31]

log2_avxBS:                              ; begin loop
vpaddd ymm4,ymm2,ymm3                    ; y4 = M
vpsrld ymm4,ymm4,1                       ; = (L+H)/2
vpcmpeqd ymm7,ymm7,ymm7                  ; set gather mask
vpgatherdd ymm13, [pow2+ymm4*4], ymm7 ; y13 = pow2[M]

vpcmpgtd ymm14,ymm13,ymm1          ; y14 = 1/Px < pow2[M]
vpcmpeqd ymm15,ymm15,ymm15         ; y15 = all 1's
vpxor ymm15,ymm14,ymm15            ; y15 = not (y14)

vpand ymm7,ymm14,ymm4
vpaddd ymm7,ymm7,ymm14
vpand ymm13,ymm3,ymm15
vpor ymm3,ymm13,ymm7       ; H = M-1 if 1/Px < pow2[M]

vpand ymm7,ymm15,ymm4
vpsubd ymm7,ymm7,ymm15
vpand ymm13,ymm2,ymm14
vpor ymm2,ymm13,ymm7       ; L = M+1 otherwise

dec ebx
test ebx,ebx
jnz log2_avxBS             ; rep floor(log2(31)) times

vpcmpeqd ymm7,ymm7,ymm7                  ; set mask
vpgatherdd ymm13, [pow2+ymm2*4], ymm7   ; y13 = pow2[L]
vpcmpgtd ymm14,ymm13,ymm1     ; y14 = 1/Px < pow2[L]
vpaddd ymm1,ymm2,ymm14        ; dec L if 1/Px < pow2[L]

 ; end log2_avxBS, return floor(log2(1/Px)) in y1

vmovdqa ymmword ptr [r8+rax*4], ymm1
add eax,8
dec ecx
test ecx,ecx
jnz next_octet
ret
log2_avx_octets ENDP
```

Conversions between Int64 and DPFP

In observing Code Listing 4, the instruction `vcvttps2dq ymm1, ymm1` (convert packed single precision to signed integer via truncation) to take floor($1/P_x$), does not have a double-precision analog in AVX-256 (nor is there one to convert int64 to DPFP), so bespoke methods are needed. The conversion int64 to DPFP is needed by decoder to calculate the tag before divisions loop; DPFP to int64 is needed by encoder to begin its log2 calculation. These conversions are now described.

(Int64 to DPFP) If x is a 64-bit integer requiring under 52 bits, then $x + 2^{52}$ is in $[2^{52}, 2^{53})$, and so $dp(x + 2^{52})$, its DPFP representation, has x as its LO bits (i.e., the significand is nothing but x, right-justified perfectly into place). So given that $dp(2^{52}) = 4330\ 0^{12}h$, we OR $(x)_2$ with this magic value to obtain $dp(x + 2^{52})$. For example, if $x = 5 = (101)_2$, then $dp(x + 2^{52}) = 4330\ 0^{11}\ 5h$. As an identity:

$$dp(x + 2^{52}) = (x)_2 \text{ OR } dp(2^{52}) \text{ (if } x < 2^{52})$$

Then $dp(x) = \{(x)_2 \text{ OR } dp(2^{52})\} - dp(2^{52})$, where the indicated subtraction is done as DPFP, not as integer subtraction.

(DPFP to Int64) Given DPFP value $x < 2^{52}$, the result y of DPFP addition of 2^{52} with x will leave the integer part of x as the significand with fractional bits removed; this follows since $x + 2^{52}$ is in $[2^{52}, 2^{53})$ and the integer part requires exactly 52 bits of the significand for its representation; there are no more bits for the fractional part. Next, we integer-subtract $dp(2^{52})$ from y, giving int64(x), as required. The subtraction is more efficiently performed using XOR, which clears out 433h, the HO 12 bits. As an identity:

$$\text{int64}(x) = \{x + dp(2^{52})\} \text{ XOR } dp(2^{52}) \text{ (if } x < 2^{52})$$

The overall conversion requires only two instructions (`vaddpd` and `vpxor`). The addition step that removes the fractional part of x involves rounding, whose mode is determined by 32-bit MXCSR register. In Code Listing 5, the rounding mode has been set to truncate (bits 13 and 14 of MXCSR are both set). The DPFP constant $4330\ 0^{12}h$ for 2^{52} has been imported from C++, along with an array of additional powers of two, just to facilitate the demonstration.

Code Listing 5. Conversions between Int64 and DPFP

```
#include <math.h>
extern "C" double two52 = pow(2, 52.0);
extern "C" double ary2[] =
    { pow(2, 42), pow(2,51), pow(2,52), pow(2,53)};

.data
extern two52:real8
extern ary2:real8
mxcsr_state dword ?

seg segment readonly align(32)
i64s qword 101, 5, 0, 12345678999
ary1 real8 42.875, 51.875, 52.875, 53.875
seg ends

.code
asm_ PROC

vstmxcsr mxcsr_state    ; store current state
bts mxcsr_state, 13     ; set bit 13 to 1
bts mxcsr_state, 14     ; set bit 14 to 1
vldmxcsr mxcsr_state    ; load new state

vmovdqa ymm0, ymmword ptr [i64s]
vbroadcastsd ymm1, real8 ptr [two52]

vpor ymm2,ymm0,ymm1
vsubpd ymm3,ymm2,ymm1
; y3 = 101.0 5.0 0.0 12345678999.0

; YMM2 = 43300002DFDC1C97-4330000000000000-
;       4330000000000005-4330000000000065

; YMM3 = 4206FEE0E4B80000-0000000000000000-
;       4014000000000000-4059400000000000

vmovapd ymm4, real8 ptr ary1
vmovupd ymm5, real8 ptr ary2
; YMM4 = 404AF00000000000-404A700000000000-
;       4049F00000000000-4045700000000000
; YMM5 = 4340000000000000-4330000000000000-
;       4320000000000000-4290000000000000
vaddpd ymm5,ymm4,ymm5      ; DPFP vals to convert

; YMM5 = 434000000000001A-4330000000000034-
;       4320000000000067-429000000000AB80
```

```
vaddpd ymm5,ymm1,ymm5

; YMM5 = 434800000000001A-434000000000001A-
;        4338000000000033-433004000000002A

vpsubd ymm6,ymm5,ymm1

; YMM6 = 001800000000001A-001000000000001A-
;        0008000000000033-000004000000002A

vpxor ymm7,ymm5,ymm1

; YMM7 = 007800000000001A-007000000000001A-
;        0008000000000033-000004000000002A

ret
asm_ ENDP
```

The truncated result of conversion of 2^{51} + 51.875 to int64 is indicated as 80^{10}33h in the second lane (from the right end) of YMM6, which is obtained via DPFP subtraction; YMM7 gives the same result via the more efficient XOR. Lanes 3 and 4 give wrong answers by attempting to convert 52.875 + 2^{52} and 53.875 + 2^{53}, which are both > 2^{52}. On the other hand, all i64's are < 2^{52} and are correctly converted to DPFP, as seen in YMM3.

Encoder Phase 3 (SPFPAC) Simulated File I/O

This phase involves writing the binary encoded tag to compressed buffer. For SPFPAC, error detection has been ignored so we unconditionally write l(t) to lengths and l(t) many bits of the binary code to codes. For simulated bit I/O, there is one 5-bit length stored per 8-bit integer so 3 bits are lost; and only one code value is stored per 32-bit integer (several could be lost). For gauging compression efficiency, it is assumed these holes have been removed, as will be done later when contiguous file I/O is considered. Currently, truncated log2 values are stored in YMM1 so we add 2 to obtain the necessary lengths, as l(t) = 2 + floor(log2(L^{-1}))—this step was stated in phase 2 but is provided here for continuation. With YMM0 storing 8 left-justified codes, we need to shift l(t) many HO bits to the low ends and use vmovdqa ymmword ptr [r13 + r11*4], ymm0 to transfer 8 dwords to codes. This code snippet demonstrates:

```
vmovdqa ymm13, ymmword ptr twos   ; y13 = 2 (8 copies)
vmovdqa ymm11, ymmword ptr _32    ; y11 = 20h (8 copies)
vpaddd ymm1, ymm1, ymm13    ; y1 = 2 + floor(log2(1/Px))
vpsubd ymm2, ymm11, ymm1    ; #bits to srl = 32 – l(t)
vpsrlvd ymm0, ymm0, ymm2    ; right justify codes
vmovdqa ymmword ptr [r13 + r11*4], ymm0   ; store codes
```

Observe that 32 – l(t) is the variable number of bits by which to shift right logical the code segments. Next, we describe how to store l(t). Currently, these lengths occupy the 8 dword lanes of YMM1. The method proposed involves packing these dwords into the LO byte lanes of XMM1. These 8 bytes can then be stored in memory using instruction vmovq qword ptr [r12+r11], xmm1. Figure 5 illustrates how this is done using vpackusdw and vpackuswb, which packs unsigned dwords into words (16 bits), and then unsigned words into bytes, respectively. Just for sake of demonstration, the length values have been written as the integers 1:8; in practice they would be larger values ranging up to 31. Before applying the two packed operations, the bytes in HO dword lanes of YMM1 have been copied into XMM2 using vextracti128.

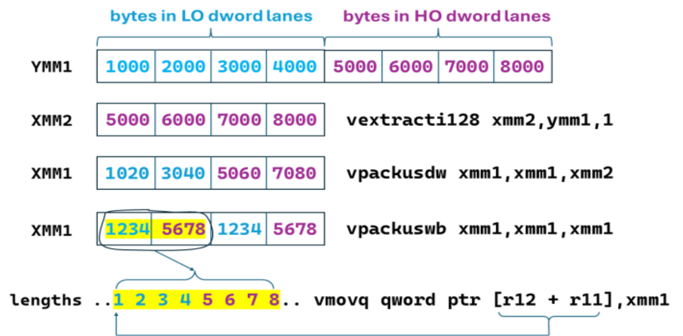


Figure 5: Store Lengths (SPFPAC) Simulated File I/O

Encoder Phase 3 (DPFPAC) Simulated File I/O

Error detection was carried out for DPFPAC. The method can be summarized as follows:

- Write l(t) to lengths unconditionally
- If l(t) < 40, write the binary code value to codes
  Else, copy 5 ASCII symbols from ibuff to codes

To do the latter, we broadcast 40 into the 4 qword lanes of YMM4, and then compare YMM4 to YMM1 to see if 40 > l(t); the results of this comparison are placed in the qword lanes of YMM7 (all 1's if true, all 0's if false). Using YMM7 as a mask to index the correct qword destinations, we conditionally store the "good" code values in array codes at the 256-bit block of memory whose starting address is R13 + R11*8, where 8 denotes the number of bytes allocated for one qword (codes stores the type uint64_t); the instruction that does this is vpmaskmovq [r13+r11*8], ymm7, ymm0. The following code snippet summarizes:

```
vpbroadcastq ymm4, forty
vpcmpgtq ymm7, ymm4, ymm1
vpmaskmovq [r13+r11*8], ymm7, ymm0
```

Next, we copy 5 unencoded ASCII bytes (40 bits) from ibuff to codes at the qword locations where nothing was stored by vpmaskmovq (these 5 symbols are called a *bad block*, even though they are correct). There does not appear to be a direct AVX-256 method to transfer data in groups of 5 at noncontiguous blocks of memory from a source memory operand to a destination memory operand, but we can resort to x86 instruction movsb (move string byte), which can perform the task block by block. The penalty will be negligible since bad blocks (i.e., bad codes) will be extreme outliers and we can perform a quick preliminary test to see if any of the 4 blocks (working in quartets for DPFPAC) are, in fact, bad. The test is done with instruction vptest ymm15, ymm15 (where YMM15 = NOT YMM7), which sets the zero flag if YMM15 is all 0's (exactly when YMM7 is all 1's and all blocks are good). We then use a conditional jump to skip the expensive section of code that checks for bad blocks and executes movsb, if necessary. The following code snippet summarizes:

```
vpcmpeqq ymm15,ymm15,ymm15      ; y15 = 1's
vpxor ymm15,ymm7,ymm15          ; y15 = not(y7)
vptest ymm15,ymm15              ; set ZF if y15 = 0's

jz proceed       ; handle bad blocks only if necessary

    ; . . . . . . . . . . . . . . . . . . . . . . . .
    ;       code to handle bad blocks goes here
    ; . . . . . . . . . . . . . . . . . . . . . . . .

proceed:

    ; write lengths to compressed buffer
    ; proceed to next quartet of blocks
```

The code to handle bad blocks involves extracting qword Boolean mask values, one by one, from YMM7 into 64-bit register RBX and testing for 0 (if so, movsb for the current block being tested is executed). For this task, a macro m_copybadblock was used, which conditionally calls instruction movsb. This macro is shared by decoder (see SI for details).

Decoder Phase 1 (Simulated File I/O)

The decoder was implemented in three ways for SPFPAC, depending on the symbol decoding procedure (one-by-one sequential, vector sequential, and vector binary interval search). The vector sequential method was not implemented for DPFPAC as it will inevitably be too slow for larger alphabets (as will be the case for one-by-one sequential, for that matter). The DPFPAC decoder is like the one for SPFPAC but involves a few extra parameters and is described later. We focus on the function header for SPFPAC decoder:

```
extern "C" void decode1_spfp_avx256(float* C,
int n_octets, uint8_t* lgths, uint32_t* codes,
uint8_t* obuff, float* F);
```

The array C is now the first parameter and so is pointed to by RCX. According to the stack frame layout from Figure 2, registers R8 and R9 now point to lengths and codes. Parameters obuff and F are located on the stack at RSP + 40 and RSP + 48, and were loaded into R12 and R13 for faster access. Phase 1 involves loading codes and code lengths into AVX registers, and then computing the associated tags as FP values. This is illustrated for SPFPAC in Figure 6.

In determining $t = code/2^{l(t)}$, the denominator obtains by accessing a table containing powers of two (not shown in Figure 6) using l(t)-values (as indices) that have been moved from byte-spaced values (in lengths) into dwords in YMM8 for a subsequent gather instruction that performs this access. To prepare for this gather, an octet of lengths is moved to the 8 LO bytes of XMM8 (HO bytes are unused) using vmovq; we move these bytes into dword lanes of YMM8 via vpmovzxbd—move zero-extend byte to dword. The code values (4-byte spacing) are loaded into YMM10 via vmovdqa (the codes array was 32-byte aligned in C++ using _aligned_malloc to facilitate) and are labeled as c1:c8 to avoid writing out generic, lengthy hex values. The integer powers of two in YMM9 and integer codes in YMM10 are converted to SPFP using vcvtdq2ps to facilitate the subsequent SPFP packed division instruction, which calculates $t = code/2^{l(t)}$.

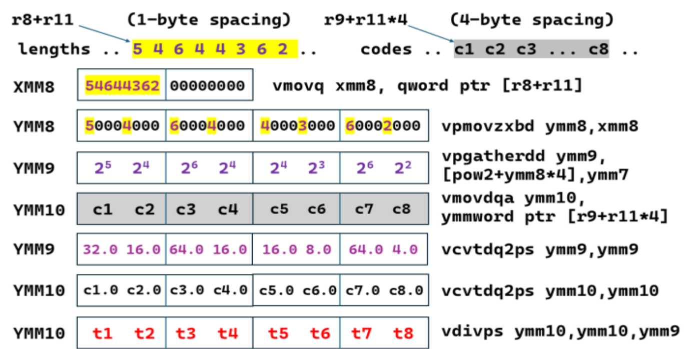


Figure 6: Decoder Phase 1 (SPFPAC) Simulated File I/O

The calculation for DPFP is similar but we process 4 tags in qword lanes instead. We use vmovd xmm8, [r8+r11] to load 4 bytes into the LO 32 bits of XMM8 and follow up with vpmovzxbq ymm8, xmm8 to move these bytes into the 4 qword lanes of YMM8. The instruction to convert 64-bit code values to DPFP does not exist in AVX-256, so we use the bespoke conversion method discussed earlier.

Decoder Phase 2

Given m possible subdivisions of the current interval, we must find the one that contains t, update the interval to the one found, and then write the corresponding symbol to output buffer. For the subdivision that contains t, we have $\alpha + C_{k-1}L \le t < \alpha + C_kL$, which is equivalent to:

$$C_{k-1} \le t^* < C_k \quad \text{where } t^* = (t - \alpha)/L \text{ is the scaled tag}$$

Satisfying the latter inequality involves less work since it is not necessary to compute the interval endpoints of each subdivision. Pseudocode for finding index k of the containing interval follows (for proper decoding with the midpoint method, coincidence of a tag with an endpoint will not occur, so we can use comparison $t^* > C_k$ instead of $t^* \ge C_k$):

- k = 0
  do {k++} while ($t^* > C_k$);
  return k – 1

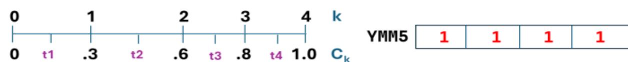
The procedure returns k – 1 since update $\alpha' = \alpha + C_{k-1}L$ must occur before $L' = LF_k$. For vectorized FPAC, implementing a sequential search one tag at a time is inefficient, since a scalar FP comparison involves moving the current scaled tag into the first lane of an AVX register, and one must decide how to save or even discard the remaining tags. In this implementation of decode_symbol, the current scaled tag has been shifted into the first lane of XMM0, and effectively acts as a FP parameter (this procedure is not called by C++, but the pseudo header indicates the two effective parameters):

```
; int decode_symbol(float *C, float tscale);

decode_symbol PROC
mov eax, 0                      ; eax = k

@@: inc eax
vmovss xmm15, real4 ptr [rcx + rax*4]   ; x15[0] = Ck
```

```
vcomiss xmm0,xmm15          ; tscale > Ck ?
ja @B                       ; if so, repeat loop

dec eax                     ; otherwise ..
ret                         ; return k – 1 in eax
decode_symbol ENDP
```

Observe that RCX points to C, and so the scalar SPFP instruction `vmovss xmm15, real4 ptr [rcx + rax*4]` loads the first lane of XMM15 with C_k for the subsequent scalar SPFP comparison `vcomiss`. After the procedure returns, the decoded index k – 1 (returned in EAX) is inserted into the appropriate lane of YMM6, which is eventually used to gather data (C_{k-1} and F_k) for the interval updates. Before decoding the next symbol, the next tag is shifted into the LO lane of XMM0, which kicks out the old one. This tag shifting process can only be done for the lower half of YMM0 (i.e., XMM0), so the two halves are swapped using `vperm2f128 ymm0,ymm0,ymm0,1`. See SI for details.

Shifting and swapping can be avoided by performing the sequential search concurrently on an entire group of tags (8 for SPFPAC). There does not appear to be a simple way to break out of the search early when a match is found, but we can force it to execute all m – 1 times (the last interval can be skipped). For smaller alphabets, this AVX method makes sense as processing all 8 tags in parallel can lower the average number of comparisons below that for processing these tags one by one with early departures. We proceed to the entry point of divisions loop where 8 unscaled tags are stored in YMM10; the scaled tags will be placed in YMM0. The index k (to gather C_k for comparison) will occupy the 8 dword lanes of YMM8. This k is incremented m – 1 times (per each of 5 iterations of divisions loop) and holds the same value in all 8 lanes. We use another register (YMM9) to hold the k-values to be used for subsequent instructions that gather C_{k-1} and F_k, for the interval updates that occur after the current octet of symbols has been decoded; we call these values K. This extra register is needed since we do not want to increment these K's after the associated matches are found.



YMM8	0	0	0	0	vpxor ymm8,ymm8,ymm8 (initialize)
YMM9	0	0	0	0	vpxor ymm9,ymm9,ymm9 (initialize)
YMM8	1	1	1	1	vpaddd ymm8,ymm8,ymm5
	0	1	1	1	YMM9 after 1st iteration
YMM8	2	2	2	2	vpaddd ymm8,ymm8,ymm5
	0	1	2	2	YMM9 after 2nd iteration
YMM8	3	3	3	3	vpaddd ymm8,ymm8,ymm5
	0	1	2	3	YMM9 after 3rd iteration = K – 1

Figure 7: Incrementing K (AVX Sequential Search)

Figure 7 illustrates the contents of YMM8 and YMM9 (only 4 out of 8 lanes are shown) for alphabet size m = 4 and 4 tags t_1:t_4 occupying the first four intervals, respectively.

To implement this method, YMM5 stores 8 copies of constant 1 in dword lanes for incrementing k in YMM8. The results of comparison t* > C_k are placed in YMM11, which is used as a mask to select the lanes of YMM9 to be incremented. The instruction `vpand ymm12, ymm5, ymm11` loads YMM12 with 1 in lanes where K will be incremented (match is not yet found), or with 0 where matches have been found. Once these matches are found, these K's are never incremented again as the comparison t* > C_k will remain false. Code Listing 6 provides an AVX sequence of instructions for vector sequential decoding. The parameter m is located at RSP + 56 on stack.

Code Listing 6. AVX Sequential Search (SPFPAC)

```
 ; calc 8 scaled tags in y0 (y1 = alpha, y2 = L)
vsubps ymm0, ymm10, ymm1    ; y0 = t - alpha
vdivps ymm0, ymm0, ymm2     ; y0 = [t1* ... t8*]

vpxor ymm8,ymm8,ymm8        ; index k to get Ck for cmp
vpxor ymm9,ymm9,ymm9        ; K = decoded index

mov eax, dword ptr [rsp+56]    ; eax = #reps of search
dec eax                        ; only m-1 reps are needed

decode_symbol_avx:
vpaddd ymm8,ymm8,ymm5          ; inc k (to get Ck)
vpcmpeqd ymm7,ymm7,ymm7        ; set gather mask
vgatherdps ymm3, [rcx + ymm8*4], ymm7     ; y3 = Ck

vcmpgtps ymm11,ymm0,ymm3       ; t* > Ck ?
vpand ymm12,ymm5,ymm11         ; mask for addition to K
vpaddd ymm9,ymm9,ymm12         ; add 1 to K if t* > Ck

dec eax
test eax,eax
jnz decode_symbol_avx
      ; end avx sequential search; ret K-1 in y9
```

The third symbol decoding technique is based on the AVX binary interval search; the implementation is virtually identical to the one used for the floor(log2) calculation, where intervals were formed from powers of two in a lookup table; indices for access into the table ranged up to 31 for SPFPAC and up to 40 for DPFPAC. For the AVX binary interval search used by decoder, we search the m intervals formed by the CDF endpoints. The search executes floor(log2(m)) times for each group of symbols encountered (8 for SPFPAC, 4 for DPFPAC). Index High is initialized to m. Both floor(log2(m)) and m are parameters of the decoder function and are passed in by C++.

Decoder Error Detection (DPFPAC) Simulated File I/O

If FP length L is under 1e-12, the calculation of t = code/$2^{l(t)}$ will be erroneous (t > 1 was observed in the debugger in certain cases); this follows since High was initialized to 40 in the encoder log2 calculation, the return value Low becomes 40 at termination, and the likely incorrect value of l(t) = 42 (after addition by 2) is stored in `lengths`. To prevent the decoders from producing unexpected results (e.g., infinite loops, program crashes, and out-of-range symbols were observed), a solution was to set these bad tags to 0 using these three instructions (YMM8 stores the length values):

```
vpbroadcastq ymm4, qword ptr [forty]
vpcmpgtq ymm12, ymm4, ymm8          ; y12 = 40 > l(t) ?

; ...............................................

; (instructions to calc t = code/2l(t) in y10)
```

```
; ...............................................

vpand ymm10,ymm10,ymm12          ; set bad tags to 0
```

Bad scaled tags are then determined as t* ≤ 0 and all symbol decoding procedures will decode each symbol in its bad block as the most frequent symbol (lowest index) s_1, which is the whitespace symbol $(32)_{10}$ in OTF. This causes no concern since these erroneous characters will be overwritten in the output buffer by their correct ASCII codes. The method uses the same macro `m_copybadblock` used by encoder.

Oxygen Transport File (4HHB) Statistics

This text file (OTF) uses m = 53 ASCII symbols. After sorting by non-increasing frequency of occurrence, the alphabet and its statistical summary appear in Table 1.

1	32		0.48055	0.48055	28	71	G	0.00340	0.98345
2	46	.	0.05180	0.53235	29	73	I	0.00280	0.98625
3	49	1	0.05140	0.58375	30	85	U	0.00270	0.98895
4	48	0	0.04420	0.62795	31	80	P	0.00260	0.99155
5	50	2	0.03420	0.66215	32	89	Y	0.00230	0.99385
6	51	3	0.02990	0.69205	33	75	K	0.00160	0.99545
7	52	4	0.02880	0.72085	34	86	V	0.00150	0.99695
8	53	5	0.02410	0.74495	35	70	F	0.00060	0.99755
9	65	A	0.02240	0.76735	36	58	:	0.00040	0.99795
10	54	6	0.02150	0.78885	37	90	Z	0.00030	0.99825
11	55	7	0.02030	0.80915	38	40	(	0.00020	0.99845
12	56	8	0.02000	0.82915	39	41	)	0.00020	0.99865
13	57	9	0.02000	0.84915	40	44	,	0.00020	0.99885
14	67	C	0.01740	0.86655	41	61	=	0.00020	0.99905
15	79	O	0.01660	0.88315	42	81	Q	0.00020	0.99925
16	84	T	0.01430	0.89745	43	88	X	0.00020	0.99945
17	77	M	0.01310	0.91055	44	42	*	0.00010	0.99955
18	10		0.01230	0.92285	45	59	;	0.00010	0.99965
19	69	E	0.01010	0.93295	46	74	J	0.00010	0.99975
20	82	R	0.00740	0.94035	47	87	W	0.00010	0.99985
21	76	L	0.00720	0.94755	48	95	_	0.00010	0.99995
22	78	N	0.00630	0.95385	49	37	%	0.00001	0.99996
23	72	H	0.00570	0.95955	50	43	+	0.00001	0.99997
24	83	S	0.00560	0.96515	51	47	/	0.00001	0.99998
25	45	-	0.00530	0.97045	52	60	<	0.00001	0.99999
26	68	D	0.00510	0.97555	53	62	>	0.00001	1.00000
27	66	B	0.00450	0.98005					

Table 1: Statistical Summary OTF

The five columns identify the sorted alphabet indices 1:53, decimal ASCII codes, ASCII symbols (nothing appears for whitespace and newline), relative frequencies, and associated CDF values. Except for whitespace, frequencies above 5e-5 were rounded to 4 decimal places, those below were estimated as 1e-5. F_1 is determined so that the cumulative frequency sum is 1.00000, as indicated by C_{53}.

The encoder and decoder need to convert between ASCII codes and sorted alphabet indices. The decoder uses a mapping D that converts column 1 (1:53) into column 2 (decimal ASCII); encoder uses a mapping E, the inverse of D, which maps ASCII codes into 1:53. These two mappings are determined during a preliminary C++ file scan and are passed as arguments, as seen in these two function headers for DPFPAC (`decode2` uses the AVX binary interval search):

```
extern "C" void encode(uint8_t* ibuff, int nquads,
double* F, double* C, uint8_t* lgths, uint64_t* codes,
int32_t* E);
```

```
extern "C" void decode2(double* C, int nquads,
uint8_t* lgths, uint64_t* codes, uint8_t* obuff,
double* F, int32_t* D, int m, int log2m);
```

Parameters E and D are located on stack at RSP + 56 and were loaded into R14. After the encoder loads 4 ASCII symbols into the dword lanes of XMM6, the instruction `vpgatherdd xmm4, [r14 + xmm6*4], xmm7` converts ASCII to sorted alphabet indices and loads them into XMM4, which is used to index and gather C and F. The decoder uses a similar gather instruction to convert decoded indices back into ASCII, so they can be stored correctly in `obuff`.

DPFP numbers are accurate up to 16 decimal places, but after calculations, rounding errors reduce reliability. Allowing interval lengths to not drop below $(1e\text{-}12)_{10}$ gave a reasonable factor of safety from precision errors occurring and, according to the midpoint method, bad tags would require over 40 bits as floor(log2(1e-12)) = 39. The OTF consists of N = 473687 total ASCII characters, giving 94737 blocks of size 5 (2 extra whitespace characters at the end of file were ignored to give an even number). Examination of OTF using C++ shows that exactly 254 (fewer than 3ppt) of these blocks have L < 1e-12 and were overwritten by error detection procedure. These bad blocks occur on lines numbered 1 to 881. The total file has 5848 lines, the largest fraction of which consists of whitespaces and numerical data—these blocks all have L > 1e-12, none of which is overwritten. A sample of bad blocks with line numbers and lengths is provided in Table 2

1	OXYGE	2.6222e-13	529	FINED	5.45182e-13
2	XYHAE	5.93201e-13	529	80 +/	4.24806e-14
6	WRONG	2.63123e-13	530	+/- 3	7.61528e-15
7	WRONG	2.63123e-13	546	D, WI	1.37245e-13
11	MOL_I	4.38399e-13	549	MBER;	4.40592e-13
18	B, D;	2.20572e-13	549	NTIFI	4.23783e-13
19	OBIN,	2.63542e-13	550	SSEQ=	1.26694e-14
21	MOL_I	4.38399e-13	550	SEQUE	3.08478e-13
.			.		
.			.		
131	(%) :	7.6888e-17	826	HELIX	2.32122e-13
133	RKING	7.10116e-13	831	HELIX	2.32122e-13
135	%) :	1.84743e-13	836	HELIX	2.32122e-13
146	(A**2	1.53216e-15	841	HELIX	2.32122e-13
147	, A**	2.15286e-14	846	HELIX	2.32122e-13
149	(A**	2.15286e-14	851	HELIX	2.32122e-13
150	*2) :	1.31478e-13	875	RIGX1	7.24205e-13
151	**2)	3.28696e-14	876	ORIGX	2.33887e-13
152	A**2)	1.53216e-15	881	MTRIX	7.76296e-13

Table 2: Sample of Bad Blocks (L < 1e-12) from OTF

Contiguous File I/O

This introductory discussion primarily applies to encoding (phase 3) where we must pack the contents of YMM1 and YMM0 (storing lengths and codes, respectively) into the destination compressed file. Decoding (phase 1) involves reading this file and loading 256-bit registers with lengths and codes, analogously. For simulated file I/O, the net result was to transfer individual 5- or 6-bit length values into an array of 8-bit integers, one length value per integer; an unpacked arrangement with 2 or 3 bits wasted per byte resulted. A similar unpacked arrangement resulted by transferring the codes into an array of 32- or 64-bit integers. These simulated

file I/O techniques boasted parallel data transfer but did not result in a contiguous representation. This problem could be circumvented if the AVX-256 data gather instructions (and *data scatter*, only present in AVX-512) involved bit rather than byte granularity. Then data could be transferred in parallel to and from memory at specified bit-indexed locations and a packed arrangement could be feasible.

So, it appears, as usual, we are forced to use byte-addressable memory access and bit masking; the method provided in the sequel is an adaptation of one found in Hyde [7], wherein a bit string contained in a general-purpose register is inserted via Boolean operations into another one of the same bit width. Although it is mentioned as an exercise left to the reader, the method in [7] does not account for the case when the bit string to be inserted (or extracted) crosses the boundary of the destination (or source) operand (in the sequel, these destinations will be 64-bit array elements); when accounting for said caveat shortly, it appears that the logic involved in performing the task of parallel insertions of multiple bit strings into a destination array is either too complex or not feasible at all, and so the process is done one by one.

In the contiguous file *I/O* model, the compressed file (referred to by its C++ identifier as `cfile`) is declared as an array of 64-bit unsigned integers (which translate into qwords in x86) and is pictured as being divided into two sections: *lengths* at LO memory followed by *codes* at HO memory. Two pointers (RDI and R9) and bit indices (R10 and R11) are used to locate these sections. RDI points to the LO byte of the qword containing the bit location where the next length is to be inserted; R9 points to the LO byte of the qword containing the code, similarly. RDI and R9 are multiples of 8 bytes, the space requirement of multiple qwords. R10 and R11 are the actual insertion locations (taken at the LO bit of the bit string) and belong to the range 0:63 of 64 total bits. RDI is named for the encoding phase, where the array elements of `cfile` are destinations for writing the contents of YMM1 and YMM0. The register RDI is changed to RSI for decoding where `cfile` is a source (its contents are read and transferred into 256-bit registers). R9 is used to locate code values for both encoding and decoding; using integer division by 64, it can be initialized with a byte offset of 8 * [(6 * `nblocks`)/64 + 1]. For example, 10 blocks require 60 bits of length storage, so the offset is 8 bytes; one extra block would make the offset 16 bytes. Figure 8 illustrates `cfile`, its pointers, and two inserted bit strings.

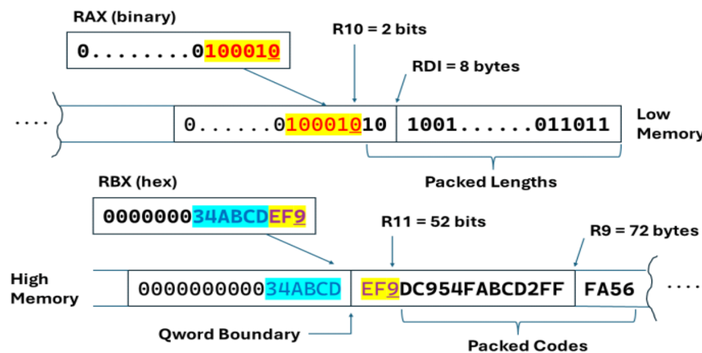


Figure 8: Contiguous Compressed File Representation

Registers RAX and RBX store zero-extended and right-justified length and code values to be inserted at the indicated offsets. Figure 8 shows the register contents before left shifting and the actual insertion (OR operation).

To insert a bit string (length or code) into a qword destination operand (i.e., a `cfile` array element), we must first see if the string fits without crossing the qword boundary separating the next HO qword. A 6-bit length will fit without crossing when R10 ≤ 58; its associated code will fit when length ≤ 64 – R11. The following macro demonstrates how a bit string (stored in generic register `Rsrc`, standing for RAX or RBX) can be inserted at generic bit offset `Rpos` (standing for R10 or R11), and associated file offset `Rfile` (RDI or R9), and assuming our preliminary test indicates it will properly fit.

```
m_write_fits MACRO Rsrc, Rpos, Rfile, inc_val

LOCAL return
mov ecx, Rpos                    ; cl = Rpos
shl Rsrc, cl                     ; shift into position
or qword ptr [Rfile], Rsrc       ; insert into file
add Rpos, inc_val                ; increment pos
cmp Rpos, 64                     ; Rpos = 64?
jne return                ; return if not equal
xor Rpos, Rpos            ; reset Rpos if src just fits
add Rfile, 8              ; inc to next qword
return:

ENDM
```

The first step is to left shift the source bit string (`Rsrc`) so its LO bit occupies the bit insertion location (`Rpos`). The next step is to OR the shifted string into destination operand (pointed to by `Rfile`). Finally, we increment `Rpos` (by 6 for length values, or by the length itself for codes) and check to see if the source string has *just* fit into the destination, which occurs exactly when its HO bit becomes bit 63—if and only if the incremented `Rpos` becomes 64—in which case we reset `Rpos` to 0 and point `Rfile` to the next qword, 8 bytes away.

If the test indicates that the source string will not fit, we parse `Rsrc` into two parts: a chopped off HO segment to be inserted into the next qword (i.e., at `Rfile` + 8); and a chopped LO segment that just fits starting at location `Rpos` in the current qword (i.e., at `Rfile` itself). This task is illustrated in Figure 9; the companion macro follows:

Figure 9: Bit-String Insertion (Qword Boundary Crossed)

```
m_write_nofit MACRO Rsrc, Rpos, Rfile, new_pos

xor edx, edx             ; clear rdx
mov ecx, Rpos            ; cl = Rpos
shld rdx, Rsrc, cl       ; rdx = chopped HO segment
shl Rsrc, cl             ; Rsrc = chopped LO segment
or qword ptr [Rfile], Rsrc     ; insert LO segment
add Rfile, 8                   ; inc to next qword
mov qword ptr [Rfile], rdx     ; insert HO segment
mov Rpos, new_pos        ; set new bit position

ENDM
```

Shift left double (`shld`) left shifts its destination (RDX) by the number of bits in the third operand (using CL, the LO byte of ECX); the bit positions opened up by the shift are filled with the most significant bits of source operand (Rsrc). In context, the effect is to shift into RDX the HO segment of the chopped-off bit string in attempting to fit the entirety starting at location `Rpos`. The LO segment of the chopped string resides at the HO end of `Rsrc` after the instruction `shl Rsrc, cl`. These two segments can now be inserted into qword destinations at `Rfile` and `Rfile` + 8 (using `or` and `mov`). The last step is to update `Rpos` with `new_pos` (passed in by parameter).

Encoder Phase 3 (Contiguous File I/O)

Code Listing 7 simulates phase 3 (see SI for the actual DPFPAC program). To run the program, one needs to insert the macro definitions `m_write_fits` and `m_write_nofit` (omitted to save space) and call procedure `encode_demo` from C++. Three sets of lengths and codes quartet data are provided to build a suitably large compressed file (`cfile` was declared arbitrarily as an array of 8 qwords). The input message (declared as `ibuff`, an array of 60 bytes) is contrived and is not encoded or connected in any way to the `lengths` and `codes` arrays. Rather, `ibuff` is provided to demonstrate the error handling procedure. Referring to the macro `m_write_lgth_code`, a pair of length and code values is loaded from YMM1 and YMM0 into RAX and RBX. If RAX ≥ 40, RBX is overwritten with 8 bytes from the input buffer (`ibuff` is pointed to by RSI, indicating its role as a source); the HO 3 bytes are then masked out so that RBX only stores the correct 5 bytes from `ibuff`. From here, the appropriate tests are applied to determine if the 6-bit value in RAX will fit into the lengths section of `cfile`, and if the code value in RBX will fit into the codes section. Depending on the outcomes of these tests, macros `m_write_fits` and `m_write_nofit` are called, accordingly. Procedure `encode_demo` drives the entire program; it begins by initializing RSI, RDI, and R9 to the starting addresses of `ibuff`, `cfile` (lengths section) and `cfile` (codes section); the bit positions R10 and R11 are then initialized to 0. Register RBP is awkwardly used (no more general-purpose registers are left) to index the `lengths` and `codes` arrays for data transfer into YMM1 and YMM0 (RBP has no role in DPFPAC and exists only for demonstration); the same goes for memory operand `nreps_encode`, which exists only to count 3 full encoding phases.

Code Listing 7. Compressed File Simulation (Encoder)

```
.data
seg segment align(32)
lgths qword 13, 14, 14, 15, 40, 9, 10, 11, 8, 9, 10, 45
codes qword 1aaah, 2bbbh, 3ccch, 4dddh,
        9999999999h, 123h, 234h, 412h,
        12h, 123h, 234h, 199999999999h
cfile qword 8 dup(?)
seg ends

nreps_encode dword 3
ibuff byte 1,2,3,4,5,6,7,1,2,3,1,2,3,4,5,6,7,1,2,3
      byte 1,2,3,4,5,6,7,1,2,3,1,2,3,4,5,6,7,1,2,3
      byte 1,2,3,4,5,6,7,1,2,3,1,2,3,4,5,6,7,1,2,3
msk5bytes qword 0ffffffffffh
.code
; cut and paste here:
; m_write_fits MACRO Rsrc, Rpos, Rfile, inc_val
; m_write_nofit MACRO Rsrc, Rpos, Rfile, new_pos

m_write_lgth_code MACRO _01

LOCAL else1, else2, endif_1, endif_2, proceed
vpextrq rax, xmm1, _01          ; rax = lgths[_01]
vpextrq rbx, xmm0, _01          ; rbx = codes[_01]
cmp eax, 40                     ; length >= 40 ?
jb proceed                      ; if not, move along
mov eax, 40                     ; change length to 40
mov rbx, qword ptr [rsi]        ; load 8 bytes ibuff
and rbx, [msk5bytes]            ; keep only LO 5 bytes
proceed:                        ; move along
add rsi, 5              ; inc ibuff index in any case
mov r15d, eax           ; r15 = copy of lgth
cmp r10d, 58            ; posL <= 58 ?
ja else1                ; if not, jump to label else1
m_write_fits rax, r10d, rdi, 6
jmp endif_1
else1:                  ; lgth crosses the border
mov r14d, r10d          ; store posL
sub r14d, 58            ; new posL = old posL - 58
m_write_nofit rax, r10d, rdi, r14d
endif_1:
mov edx, 64
sub edx, r11d           ; edx = 64 - posC
cmp r15d, edx           ; lgth <= 64 - posC ?
ja else2                ; if not, jump to label else2
m_write_fits rbx, r11d, r9, r15d
jmp endif_2
else2:                  ; code crosses the border
sub r15d, edx           ; new posC = lgth - (64 - posC)
m_write_nofit rbx, r11d, r9, r15d
endif_2:

ENDM

encode_demo PROC

lea rsi, ibuff          ; rsi = ibuff index
lea rdi, cfile          ; rdi = lgths ptr
lea r9, [cfile+16]      ; r9 = codes ptr
mov r10d, 0             ; r10 = lgth bit pos (posL)
mov r11d, 0             ; r11 = code bit pos (posC)
push rbp                ; just for demo sake
mov rbp, 0              ; idx lgths and codes

@@:
vmovdqa ymm0, ymmword ptr [codes+rbp]
vmovdqa ymm1, ymmword ptr [lgths+rbp]
m_write_lgth_code 0     ; write lgth[0] and code[0]
m_write_lgth_code 1     ; write lgth[1] and code[1]
vextracti128 xmm0, ymm0, 1     ; shift HO to LO
vextracti128 xmm1, ymm1, 1     ; shift HO to LO
m_write_lgth_code 0     ; write lgth[2] and code[2]
m_write_lgth_code 1     ; write lgth[3] and code[3]
add rbp, 32             ; index next quartet
dec nreps_encode
cmp nreps_encode, 0
jnz @B
```

```
pop rbp         ; restore or corrupt stack
ret
encode_demo ENDP
; hex dump of memory block at address cfile:
; 8d e3 3c 68 a2 2c 48 a2 a0 00 00 00 00 00 00 00
; aa 7a 77 65 e6 bb 9b 01 02 03 04 05 23 69 94 a0
; c4 48 1a 0d 0e 02 04 06 00 00 . .
```

In this loop, YMM1 and YMM0 are loaded with the next quartets. Then `m_write_lgth_code` executes 4 times. The parameter for this macro is either 0 or 1 to indicate which qword lane is to be extracted into RAX and RBX. The two HO lanes cannot be directly accessed so `vextracti128` replaces the two LO qword lanes beforehand. The C++ interface to this program did not use a display; instead, we can examine the hex dump (provided in little-endian format) of `cfile` after the code listing.

Decoder Phase 1 (Contiguous File I/O)

This phase reverses what the encoder did in its final phase; the decoder reads `cfile` and loads the appropriate YMM's in preparation for divisions and symbol decoding. Three macros are used, analogous to the ones for writing. Their headers are:

```
; m_read_fits MACRO Rdest, Rpos, Rfile, inc_val
; m_read_nofit MACRO Rdest, Rpos, Rfile, msksze
; m_read_lgth_code MACRO _01, x0, x1
```

The first two macros read the required bit string located in `cfile` and load `Rdest` (RAX or RBX) as the return parameter. The third macro drives the first two and loads the 128-bit generic XMM registers (`x0` and `x1`) with RBX and RAX in generic qword lanes (`_01`) as 0 or 1. `Rpos`, `Rfile`, and `inc_val` have the same semantics as their encoder analogs. Of note, both the `fit` and `nofit` macros require a mask to remove yet-to-be-read segments of lengths or codes that contaminate the right-justified values in RAX and RBX after shifting the current bit strings into place; this contamination was absent (as 0's) for writing to `cfile`, which was initialized to all null (ASCII 0) chars, so a mask for writing was avoided. These masks are accessed via their required sizes (passed in generic parameters `inc_val` and `msksze`) using the following lookup table containing 41 qwords (masks are accessed one at a time so 32-byte alignment is not required):

```
msk qword 0, 1, 3, 7, 0fh, 01fh, 03fh, 07fh, 0ffh,
1ffh, 3ffh, 7ffh, 0fffh, 1fffh, 3fffh, 7fffh, 0ffffh,
1ffffh, 3ffffh, 7ffffh, 0fffffh,
1fffffh, 3fffffh, 7fffffh, 0ffffffh,
1ffffffh, 3ffffffh, 7ffffffh, 0fffffffh,
1fffffffh, 3fffffffh, 7fffffffh, 0ffffffffh,
1ffffffffh, 3ffffffffh, 7ffffffffh, 0fffffffffh,
1fffffffffh, 3fffffffffh, 7fffffffffh, 0ffffffffffh
```

The decoder uses the following snippet for execution:

```
m_read_lgth_code 0, xmm3, xmm8    ; y3/8 = [* 0 0 0]
m_read_lgth_code 1, xmm3, xmm8    ; y3/8 = [* * 0 0]
m_read_lgth_code 0, xmm4, xmm6    ; y4/6 = [* 0 0 0]
m_read_lgth_code 1, xmm4, xmm6    ; y4/6 = [* * 0 0]
vinserti128 ymm3, ymm3, xmm4, 1   ; y3 = [c1 c2 c3 c4]
vinserti128 ymm8, ymm8, xmm6, 1   ; y8 = [l1 l2 l3 l4]
```

YMM3 and YMM8 are the destinations for codes and lengths, while XMM4 and XMM6 are temporary registers to store the two HO lanes, which are then inserted into the upper halves of YMM3 and YMM8 to complete a full quartet. The DPFP tags can now be computed the same way as in Figure 6 (YMM3 acts as YMM10 in the figure), where 4 qword lanes are processed rather than 8 dword lanes. Moreover, YMM3 stores a copy of codes—5 ASCII symbols when $l(t) \geq 40$—so the decoder can bypass a costly second access of `cfile` during error detection, which occurs after the divisions loop.

Decoder Error Detection (Contiguous File I/O)

Like in simulated file I/O, register YMM12 contains Boolean values signaling whether lanes in YMM3 correspond to good codes—$l(t) < 40$; bad scaled tags were computed as $t^* \leq 0$, which caused 5 whitespace symbols to be erroneously written to `obuff`. After the divisions loop has completed, we check to see if any blocks are bad, just like in simulated file I/O. If so, a section of code overwrites these wrong symbols with the associated unencoded 5 ASCII symbols stored in YMM3. The following macro and code snippet show how this is done:

```
m_writebadblock MACRO xmmBool, xmmcodes, _01

LOCAL nextblock
vpextrq rax, xmmBool, _01       ; extract Bool into rax
test rax, rax                   ; rax = 0 (bad)?
jnz nextblock                   ; if not, go to nextblock
mov rax, qword ptr [mskout5]    ; load mask
and qword ptr [rdi], rax        ; clear out bad 5 bytes
vpextrq rax, xmmcodes, _01      ; load 5 ASCII chars
or qword ptr [rdi], rax         ; insert into obuff
nextblock:

add rdi, 5      ; point obuff to next block in any case

ENDM

m_writebadblock xmm12, xmm3, 0    ; write c1 if bad
m_writebadblock xmm12, xmm3, 1    ; write c2 if bad
vextracti128 xmm12,ymm12,1       ; shift down Bools
vextracti128 xmm3,ymm3,1         ; shift down c3 and c4
m_writebadblock xmm12, xmm3, 0    ; write c3 if bad
m_writebadblock xmm12, xmm3, 1    ; write c4 if bad
```

Performance of SPFPAC and DPFPAC

Time-testing experiments were done on a 12th generation bargain-brand Intel® Core™ i3-1215u processor having a codename formerly known as Alder Lake. The base processor speed is reported by the manufacturer as 1.2GHz. This chip model is equipped with AVX and AVX2 (128- and 256-bit registers), but not AVX-512. Said information was verified by three independent sources: the first being the out-of-box specs (the processor name, clock speed, and memory features were provided) of Lenovo® brand computer, purchased at Office Depot, Inc., for 320 dollars; secondly, the Intel® Processor Identification Utility 7.1.6, downloadable from company website; and thirdly, via an x86 AVX CPU identification utility program courtesy of Kusswurm [6]—see Ch16_02—based on the CPUID instruction, which retrieves various information such as the model number, the sizes of internal caches, and AVX instruction set features. Running this program gave the following output:

```
GenuineIntel
12th Gen Intel(R) Core(TM) i3-1215U @ 1.20GHz
Cache L1: 48KB Data
Cache L1: 32KB Instruction
Cache L2: 1MB Unified
Cache L3: 10MB Unified

----- CPUID Feature Flags -----
ADX:        1
AVX:        1
AVX2:       1
AVX512F:    0  ; more zero AVX-512 flags follow
```

The AVX-512 letter-F extension stands for "foundation" [8], and when this is zero, all other AVX-512 flags will be so, indicating no capability or absence (these zeros for the extensions FMA, BMI, etc., are omitted).

Time testing in C++ was performed using both `<chrono>` and the function `clock()` contained in `<time.h>`; these report *wall time* and not CPU time. Both `<chrono>` and `<time.h>` gave the same results to the nearest tenth of a millisecond and it was decided to incorporate `<time.h>` in the SI. For all tests performed, an average time was taken by running 10 consecutive trials and dividing net time by 10. The output of `clock()` is in units of milliseconds, so division by macro constant `CLOCKS_PER_SEC` = 1000 was used to report answers in seconds. In typical sequences of 5 or so program runs, the shortest time observed was repeatably obtained on separate days, and this fastest time is the one reported without resorting to standard deviations or confidence intervals. Sample variation in x86 FPAC times was on the order of a few tenths of a millisecond (an exception being for the slower one-by-one sequential decoder applied to the larger alphabet (m = 53), which showed variation of nearly 10ms); Huffman coding times (described in Experiment 4) showed variation of about 5ms.

Experiment 1. Log2 Calculation SPFP (x87 versus AVX-256)

To test Code Listing 2 (x87 log2) against Listing 4 (AVX log2), an array of 33 SPFP L-values x, x + $(1e\text{-}6)_{10}$, and x – $(1e\text{-}6)_{10}$ where x = $1/2^k$ (k = 8:18) was created; the value of 1/x coincides with a power of two, so the binary interval search operates at or near the endpoints of its lookup table. This array was duplicated 7544 times to serve as the `inputs` array (of 31119 octets) to the two x86 procedures; the total number of logs nearly matches the number (12^5) calculated in a shortly described SPFPAC experiment. Both methods were tested without error, as compared to the C++ value of `abs((int) log2(L))`—this one-by-one time is reported as well:

```
cpp time: 0.014   (248952 truncated logs, one by one)
x87 time: 0.0084
avx time: 0.0012
```

Experiment 2. SPFPAC Encoder and Decoder Performance

An alphabet of size m = 12 (letters a:l) was chosen; all 12^5 = 248832 blocks of size 5 were concatenated into one message (1.24MB), which forms 31104 octets, each consisting of 40 symbols to be processed in one encoding or decoding phase. Since all letters are equally likely, an equiprobable distribution results, where F = 1/12 for each symbol giving H(S) = log2(12) = 3.58. Adding 1.4 gives 4.98 as an upper bound for <l(s)>. For each block, L = $(1/12)^5$, which requires 5 + (2 + floor(log2(L^{-1}))) = 24 bits. So SPFPAC has an efficiency of 24/5 = 4.80 ≤ 4.98 bits per symbol for this message. AVX log2 encoder was tested alongside three decoders (no errors resulted, inferred by comparing `ibuff` to `obuff`, symbol by symbol).

Time testing results for simulated file I/O follow:

```
encode time: 0.0016    AVX log2
decode time: 0.0054    one-by-one sequential search
decode time: 0.0026    AVX sequential search
decode time: 0.0066    AVX binary interval search
```

As one might expect, the AVX sequential search decoder is the fastest; it runs 8 independent searches concurrently, where each goes the distance of 11 repetitions. The AVX binary interval search is the slowest due to the overhead of, albeit in only floor(log2(12)) = 3 repetitions, a palpable accumulation of Boolean operations; but this seems unavoidable, where for m = 53 (OTF), only 2 more repetitions are needed, and the sequential methods slow down considerably.

Experimenting with SPFPAC shows that decoding errors manifest when L drops below $(1e\text{-}6)_{10}$. Using 1e-6 (21 bits) or even 1e-7 (25 bits) as a cutoff to encoding a block of size 5 does not guarantee an upper bound of 1.4 bits above the entropy (40 bits need to be transmitted). Although one can reduce the block size, this deleteriously impacts the upper bound. Then SPFPAC appears to have limited use with an exception for small alphabets and messages not containing infrequent symbols occurring in close proximity.

Experiment 3. DPFPAC Performance (OTF)

OTF is a 0.474MB (473687 ASCII characters) text file and was tested using two encoders (x87 log2 and AVX log2) and two decoders (one-by-one sequential search and AVX binary interval search). No errors resulted. This file was compressed to 4.87 bits per symbol (H(S) = 3.37). This efficiency was drawn by a C++ program that compiles data such as a list of the bad blocks and the entropy. During a traversal of the blocks, C++ determines DPFP interval length (L) and accumulates the bit value 2 + floor(log2(L^{-1})) (if < 40), or just adds 40 to simulate error detection, otherwise. The total accumulation is 1741453 bits; division by the number of blocks of size 5 (94737 rounded) gives an average bit per block count of 18.38. Adding 6 to account for code length storage gives 24.38 bits per block, or 4.87 per symbol.

Time testing results for simulated file I/O follow:

```
encode time: 0.0023    x87 log2
encode time: 0.0010    AVX log2
decode time: 0.0058    AVX binary interval search
decode time: 0.07      one-by-one sequential search
```

Time testing results for contiguous file I/O follow:

```
encode time: 0.0018    AVX log2
decode time: 0.0070    AVX binary interval search
decode time: 0.07      one-by-one sequential search
```

Assuming simulated file I/O times can represent a baseline or gauge, the penalty in constructing the contiguous file is much more severe for encoding (nearly a factor of 2), where the burden of symbol decoding and its load of DPFP comparisons is absent. The difference in the two times (0.0010 versus 0.0018) is about 1ms, slightly less than the difference in the two binary search decoder times (0.0058 versus 0.0070), which is merely several percentage points of difference. No significant difference was observed between the two sequential search times (both at 0.07s, but with much higher variation between trials), although as mentioned, a detailed statistical study was not performed.

Experiment 4: Time-Testing FPAC against Huffman Coding

When compared to conventional AC methods, raising a lower bound by a full bit above the entropy is a substantial compromise of efficiency. If an improvement in performance is not gained, one could deem vectorized FPAC as having little value, other than educational. Informal testing against unoptimized C++ implementations of integer AC, very similar to those found in [1,5], shows over two orders of magnitude of improvement. A better test appears to be against Huffman coding, known to strongly outperform integer AC under similar input conditions, non-adaptive in this case.

To perform this test, a commercially available [9] C++ implementation of Huffman coding was run on the current Alder Lake processor. In addition to the actual compression routine, the program comes with tree and heap ADT interfaces, which were combined via cutting and pasting from the the GITHUB® repository into one (.cpp) file. Much unnecessary safeguarding (e.g., testing for null pointers) was eliminated in route to compiling the final program (available in SI). In comparison to FPAC, glaringly poor times were initially noticed, which was attributed to inappropriate usage, in context, of dynamic memory; C++ function realloc is used by [9] to resize the compressed `buffer` and the recovered message with each extra byte of memory required (the former during encoding, the latter during decoding). This led to an abysmal drop in the expected performance (e.g., OTF was encoded and decoded in about a half second). Usage of `realloc` was omitted in favor of a single dynamic allocation (via `malloc`), identical with the ones used for `cfile` and `obuff` in FPAC. The following snippet shows the modification (260 accounts for the size, in bytes, of a shortly described compressed file header—note that more memory than required is allocated for the compressed file):

```
uint8_t* cfile = (uint8_t*)malloc(N + 260);

// the encoder then runs with no dynamic reallocation

uint8_t* obuff = (uint8_t*) malloc(N);

// the decoder then runs with no dynamic reallocation
```

The remaining C++ program is still unoptimized; in particular, the bit I/O interface uses two functions (`bit_set` and `bit_get`), which set up stack frames and use masking operations in a bit-by-bit manner. Given that FPAC processes entire bit strings, each requiring one or two masking operations, an advantage to FPAC is gained. Adjusting Huffman encoder to process bit strings was not done in this study, but could be a next direction.

Like in FPAC, Huffman encoding requires a preliminary file scan to determine the frequencies of each symbol. They are scaled (see [9] for details) and placed in a 260-byte file header for the compressed file (the input message size, in bytes, takes up the first 4 bytes); the encoding clock is turned on after this has been done to make a fair comparison to FPAC. The encoder then builds a Huffman tree using a priority queue (implemented as a heap array) in order $m \times \log2(m)$ many steps. After the tree is built, a random-access table storing the Huffman symbol codes is built. The encoder now scans each symbol in the input buffer, looks up its code, and writes it, bit by bit, to the compressed file. Upon completion, the clock is turned off and the encoding time reported.

Decoding begins by reading the compressed file header and building the array of frequencies (no header was used for FPAC, the model is simply made available to the encoder and decoder); to compare to FPAC, the decoding clock is subsequently turned on. The decoder then rebuilds the same Huffman tree used to compress the data. After this, it reads the compressed file, bit by bit. Starting at the root, the tree is traversed according to the encountered bit sequence (0 left, 1 right) until a leaf node is encountered, which contains the ASCII code for the next symbol to be written to the output buffer. After writing the symbol, we reposition ourselves at the root and repeat the process. Upon completion, the clock is turned off and the decoding time reported.

Huffman coding was used to test the two messages from experiments 2 and 3. About 5ms variation in these times was observed, the best of which is reported. For the 1.24MB message of 12^5 blocks of size 5 (alphabet size 12):

```
encode time:0.055    decode time:0.042
```

The Huffman encoder compresses this message from 1.24MB down to 0.571MB, or 3.68 bits per symbol (as compared to 4.80 for SPFPAC and H(S) = 3.58). Looking back, one sees that SPFPAC is several times faster. But the alphabet size is small and FPAC decoders are not yet substantially hindered.

For OTF (0.474MB, size 53 alphabet):

```
encode time: 0.025    decode time: 0.017
```

OTF is compressed by Huffman down to 0.204MB (or 3.44 bits per symbol, as compared to 4.87 for DPFPAC and H(S) = 3.37). For this larger alphabet, the FPAC decoder (AVX binary search, contiguous file I/O) only outperforms Huffman's by around a factor of 2. These results are not surprising since the FPAC decoders are strongly affected by alphabet size, whereas the Huffman decoder is relatively insensitive. Optimizations of Huffman coding (e.g., advanced bit I/O, perhaps via

an x86 interface) should lead to a substantial increase in performance. But then again, so would an implementation of FPAC using AVX-512.

References

1. Johnson Jr, P. D., Harris, G. A., & Hankerson, D. C. (2003). Introduction to information theory and data compression. Chapman and Hall/CRC.
2. Sayood, K. (2017). Introduction to data compression. Morgan Kaufmann.
3. Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9), 1098-1101.
4. Shannon, C. E. (1948). A mathematical theory of communication. The Bell system technical journal, 27(3), 379-423.
5. Witten, I. H., Neal, R. M., & Cleary, J. G. (1987). Arithmetic coding for data compression. Communications of the ACM, 30(6), 520-540.
6. Kusswurm, D. Modern X86 Assembly Language Programming. APRESS®, 2018.
7. Hyde, R. (2021). The Art of 64-Bit Assembly, Volume 1: x86-64 Machine Organization and Programming. No Starch Press.
8. Intel® (64). and IA-32 Architectures Software Developer's Manual-Volume 1: Basic Architecture.
9. Loudon, Kyle. Mastering Algorithms with C. O'Reilly®, 1999.
10. Mike H. B. Gray. (2024). Implementation of Floating- Point Arithmetic Coding using x86-64 AVX-256 Assembly Language. Authorea Preprints.

Copyright: *©2024 Mike H.B. Gray. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.*