Shellcode: Recycling Compression Algorithms for the Z80, 8088, 6502, 8086, and 68K Architectures.

modexp.wordpress.com/2020/05/27/komposite-shellcode

By odzhan May 27, 2020

Recycling Compression Algorithms for the Z80, 8088, 6502, 8086, and 68K Architectures.

Contents

1. Introduction

My last post about compression inadvertently missed algorithms used by the Demoscene that I attempt to correct here. Except for research by Introspec about various 8-Bit algorithms on the ZX Spectrum, it's tricky to find information in one location about compression used in Demoscene productions. The focus here will be on variations of the Lempel-Ziv (LZ) scheme published in 1977 that are suitable for resource-constrained environments such as 8, 16, and 32-bit home computers released in the 1980s. In executable compression, we can consider LZ an umbrella term for LZ77, LZSS, LZB, LZH, LZARI, and any other algorithms inspired by those designs.

Many variations of LZ surfaced in the past thirty years, and a detailed description of them all would be quite useful for historical reference. However, the priority for this post is exploring algorithms with the best ratios that also use the least amount of code possible for decompression. Considerations include an open-source compressor and the speed of compression and decompression. However, some decoders without sources for a compressor are also useful to show the conversion between architectures.

Drop me an email, if you would like to provide feedback on this post. x86 assembly codes for some of algorithms discussed here may be found here.

2. History

Designing a compression format requires trade-offs, such as compression ratio, compression speed, decompression speed, code complexity, code size, memory usage, etc. For executable compression in particular, where the sum of decompression code size and compressed size is what counts, the optimal balance between these two depends on the intended target size. – Aske Simon Christensen, author of Shrinkler and co-author of Crinkler.

Since the invention of telegraphy, telephony, and especially television, engineers have sought ways to reduce the bandwidth required for transmitting electrical signals. Before the invention of analog-to-digital converters and entropy coding methods in the 1950s, compaction of television signals required reducing the quality of the video before transmission, a technique that's referred to as *lossy* compression. Many publications on compressing television signals surfaced between the 1950s-1970s, and these eventually proved to be useful in other applications, most notably for the aerospace industry.

For example, various interplanetary spacecraft launched in the 1960s could record data faster than what they could transmit to earth. And following a review of unclassified space missions in the early 1960s, in particular, the Mariner Mars mission of 1964, NASA's Jet Propulsion Laboratory examined various compression methods for acquiring images in space. The first unclassified spacecraft to use image compression was Explorer 34 or Interplanetary Monitoring Platform 4 (IMP-4) launched in 1967. It used Chroma subsampling, invented in the 1950s specifically for color television. This method, which eventually became part of the JPEG standard, would continue being used by NASA until the invention of a more optimal encoding method called Discrete Cosine Transform (DCT)

The increase of computer mainframes in the 1950s and the collection of data on citizens for social science motivated prior research and development of *lossless* compression techniques. Microprocessors became inexpensive in the late 1970s, leading the way for average consumers to purchase a computer of their own. However, this didn't immediately reduce the cost of disk storage. And the vast majority of user data remained stored on magnetic tapes or floppy diskettes rather than hard disk drives offered only as an optional component.

Hard disk drives remained expensive between 1980-2000, encouraging the development of tools to reduce the size of files. The first program to compress executables on the PC was Realia Spacemaker, which was written by Robert Dewar and published in 1982. The precise algorithm used by this program remains undocumented. However, the year of publication would suggest it uses Run-length encoding (RLE). Qkumba informed me about two things via email. First, games for the Apple II used RLE in the early 1980s for shrinking images used as title screens. Examples include Beach-Head, G.I. Joe and Black Magic, to name a few. Second, games by Infocom used Huffman-like text compression. Microsoft EXEPACK by Reuben Borman and published in 1985 also used RLE for compression.

Haruhiko Okumura uploaded an implementation of the LZSS compression algorithm to a Bulletin Board System (BBS) in 1988. Inspired by Okumura, Fabrice Bellard published LZEXE in 1989, which appears to be the first executable compressor to use LZSS.

3. Entropy Coding

Samuel Morse published his coding system for the electrical telegraph in 1838. It assigned short symbols for the most common letters of an alphabet, and this may be the first example of compression used for electrical signals. An entropy coder works similarly. It removes redundancy by assigning short codewords for symbols occurring more frequently and longer codewords for symbols with less frequency. The following table lists some examples.

Type	Publication and Author
Shannon	A Mathematical Theory of Communication published in 1948 by Claude E. Shannon.
Huffman	A Method for the Construction of Minimum Redundancy Codes published in 1952 by David A. Huffman.
Arithmetic	Generalized Kraft Inequality and Arithmetic Coding published in 1976 by Jorma Rissanen.
Range	There are two papers of interest here. One is Source Coding Algorithms for Fast Data Compression published in 1976 by Richard Clark Pasco. The other is Range encoding: An Algorithm for Removing Redundancy from a Digitised Message published in 1979 by G.N.N. Martin.
ANS	Asymmetric Numeral Systems: Entropy Coding Combining Speed of Huffman Coding with Compression Rate of Arithmetic Coding published in 2014 by Jarosław Duda.

Arithmetic or range coders fused with an LZ77-style compressor result in high compression ratios and compact decompressors, which makes them attractive to the demoscene. They are slower than a Huffman coder, but much more efficient. ANS is the favored coder used in mission-critical systems today, providing efficiency and speed.

4. Universal Code

There are many variable-length coding methods used for integers of arbitrary upper bound, and most of the algorithms presented in this post use Elias gamma coding for the offset and length of a match reference. The following table contains a list of papers referenced in Punctured Elias Codes for variable-length coding of the integers published by Peter Fenwick in 1996.

Coding	Author and publication
Golomb	Run-length encodings published in 1966 by Solomon W. Golomb.
Levenshtein	On the redundancy and delay of separable codes for the natural numbers. published in 1968 by Vladimir I. Levenshtein.
Elias	Universal Codeword Sets and Representations of the Integers published in 1975 by Peter Elias.
Rodeh-Even	Economical Encoding of Commas Between Strings published in 1978 by Michael Rodeh and Shimon Even.
Rice	Some Practical Universal Noiseless Coding Techniques published in 1979 by Robert F. Rice.

5. Lempel-Ziv (LZ77/LZ1)

Designed by Abraham Lempel and Jacob Ziv and described in A Universal Algorithm for Sequential Data Compression published in 1977. It compresses files by searching for the repetition of strings or sequences of bytes and storing a reference pointer and length to an earlier occurrence. The size of a reference pointer and length will define the overall speed of the compression and compression ratio. The following decoder uses a 12-Bit reference pointer (4096 bytes) and 4-Bit length (16 bytes). It will work with a a compressor written by Andy Herbert. However, you must change the compressor to use 16-bits for a match reference. Charles Bloom discusses small LZ decoders in a blog post that may be of interest to readers.

```
uint32_t lz77_depack(
  void *outbuf,
  uint32_t outlen,
  const void *inbuf)
{
    uint32_t ofs, len;
    uint8_t  *in, *out, *end, *ptr;

    in  = (uint8_t*)inbuf;
    out = (uint8_t*)outbuf;
    end = out + outlen;

    while(out < end) {
      len = *(uint16_t*)in;
      in += 2;
      ofs = len >> 4;

      // offset?
      if(ofs) {
        // copy reference
        len = (len & 15) + 1;
        ptr = out - ofs;
        while(len--) *out++ = *ptr++;
      }
      // copy literal
      *out++ = *in++;
    }
    // return depacked length
    return (out - (uint8_t*)outbuf);
}
```

The assembly is optimized for size, currently at 54 bytes.

```
lz77_depack:
_lz77_depack:
    pushad

    lea    esi, [esp+32+4]
    lodsd
    xchg   edi, eax           ; edi = outbuf
    lodsd
    lea    ebx, [eax+edi]     ; ebx = outlen + outbuf
    lodsd
    xchg   esi, eax           ; esi = inbuf
    xor    eax, eax
lz77_main:
    cmp    edi, ebx           ; while (out < end)
    jnb    lz77_exit

    lodsw                     ; ofs = *(uint16_t*)in;
    movzx  ecx, al            ; len = ofs & 15;
    shr    eax, 4             ; ofs >>= 4;
    jz     lz77_copybyte

    and    ecx, 15
    inc    ecx                ; len++;
    push   esi
    mov    esi, edi           ; ptr = out - ofs;
    sub    esi, eax
    rep    movsb              ; while(len--) *out++ = *ptr++;
    pop    esi
lz77_copybyte:
    movsb                     ; *out++ = *src++;
    jmp    lz77_main
lz77_exit:
    ; return (out - (uint8_t*)outbuf);
    sub    edi, [esp+32+4]
    mov    [esp+28], edi
    popad
    ret
```

6. Lempel-Ziv-Storer-Szymanski (LZSS)

Designed by James Storer, Thomas Szymanski, and described in Data Compression via Textual Substitution published in 1982. The match reference in the LZ77 decoder occupies 16-bits or two bytes even when no match exists. That means for every literal are two additional redundant bytes, which isn't very efficient. LZSS improves the LZ77 format by using one bit to distinguish between a match reference and a literal, and this improves the overall compression ratio. Introspec informed me via email the importance of this paper in describing the many variations of the original LZ77 scheme. Many of which remain unexplored. It also has an overview of the early literature, which is worth examining in more detail. Haruhiko Okumura shared his implementations of LZSS via a BBS in 1988, and this inspired the development of various executable compressors released in the late 1980s and 1990s. The following decoder works with a compressor by Sebastian Steinhauer.

```
// to keep track of flags
typedef struct _lzss_ctx_t {
    uint8_t w;
    uint8_t *in;
} lzss_ctx;

// read a bit
uint8_t get_bit(lzss_ctx *c) {
    uint8_t x;

    x = c->w;
    c->w <<= 1;

    if(c->w == 0) {
      x = *c->in++;
      c->w = (x << 1) | 1;
    }
    return x >> 7;
}

uint32_t lzss_depack(
  void *outbuf,
  uint32_t outlen,
  const void *inbuf)
{
    uint8_t  *out, *end, *ptr;
    uint32_t i, ofs, len;
    lzss_ctx c;

    // initialize pointers
    out = (uint8_t*)outbuf;
    end = out + outlen;

    // initialize context
    c.in = (uint8_t*)inbuf;
    c.w  = 128;

    while(out < end) {
      // if bit is not set
      if(!get_bit(&c)) {
        // store literal
        *out++ = *c.in++;
      } else {
        // decode offset and length
        ofs = *(uint16_t*)c.in;
        c.in += 2;
        len = (ofs & 15) + 3;
        ofs >>= 4;
        ptr = out - ofs - 1;
        // copy bytes
        while(len--) *out++ = *ptr++;
      }
    }
    // return length
```

```
    return (out - (uint8_t*)outbuf);
}
```

The assembly is a straight forward translation of the C code, currently at 69 bytes.

```
lzss_depackx:
_lzss_depackx:
    pushad

    lea    esi, [esp+32+4]
    lodsd
    xchg   edi, eax          ; edi = outbuf
    lodsd
    lea    ebx, [edi+eax]    ; ebx = edi + outlen
    lodsd
    xchg   esi, eax          ; esi = inbuf
    mov    al, 128           ; set flags
lzss_main:
    cmp    edi, ebx          ; while(out < end)
    jnb    lzss_exit

    add    al, al            ; c->w <<= 1
    jnz    lzss_check_bit

    lodsb                    ; c->w = *c->in++;
    adc    al, al
lzss_check_bit:
    jc     read_pair         ; if bit set, read len,offset

    movsb                    ; *out++ = *c.in++;
    jmp    lzss_main
read_pair:
    movzx  edx, word[esi]    ; ofs = *(uint16_t*)c.in;
    add    esi, 2            ; c.in += 2;
    mov    ecx, edx          ; len = (ofs % LEN_SIZE) + LEN_MIN;
    and    ecx, 15           ;
    add    ecx, 3            ;
    shr    edx, 4            ; ofs >>= 4
    push   esi
    lea    esi, [edi-1]      ; ptr = out - ofs - 1;
    sub    esi, edx          ;
    rep    movsb             ; while(len--) *out++ = *ptr++;
    pop    esi
    jmp    lzss_main
lzss_exit:
    ; return (out - (uint8_t*)outbuf);
    sub    edi, [esp+32+4]
    mov    [esp+28], edi
    popad
    ret
```

7. Lempel-Ziv-Bell (LZB)

Designed by Tim Bell and described in his 1986 Ph.D. dissertation A Unifying Theory and Improvements for Existing Approaches to Text Compression. It uses a pre-processor based on LZSS and Elias gamma coding of the match length, which results in a compression ratio similar to LZH and LZARI by Okumura. However, it does not suffer the performance penalty of using Huffman or arithmetic coding. Introspec considers it to be the first implementation that uses variable-length coding for reference matches, which is the basis for most modern LZ77-style compressors.

A key exhibit in a $300 million lawsuit brought by Stac Electronics (SE) against Microsoft was Bell's thesis. The 1993 case centered around a disk compression utility included with MS-DOS 6.0 called DoubleSpace. SE accused Microsoft of patent violations by using the same compression technologies used in its Stacker product. The courts agreed, and SE were awarded $120 million in compensatory damages.

8. Intel 8088 / 8086

For many years, bigger nerds than myself would remind me of what a mediocre architecture the x86 is and that it didn't deserve to be the most popular CPU for personal computers. But if it's so bad, how did it become the predominant architecture? It probably commenced in the 1970s with the release of the 8080, and an operating system designed for it by Gary Kildall called Control Program Monitor or Control Program for Microcomputers (CP/M).

Year	Model	Data Width (bits)	Address Width (bits)
1971	4004	4	12
1972	8008	8	14
1974	4040	4	12
1974	8080	8	16
1976	8085	8	16
1978	8086	16	20
1979	8088	8	20

Kildall initially designed and developed CP/M for the 8-Bit 8080 and licensed it to run devices such as the IMSAI 8080 (seen in the movie Wargames). Kildall was motivated by the enormous potential for microcomputers to become regular home appliances. And when IBM wanted to build a microcomputer of its own in 1980, CP/M was the most successful operating system on the market.

IBM made two decisions: use the existing software and hardware for the 8085-based IBM System/23 by using the 8088 instead of the 8086. (the cost per CPU unit was also a factor); and use its product to run CP/M to remain competitive with other microcomputers on the market.

Regrettably, Kildall missed a unique opportunity to supply CP/M for the IBM Personal Computer. Instead, Bill Gates / Microsoft obtained licensing to use a cloned version of CP/M called the Quick and Dirty Operating System (QDOS). QDOS was later rebranded to 86-DOS, before being shipped with the first IBM PC as "IBM PC DOS". Microsoft later purchased 86-DOS, rebranded it Microsoft Disk Operating System (MS-DOS), and forced IBM into a licensing agreement so Microsoft were free to sell MS-DOS to other companies. Kildall would later remark in his unpublished memoir Computer Connections, People, Places, and Events in the Evolution of the Personal Computer Industry. that **"Gates is more an opportunist than a technical type and severely opinionated even when the opinion he holds is absurd."**

8.1 LZE

Designed by Fabrice Bellard in 1989 and included in the closed-source MS-DOS packer LZEXE by the same. Inspired by LZSS but provides a higher compression ratio. Hiroaki Goto reverse engineered this in 1995 and published an open-source implementation in 2008. The following is a 32-Bit translation of the 16-Bit decoder with some additional optimizations. There's also a 68K version for anyone interested and a Z80 version by Kei Moroboshi published in 2017.

```
lze_depack:
_lze_depack:
    pushad
    mov    edi, [esp+32+4] ; edi = out
    mov    esi, [esp+32+8] ; esi = in

    call   init_get_bit
lze_get_bit:
    add    dl, dl           ;
    jnz    exit_get_bit

    mov    dl, [esi]        ; dl = *src++;
    inc    esi
    rcl    dl, 1
exit_get_bit:
    ret
init_get_bit:
    pop    ebp
    mov    dl, 128
lze_cl:
    movsb
lze_main:
    call   ebp              ; if(get_bit()) continue;
    jc     lze_cl

    call   ebp              ; if(get_bit()) {
    jc     lze_copy3

    xor    ecx, ecx         ; len = 0

    call   ebp              ; get_bit()
    adc    ecx, ecx

    call   ebp              ; get_bit()
    adc    ecx, ecx

    lodsb                   ; a.b[0] = *in++;
    mov    ah, -1           ; a.b[1] = 0xFF;
lze_copy1:
    inc    ecx              ; len++;
    jmp    lze_copy2
lze_copy3:                  ; else
    lodsw
    xchg   al, ah
    mov    ecx, eax
    shr    eax, 3           ; ofs /= 8
    or     ah, 0e0h
    and    ecx, 7           ; len %= 8
    jnz    lze_copy1
    mov    cl, [esi]        ; len = *src++;
    inc    esi
    ; EOF?
    jecxz  lze_exit         ; if(len == 0) break;
lze_copy2:
    movsx  eax, ax
```

```
    push   esi
    lea    esi, [edi+eax]
    inc    ecx
    rep    movsb
    pop    esi
    jmp    lze_main
    ; return (out - (uint8_t*)outbuf);
lze_exit:
    sub    edi, [esp+32+4]
    mov    [esp+28], edi
    popad
    ret
```

8.2 LZ4

Designed by Yann Collet and published in 2011. LZ4 is fast for both compression and decompression with a small decoder. Speed is somewhere between DEFLATE and LZO, while the compression ratio is similar to LZO but worse than DEFLATE. Despite the compression ratio being worse than DEFLATE, LZ4 doesn't require a Huffman or arithmetic/range decoder. The following 32-Bit code is a conversion of the 8088/8086 implementation by Trixter. Jørgen Ibsen has implemented LZ4 with optimal parsing using BriefLZ algorithms.

```
lz4_depack:
_lz4_depack:
    pushad
    lea     esi,[esp+32+4]
    lodsd                   ;load target buffer
    xchg    eax,edi
    lodsd
    xchg    eax,ebx         ;BX = chunk length minus header
    lodsd                   ;load source buffer
    xchg    eax,esi
    add     ebx,esi         ;BX = threshold to stop decompression
    xor     ecx,ecx
@@parsetoken:               ;CX=0 here because of REP at end of loop
    mul     ecx
    lodsb                   ;grab token to AL
    mov     dl,al           ;preserve packed token in DX
@@copyliterals:
    shr     al,4            ;unpack upper 4 bits
    call    buildfullcount  ;build full literal count if necessary
@@doliteralcopy:            ;src and dst might overlap so do this by bytes
    rep     movsb           ;if cx=0 nothing happens
;At this point, we might be done; all LZ4 data ends with five literals and the
;offset token is ignored.  If we're at the end of our compressed chunk, stop.
    cmp     esi,ebx         ;are we at the end of our compressed chunk?
    jae     done            ;if so, jump to exit; otherwise, process match
@@copymatches:
    lodsw                   ;AX = match offset
    xchg    edx,eax         ;AX = packed token, DX = match offset
    and     al,0Fh          ;unpack match length token
    call    buildfullcount  ;build full match count if necessary
@@domatchcopy:
    push    esi             ;ds:si saved, xchg with ax would destroy ah
    mov     esi,edi
    sub     esi,edx
    add     ecx,4           ;minmatch = 4
                            ;Can't use MOVSWx2 because [es:di+1] is unknown
    rep     movsb           ;copy match run if any left
    pop     esi
    jmp     @@parsetoken
buildfullcount:
                            ;CH has to be 0 here to ensure AH remains 0
    cmp     al,0Fh          ;test if unpacked literal length token is 15?
    xchg    ecx,eax         ;CX = unpacked literal length token; flags unchanged
    jne     builddone       ;if AL was not 15, we have nothing to build
buildloop:
    lodsb                   ;load a byte
    add     ecx,eax         ;add it to the full count
    cmp     al,0FFh         ;was it FF?
    je      buildloop       ;if so, keep going
builddone:
    ret
done:
    sub     edi,[esp+32+4];subtract original offset from where we are now
    mov     [esp+28], edi
```

```
    popad
    ret
```

8.3 LZSA

Designed by Emmanuel Marty with participation from Introspec and published in 2018. Introspec explains the difference between the two formats, LZSA1 and LZSA2.

LZSA1 is designed to directly compete with LZ4. If you compress using "lzsa -f1 -r INPUT OUTPUT", you are very likely to get higher compression ratio than LZ4 and probably slightly lower decompression speed compared to LZ4 (I am comparing speeds of LZSA1 fast decompressor and LZ4 fast decompressor, both hand-tuned by myself). If you really want to compete with LZ4 on speed, you need to compress using one of the "boost" options "lzsa -f1 -r -m4 INPUT OUTPUT" (better ratio, similar speed to LZ4) or "lzsa -f1 -r -m5 INPUT OUTPUT" (similar ratio, faster decompression than LZ4).

LZSA2 is approximately in the same league as BitBuster or ZX7. It's likely to be worse if you're compressing pure graphics (at least this is what we are seeing on ZX Spectrum), but it has much larger window and is pretty decent at compressing mixed data (e.g. a complete game binary or something similar). We accepted that the compression ratio is not the best because we wanted to preserve some of its speed. You should expect LZSA2 to decompress data about 50% faster than best I can do for ZX7. I did not do tests on BitBuster, but I just had a look at decompressor for ver.1.2 and there is no way it can compete with LZSA2 on speed.

```
lzsa1_decompress:
_lzsa1_decompress:
    pushad

    mov    edi, [esp+32+4]    ; edi = outbuf
    mov    esi, [esp+32+8]    ; esi = inbuf

    xor    ecx, ecx
.decode_token:
    mul    ecx
    lodsb                     ; read token byte: O|LLL|MMMM
    mov    dl, al             ; keep token in dl

    and    al, 070H           ; isolate literals length in token (LLL)
    shr    al, 4              ; shift literals length into place

    cmp    al, 07H            ; LITERALS_RUN_LEN?
    jne    .got_literals      ; no, we have the full literals count from the token,
go copy

    lodsb                     ; grab extra length byte
    add    al, 07H            ; add LITERALS_RUN_LEN
    jnc    .got_literals      ; if no overflow, we have the full literals count, go
copy
    jne    .mid_literals

    lodsw                     ; grab 16-bit extra length
    jmp    .got_literals

.mid_literals:
    lodsb                     ; grab single extra length byte
    inc    ah                 ; add 256

.got_literals:
    xchg   ecx, eax
    rep    movsb              ; copy cx literals from ds:si to es:di

    test   dl, dl             ; check match offset size in token (O bit)
    js     .get_long_offset

    dec     ecx
    xchg    eax, ecx          ; clear ah - cx is zero from the rep movsb above
    lodsb
    jmp     .get_match_length

.get_long_offset:
    lodsw                     ; Get 2-byte match offset

.get_match_length:
    xchg    eax, edx          ; edx: match offset  eax: original token
    and     al, 0FH           ; isolate match length in token (MMMM)
    add     al, 3             ; add MIN_MATCH_SIZE

    cmp     al, 012H          ; MATCH_RUN_LEN?
    jne     .got_matchlen     ; no, we have the full match length from the token, go
```

```
copy

    lodsb                     ; grab extra length byte
    add     al,012H           ; add MIN_MATCH_SIZE + MATCH_RUN_LEN
    jnc     .got_matchlen     ; if no overflow, we have the entire length
    jne     .mid_matchlen

    lodsw                     ; grab 16-bit length
    test    eax, eax          ; bail if we hit EOD
    je      .done_decompressing
    jmp     .got_matchlen

.mid_matchlen:
    lodsb                     ; grab single extra length byte
    inc     ah                ; add 256

.got_matchlen:
    xchg    ecx, eax          ; copy match length into ecx
    xchg    esi, eax
    mov     esi, edi          ; esi now points at back reference in output data
    movsx   edx, dx           ; sign-extend dx to 32-bits.
    add     esi, edx
    rep     movsb             ; copy match
    xchg    esi, eax          ; restore esi
    jmp     .decode_token     ; go decode another token

.done_decompressing:
    sub    edi, [esp+32+4]
    mov    [esp+28], edi      ; eax = decompressed size
    popad
    ret                       ; done
```

8.4 aPLib

Designed by Jørgen Ibsen and published in 1998, it continues to remain a closed-source compressor. Fortunately, an open-source version of the compressor called aPUltra is available, which was released by Emmanuel Marty in 2019. The small compressor in x86 assembly follows.

```
apl_decompress:
_apl_decompress:
    pushad

    %ifdef CDECL
      mov    esi, [esp+32+4]  ; esi = aPLib compressed data
      mov    edi, [esp+32+8]  ; edi = output
    %endif

    ; === register map ===
    ;  al: bit queue
    ;  ah: unused, but value is trashed
    ; ebx: follows_literal
    ; ecx: scratch register for reading gamma2 codes and storing copy length
    ; edx: match offset (and rep-offset)
    ; esi: input (compressed data) pointer
    ; edi: output (decompressed data) pointer
    ; ebp: offset of .get_bit

    mov     al,080H         ; clear bit queue(al) and set high bit to move into carry
    xor     edx, edx        ; invalidate rep offset in edx

    call    .init_get_bit
.get_dibits:
    call    ebp             ; read data bit
    adc     ecx,ecx         ; shift into cx
.get_bit:
    add     al,al           ; shift bit queue, and high bit into carry
    jnz     .got_bit        ; queue not empty, bits remain
    lodsb                   ; read 8 new bits
    adc     al,al           ; shift bit queue, and high bit into carry
.got_bit:
    ret
.init_get_bit:
    pop     ebp             ; load offset of .get_bit, to be used with call ebp
    add     ebp, .get_bit - .get_dibits
.literal:
    movsb                   ; read and write literal byte
.next_command_after_literal:
    push    03H
    pop     ebx             ; set follows_literal(bx) to 3

.next_command:
    call    ebp             ; read 'literal or match' bit
    jnc     .literal        ; if 0: literal

                            ; 1x: match
    call    ebp             ; read '8+n bits or other type' bit
    jc      .other          ; 11x: other type of match
                            ; 10: 8+n bits match
    call    .get_gamma2     ; read gamma2-coded high offset bits
    sub     ecx,ebx         ; high offset bits == 2 when follows_literal == 3 ?
                            ; (a gamma2 value is always >= 2, so substracting
follows_literal when it
                            ; is == 2 will never result in a negative value)
```

```
    jae     .not_repmatch   ; if not, not a rep-match
    call    .get_gamma2     ; read match length
    jmp     .got_len        ; go copy
.not_repmatch:
    mov     edx,ecx         ; transfer high offset bits to dh
    shl     edx,8
    mov     dl,[esi]        ; read low offset byte in dl
    inc     esi
    call    .get_gamma2     ; read match length
    cmp     edx,7D00H       ; offset >= 32000 ?
    jae     .increase_len_by2 ; if so, increase match len by 2
    cmp     edx,0500H       ; offset >= 1280 ?
    jae     .increase_len_by1 ; if so, increase match len by 1
    cmp     edx,0080H       ; offset < 128 ?
    jae     .got_len        ; if so, increase match len by 2, otherwise it would be a
7+1 copy
.increase_len_by2:
    inc     ecx             ; increase length
.increase_len_by1:
    inc     ecx             ; increase length
    ; copy ecx bytes from match offset edx
.got_len:
    push    esi             ; save esi (current pointer to compressed data)
    mov     esi,edi         ; point to destination in edi - offset in edx
    sub     esi,edx
    rep     movsb           ; copy matched bytes
    pop     esi             ; restore esi
    mov     bl,02H          ; set follows_literal to 2 (ebx is unmodified by match
commands)
    jmp     .next_command
    ; read gamma2-coded value into ecx
.get_gamma2:
    xor     ecx,ecx         ; initialize to 1 so that value will start at 2
    inc     ecx             ; when shifted left in the adc below
.gamma2_loop:
    call    .get_dibits     ; read data bit, shift into cx, read continuation bit
    jc      .gamma2_loop    ; loop until a zero continuation bit is read
    ret
    ; handle 7 bits offset + 1 bit len or 4 bits offset / 1 byte copy
.other:
    xor     ecx,ecx
    call    ebp             ; read '7+1 match or short literal' bit
    jc      .short_literal  ; 111: 4 bit offset for 1-byte copy
                            ; 110: 7 bits offset + 1 bit length

    movzx   edx,byte[esi]   ; read offset + length in dl
    inc     esi
    inc     ecx             ; prepare cx for length below
    shr     dl,1            ; shift len bit into carry, and offset in place
    je      .done           ; if zero offset: EOD
    adc     ecx,ecx         ; len in cx: 1*2 + carry bit = 2 or 3
    jmp     .got_len
    ; 4 bits offset / 1 byte copy
.short_literal:
    call    .get_dibits     ; read 2 offset bits
```

```
    adc     ecx,ecx
    call    .get_dibits     ; read 2 offset bits
    adc     ecx,ecx
    xchg    eax,ecx         ; preserve bit queue in cx, put offset in ax
    jz      .write_zero     ; if offset is 0, write a zero byte
                            ; short offset 1-15
    mov     ebx,edi         ; point to destination in es:di - offset in ax
    sub     ebx,eax         ; we trash bx, it will be reset to 3 when we loop
    mov     al,[ebx]        ; read byte from short offset
.write_zero:
    stosb                   ; copy matched byte
    xchg    eax,ecx         ; restore bit queue in al
    jmp     .next_command_after_literal
.done:
    sub     edi, [esp+32+8] ; compute decompressed size
    mov     [esp+28], edi
    popad
    ret
```

9. MOS Technology 6502

This 8-Bit CPU was the product of Motorola management, ignoring customer concerns about the cost of the 6800 CPU launched by the company in 1974. Following consultations with potential customers for the 6800. Chuck Peddle tried to convince Motorola to develop a low-cost alternative for consumers on a limited budget.

Motorola ordered Peddle to cease working on this idea, which resulted in his departure from the company with several other employees that began working on the 6502 at MOS Technology. Used in the Commodore 64, the Apple II, and the BBC Micro home computers, including various gaming consoles, Motorola acknowledged missing a golden opportunity. The company would later express regret for dismissing Peddle's idea since the 6502 was far more successful than the 6800.

Trivia: The Terminator movie from 1984 uses CPU instructions from the 6502.

Those of you that want to program a Commodore 64 without purchasing one can always use an emulator like VICE. For the Apple II, there's AppleWin. (Yes, Windows only). Since Qkumba already implemented several popular depackers for 6502, I requested a translation of the Exomizer compression algorithm. Using this translation, I created the following table, which lists 6502 instructions and their equivalent for x86. The EBX and ECX registers replace the X and Y registers, respectively. Using #$80 as an immediate value is simply for demonstration, and you'll find a full list of instructions here.

6502	x86	Description
lda #$80	mov al, 0x80	Load byte into accumulator.

sta [address]	mov [address], al	Store accumulator in memory.
cmp #$80	cmp al, 0x80	Compare byte with accumulator.
cpx #$80	cmp bl, 0x80	Compare byte with X.
cpy #$80	cmp cl, 0x80	Compare byte with Y.
asl	shl al, 1	ASL shifts all bits left one position. 0 is shifted into bit 0 and the original bit 7 is shifted into the Carry.
lsr	shr al, 1	Logical shift right.
bit #$7	test al, 7	Perform a bitwise AND, set the flags and discard the result.
sec	stc	SEt the Carry flag.
adc #$80	adc al, 0x80	Add byte with Carry.
sbc #$1	sbb al, 1	Subtract byte with Carry.
rts	ret	Return from subroutine.
jsr	call	Save next address and jump to subroutine.
eor #$80	xor al, 0x80	Perform an exclusive OR.
ora #$80	or al, 0x80	Perform a bitwise OR.
and #$80	and al, 0x80	Bitwise AND with accumulator
rol	rcl al, 1	Shifts all bits left one position. The Carry is shifted into bit 0 and the original bit 7 is shifted into the Carry.
ror	rcr al, 1	Shifts all bits right one position. The Carry is shifted into bit 7 and the original bit 0 is shifted into the Carry.
bpl	jns	Branch on PLus. Jump if Not Signed.
bmi	js	Branch on MInus. Jump if Signed.
bcc:bcs	jnc:jc	Branch on Carry Clear. Branch on Carry Set.
bne:beq	jne:je	Branch on Not Equal. Branch on EQual.
bvc:bvs	jno:jo	Branch on oVerflow Clear. Branch on oVerflow Set.
php	pushf	PusH Processor status.

plp	popf	PuLl Processor status.
pha	push eax	PusH Accumulator.
pla	pop eax	PuLl Accumulator.
tax	movzx ebx, al / mov bl, al	Transfer A to X.
tay	movzx ecx, al / mov cl, al	Transfer A to Y.
txa	mov al, bl	Transfer X to A.
tya	mov al, cl	Transfer Y to A.
inx	inc ebx / inc bl	INcrement X.
iny	inc ecx / inc cl	INcrement Y.
dex	dec ebx / dec bl	DEcrement X.
dey	dec ecx / dec cl	DEcrement Y.

9.1 Exomizer

Designed by Magnus Lind and published in 2002. Exomizer is popular for devices such as the Commodore VIC20, the C64, the C16/plus4, the C128, the PET 4032, the Atari 400/800 XL/XE, the Apple II+e, the Oric-1, the Oric Atmos, and the BBC Micro B. It inspired the development of other executable compressors, most notably PackFire. Qkumba was kind enough to provide a translation of the Exomizer 3 decoder translated from 6502 to x86. However, due to the complexity of the source code, only a snippet of code is shown here. The Y register maps to the EDI register while the X register maps to the ESI register.

```
%MACRO mac_get_bits 0
        call get_bits                   ;jsr get_bits
%ENDM
get_bits:
        adc  al, 0x80                   ;adc #$80                ; needs c=0, affects
v
        pushfd
        shl  al, 1                      ;asl
        lahf
        jns  gb_skip                    ;bpl gb_skip
gb_next:
        shl  byte [zp_bitbuf], 1        ;asl zp_bitbuf
        jne  gb_ok                      ;bne gb_ok
        mac_refill_bits                 ;+mac_refill_bits
gb_ok:
        rcl  al, 1                      ;rol
        lahf
        test al, al
        js   gb_next                    ;bmi gb_next
gb_skip:
        popfd
        sahf
        jo   gb_get_hi                  ;bvs gb_get_hi
        ret                             ;rts
gb_get_hi:
        stc                             ;sec
        mov  [zp_bits_hi], al           ;sta zp_bits_hi
        jmp  get_crunched_byte          ;jmp get_crunched_byte
%ENDIF
; ------------------------------------------------------------------
; calculate tables (62 bytes) + get_bits macro
; x and y must be #0 when entering
;
        clc                             ;clc
table_gen:
        movzx esi, al                   ;tax
        mov   eax, edi                  ;tya
        and   al, 0x0f                  ;and #$0f
        mov   [edi + tabl_lo], al       ;sta tabl_lo,y
        je    shortcut                  ;beq shortcut            ; start a new
sequence
; ------------------------------------------------------------------
        mov   eax, esi                  ;txa
        adc   al, [edi + tabl_lo - 1]   ;adc tabl_lo - 1,y
        mov   [edi + tabl_lo], al       ;sta tabl_lo,y
        mov   al, [zp_len_hi]           ;lda zp_len_hi
        adc   al, [edi + tabl_hi - 1]   ;adc tabl_hi - 1,y
shortcut:
        mov   [edi + tabl_hi], al       ;sta tabl_hi,y
; ------------------------------------------------------------------
        mov   al, 0x01                  ;lda #$01
        mov   [zp_len_hi], al           ;sta <zp_len_hi
        mov   al, 0x78                  ;lda #$78                ; %01111000
        mac_get_bits                    ;+mac_get_bits
; ------------------------------------------------------------------
```

```
        shr   al, 1                    ;lsr
        movzx esi, al                  ;tax
        je    rolled                   ;beq rolled
        pushfd                         ;php
rolle:
        shl  byte [zp_len_hi],1        ;asl zp_len_hi
        stc                            ;sec
        rcr  al, 1                     ;ror
        dec  esi                       ;dex
        jne  rolle                     ;bne rolle
        popfd                          ;plp
rolled:
        rcr  al, 1                     ;ror
        mov  [edi + tabl_bi], al       ;sta tabl_bi,y
        test al, al
        js   no_fixup_lohi             ;bmi no_fixup_lohi
        mov  al, [zp_len_hi]           ;lda zp_len_hi
        mov  ebx, esi
        mov  [zp_len_hi], bl           ;stx zp_len_hi
        jmp  skip_fix                  ;!BYTE $24
no_fixup_lohi:
        mov  eax, esi                  ;txa
; ------------------------------------------------------------------
skip_fix:
        inc  edi                       ;iny
        cmp  edi, encoded_entries      ;cpy #encoded_entries
        jne  table_gen                 ;bne table_gen
```

9.2 Pucrunch

Designed by Pasi Ojala and published in 1997. It's described by the author as a Hybrid LZ77 and RLE compressor, using Elias gamma coding for reference length, and a mixture of gamma and linear code for the offset. It requires no additional memory for decompression. The description and source code are well worth a read for those of you that want to understand the characteristics of other LZ77-style compressors.

10. Zilog 80

I was able to design whatever I wanted. And personally I wanted to develop the best and the most wonderful 8-Bit microprocessor in the world. — Masatoshi Shima

After helping to design microprocessors at Intel (4-Bit 4004, the 8-Bit 8008 and 8080), Ralph Ungermann and Federico Faggin left Intel in 1974 to form Zilog. Masatoshi Shima, who also worked at Intel, would later join the company in 1975 to work on an 8-Bit CPU released in 1976 they called the Z80. The Z80 is essentially a clone of the Intel 8080 with support for more instructions, more registers, and 16-Bit capabilities. Many of the Z80 instructions, to the best of my knowledge, do not have an equivalent on the x86. Proceed with caution, as with no prior experience writing for the Z80, some of the mappings presented here may be incorrect.

Z80	x86	Z80 Description
bit	test	Perform a bitwise AND, set state flags and discard result.
ccf	cmc	Inverts/Complements the carry flag.
cp	cmp	Performs subtraction from A. Sets flags and discards result.
djnz	loop	Decreases B and jumps to a label if Not Zero. If mapping BC to CX, LOOP works or REP depending on operation.
ex	xchg	Exchanges two 16-bit values.
exx		EXX exchanges BC, DE, and HL with shadow registers with BC’, DE’, and HL’. Unfortunately, nothing like this available for x86. Try to use spare registers or rewrite algorithm to avoid using EXX.
jp	jcc	Conditional or unconditional jump to absolute address.
jr	jcc	Conditional or unconditional jump to relative address not exceeding 128-bytes ahead or behind.
ld	mov	Load/Copy immediate value or register to another register.
ldi	movsb	Performs a “LD (DE),(HL)”, then increments DE and HL. Map SI to HL, DI to DE and you can perform the same operation quite easily on x86.
ldir	rep movsb	Repeats LDI (LD (DE),(HL), then increments DE, HL, and decrements BC) until BC=0. Note that if BC=0 before this instruction is called, it will loop around until BC=0 again.
res	btr	Reset bit. BTR doesn’t behave exactly the same, but it’s close enough. An alternative might be masking with AND.
rl / rla / rlc / rlca	rcl or adc	The register is shifted left and the carry flag is put into bit zero of the register. The 7th bit is put into the carry flag. You can perform the same operation using ADC (Add with Carry).
rld		Performs a 4-bit leftward rotation of the 12-bit number whose 4 most signigifcant bits are the 4 least significant bits of A, and its 8 least significant bits are in (HL).
rr / rra / r	rcr	9-bit rotation to the right. The carry is copied into bit 7, and the bit leaving on the right is copied into the carry.
rra		Performs a RR A faster, and modifies the flags differently.
sbc	sbb	Sum of second operand and carry flag is subtracted from the first operand. Results are written into the first operand.
sla	sal	

sll/sl1	shl	An “undocumented” instruction. Functions like sla, except a 1 is inserted into the low bit.
sra	sar	Arithmetic shift right 1 bit, bit 0 goes to carry flag, bit 7 remains unchanged.
srl	shr	Like SRA, except a 0 is put into bit 7. The bits are all shifted right, with bit 0 put into the carry flag.

10.1 Mega LZ

Designed by the demo group MAYhEM and published in 2005. The original Z80 decoder by fyrex was optimized by Introspec in 2017 while researching 8-Bit compression algorithms. The x86 assembly based on that uses the following register mapping.

Register Mapping	
Z80	x86
A	AL
B	EBX
C	ECX
D	DH
E	DL
HL	ESI
DE	EDI

The EBX and ECX registers are to replace the B and C registers, respectively, to save a few bytes required for incrementing and decrementing 8-bit registers on x86.

```
megalz_depack:
_megalz_depack:
    pushad

    mov    esi, [esp+32+12]  ; esi = inbuf
    mov    edi, [esp+32+ 4]  ; edi = outbuf

    call   init_get_bit

    add    al, al            ; add a, a
    jnz    exit_get_bit      ; ret nz
    lodsb                    ; ld a, (hl)
                             ; inc hl
    adc    al, al            ; rla
exit_get_bit:
    ret                      ; ret
init_get_bit:
    pop    ebp               ;
    mov    al, 128           ; ld a, 128
mlz_literal:
    movsb                    ; ldi
mlz_main:
    call   ebp               ; GET_BIT
    jc     mlz_literal       ; jr c, mlz_literal
    xor    edx, edx
    mov    dh, -1            ; ld d, #FF
    xor    ebx, ebx          ; ld bc, 2
    push   2
    pop    ecx
    call   ebp               ; GET_BIT
    jc     CASE01x           ; jr c, CASE01x
    call   ebp               ; GET_BIT
    jc     mlz_short_ofs     ; jr c, mlz_short_ofs
CASE000:
    dec    ecx               ; dec c
    mov    dl, 63            ; ld e, %00111111
ReadThreeBits:
    call   ebp               ; GET_BIT
    adc    dl, dl            ; rl e
    jnc    ReadThreeBits     ; jr nc, ReadThreeBits
mlz_copy_bytes:
    push   esi               ; push hl
    movsx  edx, dx           ; sign-extend dx to 32-bits
    lea    esi, [edi+edx]    ;
    rep    movsb             ; ldir
    pop    esi               ; pop hl
    jmp    mlz_main          ; jr mlz_main
CASE01x:
    call   ebp               ; GET_BIT
    jnc    CASE010           ; jr nc, CASE010
    dec    ecx               ; dec c
ReadLogLength:
    call   ebp               ; GET_BIT
    inc    ebx               ; inc b
    jnc    ReadLogLength     ; jr nc, ReadLogLength
```

```
mlz_read_len:
    call   ebp                 ; GET_BIT
    adc    cl, cl              ; rl c
    jc     mlz_exit            ; jr c, mlz_exit
    dec    ebx                 ; djnz mlz_read_len
    jnz    mlz_read_len
    inc    ecx                 ; inc c
CASE010:
    inc    ecx                 ; inc c
    call   ebp                 ; GET_BIT
    jnc    mlz_short_ofs       ; jr nc, mlz_short_ofs
    mov    dh, 31              ; ld d, %00011111
mlz_long_ofs:
    call   ebp                 ; GET_BIT
    adc    dh, dh              ; rl d
    jnc    mlz_long_ofs        ; jr nc, mlz_long_ofs
    dec    edx                 ; dec d
mlz_short_ofs:
    mov    dl, [esi]           ; ld e, (hl)
    inc    esi                 ; inc hl
    jmp    mlz_copy_bytes      ; jr mlz_copy_bytes
mlz_exit:
    sub    edi, [esp+32+4]
    mov    [esp+28], edi       ; eax = decompressed length
    popad
    ret
```

10.2 ZX7

Designed by Einar Saukas and published in 2012. ZX7 is an optimal LZ77 algorithm for the ZX-Spectrum using a combination of fixed length and variable length Gamma codes for the match length and offset. The following is a translation of the standard Z80 depacker to a 32-bit x86 assembly in 111 bytes.

Register Mapping	
Z80	x86
A	AL
B	CH
C	CL
BC	CX
D	DH
E	DL
HL	ESI

DE	EDX or EDI

```
dzx7_standard:
_dzx7_standard:
    pushad

    ; tested on Windows
    mov    esi, [esp+32+12]     ; hl = source
    mov    edi, [esp+32+ 4]     ; de = destination

    mov    al, 0x80             ; ld      a, $80
dzx7s_copy_byte_loop:
    ; copy literal byte
    movsb                       ; ldi
dzx7s_main_loop:
    call   dzx7s_next_bit       ; call    dzx7s_next_bit
; next bit indicates either literal or sequence
    jnc    dzx7s_copy_byte_loop ; jr      nc, dzx7s_copy_byte_loop

; determine number of bits used for length (Elias gamma coding)
    push   edi                  ; push    de
    mov    ecx, 0               ; ld      bc, 0
    mov    dh, ch               ; ld      d, b
dzx7s_len_size_loop:
    inc    dh                   ; inc     d
    call   dzx7s_next_bit       ; call    dzx7s_next_bit
    jnc    dzx7s_len_size_loop  ; jr      nc, dzx7s_len_size_loop
; determine length
dzx7s_len_value_loop:
    jc     skip_call
    call   dzx7s_next_bit       ; call    nc, dzx7s_next_bit
skip_call:
    rcl    cl, 1                ; rl      c
    rcl    ch, 1                ; rl      b
    ; check end marker
    jc     dzx7s_exit           ; jr      c, dzx7s_exit
    dec    dh                   ; dec     d
    jnz    dzx7s_len_value_loop ; jr      nz, dzx7s_len_value_loop
    ; adjust length
    inc    cx                   ; inc     bc

; determine offset
    ; load offset flag (1 bit) + offset value (7 bits)
    mov    dl, [esi]            ; ld      e, (hl)
    inc    esi                  ; inc     hl
    ; opcode for undocumented instruction "SLL E" aka "SLS E"
    shl    dl, 1                ; defb    $cb, $33
    ; if offset flag is set, load 4 extra bits
    jnc    dzx7s_offset_end     ; jr      nc, dzx7s_offset_end
    ; bit marker to load 4 bits
    mov    dh, 0x10             ; ld      d, $10
dzx7s_rld_next_bit:
    call   dzx7s_next_bit       ; call    dzx7s_next_bit
    ; insert next bit into D
    rcl    dh, 1                ; rl      d
    ; repeat 4 times, until bit marker is out
    jnc    dzx7s_rld_next_bit   ; jr      nc, dzx7s_rld_next_bit
```

```
    ; add 128 to DE
    inc    dh                  ; inc     d
    ; retrieve fourth bit from D
    shr    dh, 1               ; srl   d
dzx7s_offset_end:
    ; insert fourth bit into E
    rcr    dl, 1               ; rr      e

; copy previous sequence
    ; store source, restore destination
    xchg   esi, [esp]          ; ex      (sp), hl
    ; store destination
    push   esi                 ; push    hl
    ; HL = destination - offset - 1
    sbb    esi, edx            ; sbc     hl, de
    ; DE = destination
    pop    edi                 ; pop     de
    rep    movsb               ; ldir
dzx7s_exit:
    pop    esi                 ; pop     hl
    jnc    dzx7s_main_loop     ; jr      nc, dzx7s_main_loop
    sub    edi, [esp+32+4]
    mov    [esp+28], edi
    popad
    ret
dzx7s_next_bit:
    ; check next bit
    add    al, al              ; add     a, a
    ; no more bits left?
    jnz    exit_get_bit        ; ret     nz
    ; load another group of 8 bits
    mov    al, [esi]           ; ld      a, (hl)
    inc    esi                 ; inc     hl
    rcl    al, 1               ; rla
exit_get_bit:
    ret                        ; ret
```

The following is a 32-Bit version of a size-optimized 16-bit code implemented by Trixter and Qkumba in 2016. It's currently 81 bytes.

```
zx7_depack:
_zx7_depack:
    pushad
    mov    edi, [esp+32+ 4] ; output
    mov    esi, [esp+32+12] ; input

    call   init_get_bit
    add    al, al           ; check next bit
    jnz    exit_get_bit     ; no more bits left?
    lodsb                   ; load another group of 8 bits
    adc    al, al
exit_get_bit:
    ret
init_get_bit:
    pop    ebp
    mov    al, 80h
    xor    ecx, ecx
copy_byte:
    movsb                    ; copy literal byte
main_loop:
    call   ebp
    jnc    copy_byte         ; next bit indicates either
                             ; literal or sequence
; determine number of bits used for length (Elias gamma coding)
    xor    ebx, ebx
len_size_loop:
    inc    ebx
    call   ebp
    jnc    len_size_loop
    jmp    len_value_skip
; determine length
len_value_loop:
    call   ebp
len_value_skip:
    adc    cx, cx
    jc     zx7_exit       ; check end marker

    dec    ebx
    jnz    len_value_loop

    inc    ecx            ; adjust length
                          ; determine offset
    mov    bl, [esi]      ; load offset flag (1 bit) +
                          ; offset value (7 bits)
    inc    esi
    stc
    adc    bl, bl
    jnc    offset_end     ; if offset flag is set, load
                          ; 4 extra bits
    mov    bh, 10h        ; bit marker to load 4 bits
rld_next_bit:
    call   ebp
    adc    bh, bh         ; insert next bit into D
    jnc    rld_next_bit   ; repeat 4 times, until bit
                          ; marker is out
```

```
    inc    bh              ; add 256 to DE
offset_end:
    shr    ebx, 1          ; insert fourth bit into E
    push   esi
    mov    esi, edi
    sbb    esi, ebx        ; destination = destination - offset - 1
    rep    movsb
    pop    esi             ; restore source address
    jmp    main_loop
zx7_exit:
    sub    edi, [esp+32+4]
    mov    [esp+28], edi
    popad
    ret
```

10.3 ZX7 Mini

Designed by Antonio Villena and published in 2019. This version uses less code at the expense of the compression ratio. Nevertheless, it's a great example to demonstrate the conversion between Z80 and x86.

Register Mapping

Z80	x86
A	AL
BC	ECX
D	DH
E	DL
HL	ESI
DE	EDI

```
zx7_depack:
_zx7_depack:
    pushad

    mov    esi, [esp+32+4] ; esi = in
    mov    edi, [esp+32+8] ; edi = out

    call   init_getbit
getbit:
    add    al, al          ; add     a, a
    jnz    exit_getbit     ; ret     nz
    lodsb                  ; ld      a, (hl)
                           ; inc     hl
    adc    al, al          ; adc     a, a
exit_getbit:
    ret
init_getbit:
    pop    ebp             ;
    mov    al, 80h         ; ld      a, $80
copyby:
    movsb                  ; ldi
mainlo:
    call   ebp             ; call    getbit
    jnc    copyby          ; jr      nc, copyby
    push   1               ; ld      bc, 1
    pop    ecx
lenval:
    call   ebp             ; call    getbit
    rcl    cl, 1           ; rl      c
    jc     exit_depack     ; ret     c
    call   ebp             ; call    getbit
    jnc    lenval          ; jr      nc, lenval
    push   esi             ; push    hl
    movzx  edx, byte[esi]  ; ld      l, (hl)
    mov    esi, edi
    sbb    esi, edx        ; sbc     hl, de
    rep    movsb           ; ldir
    pop    esi             ; pop     hl
    inc    esi             ; inc     hl
    jmp    mainlo          ; jr      mainlo
exit_depack:
    sub    edi, [esp+32+8] ;
    mov    [esp+28], edi
    popad
    ret
```

10.4 LZF

Designed by Ilya Muravyov and published here in 2013. The x86 assembly is a translation of a size-optimized version by introspec. The compressor is closed, so this is another example to demonstrate the conversion between Z80 and x86.

```
lzf_depack:
_lzf_depack:
    pushad
    mov    edi, [esp+32+4]   ; edi = outbuf
    mov    esi, [esp+32+8]   ; esi = inbuf

    xor    ecx, ecx          ; ld b,0
    jmp    MainLoop          ; jr MainLoop  ; all copying is done by LDIR; B needs to
be zero
ProcessMatches:
    push   eax               ; exa
    lodsb                    ; ld a,(hl)
                             ; inc hl
                             ; rlca
                             ; rlca
    rol    al, 3             ; rlca
    inc    al                ; inc a
    and    al, 00000111b     ; and %00000111
    jnz    CopyingMatch      ; jr nz,CopyingMatch
LongMatch:
    lodsb                    ; ld a,(hl)
    add    al, 8             ; add 8
                             ; inc hl ; len == 9 means an extra len byte needs to be
read
                             ; jr nc,CopyingMatch
                             ; inc b
    adc    ch, ch
CopyingMatch:
    mov    cl, al            ; ld c,a
    inc    ecx               ; inc bc
    pop    eax               ; exa
    cmp    al, 20h           ; token == #20 suggests a possibility of the end marker
(#20,#00)
    jnz    NotTheEnd         ; jr nz,NotTheEnd
    xor    al, al            ; xor a
    cmp    [esi], al         ; cp (hl)
    jz     exit              ; ret z   ; is it the end marker? return if it is
NotTheEnd:
    and    al, 1fh           ; and %00011111 ; A' = high(offset); also, reset flag C
for SBC below
    push   esi               ; push hl
    movzx  edx, byte[esi]    ; ld l,(hl)
    mov    dh, al            ; ld h,a                ; HL = offset
    movsx  edx, dx           ;
                             ; push de
    mov    esi, edi          ; ex de,hl              ; DE = offset, HL = dest
    sbb    esi, edx          ; sbc hl,de             ; HL = dest-offset
                             ; pop de
    rep    movsb             ; ldir
    pop    esi               ; pop hl
    inc    esi               ; inc hl
MainLoop:
    mov    al, [esi]         ; ld a,(hl)
    cmp    al, 20h           ; cp #20
    jnc    ProcessMatches    ; jr nc,ProcessMatches  ; tokens "000lllll" mean "copy
```

```
lllll+1 literals"
    inc    al                 ; inc a
    mov    cl, al             ; ld c,a
    inc    esi                ; inc hl
    rep    movsb              ; ldir   ; actual copying of the literals
    jmp    MainLoop           ; jr MainLoop
exit:
    sub    edi, [esp+32+4]
    mov    [esp+28], edi
    popad
    ret
```

11. Motorola 68000 (68K)

"Motorola, with its superior technology, lost the single most important design contest of the last 50 years" Walden C. Rhines

A revolutionary CPU released in 1979 that includes eight 32-Bit general-purpose data registers (D0-D7), and eight address registers (A0-A7) used for function arguments and stack pointer. The 68K was used in the Commodore Amiga, the Atari ST, the Macintosh, including various fourth-generation gaming consoles like the Sega Megadrive, and arcade systems like Namco System 2. The 68K was more compelling than the Z80, 6502, 8088, and 8086, so why did it lose to Intel in the home computer war of the 1980s? A history of the Amiga, part 10: The downfall of Commodore offers some plausible answers. IBM choosing Control Program/Monitor by Gary Kildall for its 1980 PC operating system is also likely a factor.

The following table lists some 68K instructions and the x86 instructions used to replace them.

68K	x86	Description
move	mov	Copy data from source to destination
add	add	Add binary.
addx	adc	Add with borrow/carry.
sub	sub	Subtract binary.
subx	sbb	Subtract with borrow/carry.
rts	ret	Return from subroutine.
dbf/dbt	loopne/loope	Test condition, decrement, and branch.
bsr	call	Branch to subroutine
bcs:bcc	jc:jnc	Branch/Jump if carry set. Jump if carry clear.

beq:bne	je:jne	Branch/Jump if equal. Not equal.
ble	jle	Branch/Jump if less than or equal.
bra	jmp	Branch always.
lsr	shr	Logical shift right.
lsl	shl	Logical shift left.
bhs	jae	Branch on higher than or same.
bpl	jns	Branch on higher than or same.
bmi	js	Branch on minus. Jump if signed.
tst	test	Test bit zero of a register.
exg	xchg	Exchange registers.

11.1 PackFire

Designed by neural and published in 2010, PackFire comprises two algorithms tailored for demos targeting the Atari ST. The first borrows ideas from Exomizer and is suitable for small files not exceeding ~40KB. The other borrows ideas from LZMA, which is more suited to compressing larger files. The LZMA-variant requires 16KB of RAM for the range decoder, which isn't a problem for the Atari ST with between 512-1024KB of RAM available. However, translating code written for the 68K to x86 isn't easy because the x86 is a less advanced architecture. Since being released, badcode has published decoders for a variety of other architectures, including 32-Bit ARM. The following is the Exomizer-style decoder for files not exceeding ~40KB, which probably isn't very useful unless you write demos for retro hardware.

```
packfire_depack:
_packfire_depack:
    pushad

    mov    ebp, [esp+32+4]   ; eax = inbuf (a0)
    mov    edi, [esp+32+8]   ; edi = outbuf (a1)

    lea    esi, [ebp+26]     ; lea     26(a0),a2
    lodsb                    ; move.b  (a2)+,d7
lit_copy:
    movsb                    ; move.b  (a2)+,(a1)+
main_loop:
    call   get_bit           ; bsr.b   get_bit
    jc     lit_copy          ; bcs.b   lit_copy

    cdq                      ; moveq   #-1,d3
    dec    edx
get_index:
    inc    edx               ; addq.l  #1,d3
    call   get_bit           ; bsr.b   get_bit
    jnc    get_index         ; bcc.b   get_index

    cmp    edx, 0x10         ; cmp.w   #$10,d3
    je     depack_stop       ; beq.b   depack_stop

    call   get_pair          ; bsr.b   get_pair
    push   edx               ; move.w  d3,d6 ; save it for the copy
    cmp    edx, 2            ; cmp.w   #2,d3
    jle    out_of_range      ; ble.b   out_of_range

    cdq                      ; moveq   #0,d3
out_of_range:
                             ; move.b  table_len(pc,d3.w),d1
                             ; move.b  table_dist(pc,d3.w),d0
    ; code without tables
    push   4                 ; d1 = 4
    pop    ecx
    push   16                ; d0 = 16
    pop    ebx
    dec    edx               ; d3--
    js     L0

    dec    edx
    mov    cl, 2             ; d1 = 2
    mov    bl, 48            ; d0 = 48
    js     L0

    mov    cl, 4             ; d1 = 4
    mov    bl, 32            ; d0 = 32
L0:
    call   get_bits          ; bsr.b   get_bits
    call   get_pair          ; bsr.b   get_pair
    pop    ecx
    push   esi
    mov    esi, edi          ; move.l  a1,a3
```

```
    sub     esi, edx           ; sub.l   d3,a3
copy_bytes:
    rep     movsb              ; move.b  (a3)+,(a1)+
                               ; subq.w  #1,d6
                               ; bne.b   copy_bytes
    pop     esi
    jmp     main_loop          ; bra.b   main_loop
get_pair:
    pushad
    cdq                        ; sub.l   a6,a6
                               ; moveq   #$f,d2
calc_len_dist:
    mov     ebx, edx           ; move.w  a6,d0
    and     ebx, 15            ; and.w   d2,d0
    jne     node               ; bne.b   node
    push    1
    pop     edi                ; moveq   #1,d5
node:
    mov     eax, edx           ; move.w  a6,d4
    shr     eax, 1             ; lsr.w   #1,d4
    mov     cl, [ebp+eax]      ; move.b  (a0,d4.w),d1
    push    1                  ; moveq   #1,d4
    pop     eax
    and     ebx, eax           ; and.w   d4,d0
    je      nibble             ; beq.b   nibble
    shr     ecx, 4             ; lsr.b   #4,d1
nibble:
    mov     ebx, edi           ; move.w  d5,d0
    and     ecx, 15            ; and.w   d2,d1
    shl     eax, cl            ; lsl.l   d1,d4
    add     edi, eax           ; add.l   d4,d5
    inc     edx                ; addq.w  #1,a6

    ; dbf  d3,calc_len_dist
    dec     dword[esp+pushad_t.edx]
    jns     calc_len_dist
    ; save d0 and d1
    mov     [esp+pushad_t.ebx], ebx
    mov     [esp+pushad_t.ecx], ecx
    popad
get_bits:
    cdq                        ; moveq   #0,d3
getting_bits:
    dec     ecx                ; subq.b  #1,d1
    jns     cont_get_bit       ; bhs.b   cont_get_bit
    add     edx, ebx           ; add.w   d0,d3
    ret
depack_stop:
    sub     edi, [esp+32+8]    ;
    mov     [esp+pushad_t.eax], edi
    popad
    ret                        ; rts
cont_get_bit:
    call    get_bit            ; bsr.b   get_bit
    adc     edx, edx           ; addx.l  d3,d3
```

```
    jmp    getting_bits      ; bra.b   getting_bits
get_bit:
    add    al, al            ; add.b   d7,d7
    jne    byte_done         ; bne.b   byte_done
    lodsb                    ; move.b  (a2)+,d7
    adc    al, al            ; addx.b  d7,d7
byte_done:
    ret                      ; rts
```

11.2 Shrinkler

Designed by Aske Simon Christensen (Blueberry/Loonies) and published in 1999. It stores compressed data in Big-Endian 32-bit words, and the x86 translation must use BSWAP before reading bits of the stream. The compressor is open source and could be updated to use Little-Endian format instead. Christensen is also a co-author of the Crinkler executable compressor along with Rune Stubbe (Mentor/TBC) that's popular for 4K intros on Windows.

The following is a description from Blueberry:

Shrinkler is optimized for target sizes around 4k (while still being good for 64k), which strongly favors decompression code size. It tries to achieve the best size for this target, somewhat at the expense of decompression speed. At the same time, it is intended to be useful on Amiga 500, which means that decompression speed should still be reasonable, and decompression memory usage should be small. Shrinkler decrunches a 64k intro in typically less than half a minute on Amiga 500, which is an acceptable wait time for starting an intro. And the memory needed for the probabilities fits within the default stack size of 4k on Amiga.

Shrinkler also has special tweaks gearing it towards 16-bit oriented data (as all 68000 instructions are a multiple of 16 bits). Specifically, it keeps separate literal context groups for even and odd bytes, since these distributions are usually very different for Amiga data. Same thing for the flag indicating whether the a literal or a match is coming up. This gives a great boost for Amiga intros, but it has no benefit for data that has arbitrary alignment. It usually doesn't hurt either, except for the slight cost in decompression code size.

The following is a translation of the 68K assembly to x86, with help from Blueberry.

```
    %define INIT_ONE_PROB       0x8000
    %define ADJUST_SHIFT        4
    %define SINGLE_BIT_CONTEXTS 1
    %define NUM_CONTEXTS        1536

    struc pushad_t
      .edi resd 1
      .esi resd 1
      .ebp resd 1
      .esp resd 1
      .ebx resd 1
      .edx resd 1
      .ecx resd 1
      .eax resd 1
    endstruc

    ; temporary variables for range decoder
    %define d2   4*0
    %define d3   4*1
    %define d4   4*2
    %define prob 4*3

    %ifndef BIN
      global ShrinklerDecompress
      global _ShrinklerDecompress
    %endif

ShrinklerDecompress:
_ShrinklerDecompress:
    ; save d2-d7/a4-a6 in -(a7) the stack
    pushad                   ; movem.l  d2-d7/a4-a6,-(a7)

    ; esi = inbuf
    mov    esi, [esp+32+4]   ; move.l a0,a4
    ; edi = outbuf
    mov    edi, [esp+32+8]   ; move.l a1,a5
                             ; move.l a1,a6
    ; allocate local memory for range decoder
    sub    esp, 4096
    test   [esp], esp        ; stack probe
    mov    ebp, esp          ; ebp = stack pointer

    ; Init range decoder state
    mov    dword[ebp+d2], 0  ; moveq.l  #0,d2
    mov    dword[ebp+d3], 1  ; moveq.l  #1,d3
    mov    dword[ebp+d4], 1  ; moveq.l  #1,d4
    ror    dword[ebp+d4], 1  ; ror.l  #1,d4

    ; Init probabilities
    mov    edx, NUM_CONTEXTS ; move.l #NUM_CONTEXTS, d6
.init:
    ; move.w  #INIT_ONE_PROB,-(a7)
    mov    word[prob+ebp+edx*2-2], INIT_ONE_PROB
    sub    dx, 1             ; subq.w #1,d6
    jne    .init             ; bne.b  .init
```

```
    ; D6 = 0
.lit:
    ; Literal
    add    dl, 1             ; addq.b #1,d6
.getlit:
    call   GetBit            ; bsr.b  GetBit
    adc    dl, dl            ; addx.b d6,d6
    jnc    .getlit           ; bcc.b  .getlit

    mov    [edi], dl         ; move.b d6,(a5)+
    inc    edi
                             ; bsr.b  ReportProgress
.switch:
    ; After literal
    call   GetKind           ; bsr.b  GetKind
    jnc    .lit              ; bcc.b  .lit
    ; Reference
    mov    edx, -1           ; moveq.l  #-1,d6
    call   GetBit            ; bsr.b  GetBit
    jnc    .readoffset       ; bcc.b  .readoffset
.readlength:
    mov    edx, 4            ; moveq.l  #4,d6
    call   GetNumber         ; bsr.b  GetNumber
.copyloop:
    mov    al, [edi + ebx]   ; move.b (a5,d5.l),(a5)+
    stosb
    sub    ecx, 1            ; subq.l #1,d7
    jne    .copyloop         ; bne.b  .copyloop
                             ; bsr.b  ReportProgress
    ; After reference
    call   GetKind           ; bsr.b  GetKind
    jnc    .lit              ; bcc.b  .lit
.readoffset:
    mov    edx, 3            ; moveq.l  #3,d6
    call   GetNumber         ; bsr.b  GetNumber
    mov    ebx, 2            ; moveq.l  #2,d5
    sub    ebx, ecx          ; sub.l  d7,d5
    jne    .readlength       ; bne.b  .readlength

    add    esp, 4096         ; lea.l  NUM_CONTEXTS*2(a7),a7
    sub    edi, [esp+32+8]
    mov    [esp+pushad_t.eax], edi
    popad                    ; movem.l  (a7)+,d2-d7/a4-a6
    ret                      ; rts

ReportProgress:
    ; move.l  a2,d0
    ; beq.b .nocallback
    ; move.l  a5,d0
    ; sub.l a6,d0
    ; move.l  a3,a0
    ; jsr (a2)
.nocallback:
    ; rts
```

```
GetKind:
    ; Use parity as context
                            ; move.l a5,d1
    mov    edx, 1           ; moveq.l  #1,d6
    and    edx, edi         ; and.l  d1,d6
    shl    dx, 8            ; lsl.w  #8,d6
    jmp    GetBit           ; bra.b  GetBit

GetNumber:
    ; EDX = Number context
    ; Out: Number in ECX
    shl    dx, 8            ; lsl.w  #8,d6
.numberloop:
    add    dl, 2            ; addq.b #2,d6
    call   GetBit           ; bsr.b  GetBit
    jc     .numberloop      ; bcs.b  .numberloop
    mov    ecx, 1           ; moveq.l  #1,d7
    sub    dl, 1            ; subq.b #1,d6
.bitsloop:
    call   GetBit           ; bsr.b  GetBit
    adc    ecx, ecx         ; addx.l d7,d7
    sub    dl, 2            ; subq.b #2,d6
    jnc    .bitsloop        ; bcc.b  .bitsloop
    ret                     ; rts

    ; EDX = Bit context

    ; d2 = Range value
    ; d3 = Interval size
    ; d4 = Input bit buffer

    ; Out: Bit in C and X
readbit:
    mov    eax, [ebp+d4]
    add    eax, eax         ; add.l  d4,d4
    jne    nonewword        ; bne.b  nonewword
    lodsd                   ; move.l (a4)+,d4
    bswap  eax              ; data is stored in big-endian format
    adc    eax, eax         ; addx.l d4,d4
nonewword:
    mov    [ebp+d4], eax
    mov    [esp+pushad_t.esi], esi
    adc    bx, bx           ; addx.w d2,d2
    add    cx, cx           ; add.w  d3,d3
    jmp    check_interval
GetBit:
    pushad
    mov    ebx, [ebp+d2]
    mov    ecx, [ebp+d3]
check_interval:
    test   cx, cx           ; tst.w  d3
    jns    readbit          ; bpl.b  readbit

    ; lea.l 4+SINGLE_BIT_CONTEXTS*2(a7,d6.l),a1
    ; add.l d6,a1
```

```
    lea    edi, [ebp+prob+2*edx+SINGLE_BIT_CONTEXTS*2]
    movzx  eax, word[edi]    ; move.w (a1),d1
    ; D1/EAX = One prob

    shr    ax, ADJUST_SHIFT  ; lsr.w  #ADJUST_SHIFT,d1
    sub    [edi], ax         ; sub.w  d1,(a1)
    add    ax, [edi]         ; add.w  (a1),d1

    mul    cx                ; mulu.w d3,d1
                             ; swap.w d1

    sub    bx, dx            ; sub.w  d1,d2
    jb     .one              ; blo.b  .one
.zero:
    ; oneprob = oneprob * (1 - adjust) = oneprob - oneprob * adjust
    sub    cx, dx            ; sub.w  d1,d3
    ; 0 in C and X
                             ; rts
    jmp    exit_get_bit
.one:
    ; onebrob = 1 - (1 - oneprob) * (1 - adjust) = oneprob - oneprob * adjust +
adjust
    ; add.w #$ffff>>ADJUST_SHIFT,(a1)
    add    word[edi], 0xFFFF >> ADJUST_SHIFT
    mov    cx, dx            ; move.w d1,d3
    add    bx, dx            ; add.w  d1,d2
    ; 1 in C and X
exit_get_bit:
    mov    word[ebp+d2], bx
    mov    word[ebp+d3], cx
    popad
    ret                      ; rts
```

The following is my own attempt to implement a size-optimized version of the same depacker in x86 assembly. However, there’s likely room for improvement here, and this code will be updated later.

```
    %define INIT_ONE_PROB       0x8000
    %define ADJUST_SHIFT        4
    %define SINGLE_BIT_CONTEXTS 1
    %define NUM_CONTEXTS        1536

    struc pushad_t
      .edi resd 1
      .esi resd 1
      .ebp resd 1
      .esp resd 1
      .ebx resd 1
      .edx resd 1
      .ecx resd 1
      .eax resd 1
    endstruc

    struc shrinkler_ctx
      .esp      resd 1      ; original value of esp before allocation
      .range    resd 1      ; range value
      .ofs      resd 1
      .interval resd 1      ; interval size
    endstruc

    bits 32

    %ifndef BIN
      global shrinkler_depackx
      global _shrinkler_depackx
    %endif

shrinkler_depackx:
_shrinkler_depackx:
    pushad
    mov    ebx, [esp+32+4]   ; edi = outbuf
    mov    esi, [esp+32+8]   ; esi = inbuf

    mov    eax, esp
    xor    ecx, ecx          ; ecx = 4096
    mov    ch, 10h
    sub    esp, ecx          ; subtract 1 page
    test   [esp], esp        ; stack probe

    mov    edi, esp
    stosd                    ; save original value of esp
    cdq
    xchg   eax, edx
    stosd                    ; range value = 0
    stosd                    ; offset = 0
    inc    eax
    stosd                    ; interval length = 1

    call   init_get_bit
GetBit:
    pushad
    mov    ebp, [ebx+shrinkler_ctx.range   ]
```

```
    mov    ecx, [ebx+shrinkler_ctx.interval]
    jmp    check_interval
readbit:
    add    al, al
    jne    nonewword
    lodsb
    adc    al, al
nonewword:
    mov    [esp+pushad_t.eax], eax
    mov    [esp+pushad_t.esi], esi
    adc    ebp, ebp
    add    ecx, ecx
check_interval:
    test   cx, cx
    jns    readbit

    lea    edi, [shrinkler_ctx_size + ebx + 2*edx + SINGLE_BIT_CONTEXTS*2]
    mov    ax, word[edi]

    shr    eax, ADJUST_SHIFT
    sub    [edi], ax
    add    ax, [edi]

    cdq
    mul    cx

    sub    ebp, edx
    jc    .one
.zero:
    ; oneprob = oneprob * (1 - adjust) = oneprob - oneprob * adjust
    sub    ecx, edx
    ; 0 in C and X
    jmp    exit_getbit
.one:
    ; onebrob = 1 - (1 - oneprob) * (1 - adjust) = oneprob - oneprob * adjust +
adjust
    add    word[edi], (0xFFFF >> ADJUST_SHIFT)
    xchg   edx, ecx
    add    ebp, ecx
    ; 1 in C and X
exit_getbit:
    mov    [ebx+shrinkler_ctx.range   ], ebp
    mov    [ebx+shrinkler_ctx.interval], ecx
    popad
    ret
GetKind:
    ; Use parity as context
    mov    edx, edi
    and    edx, 1
    shl    edx, 8
    jmp    ebp
GetNumber:
    cdq
    adc    dh, 3
.numberloop:
```

```
    inc    edx
    inc    edx
    call   ebp
    jc    .numberloop
    push   1
    pop    ecx
    dec    edx
.bitsloop:
    call   ebp
    adc    ecx, ecx
    sub    dl, 2
    jnc   .bitsloop
    ret

init_get_bit:
    pop    ebp               ; ebp = GetBit

    ; Init probabilities
    mov    ch, NUM_CONTEXTS >> 8
    xor    eax, eax
    mov    ah, 1<<7
    rep    stosw
    xchg   al, ah

    mov    edi, ebx
    mov    ebx, esp

    ; edx = 0
    cdq
.lit:
    ; Literal
    inc    edx
.getlit:
    call   ebp
    adc    dl, dl
    jnc    .getlit

    mov    [edi], dl
    inc    edi
.switch:
    ; After literal
    call   GetKind
    jnc    .lit

    ; Reference
    cdq
    dec    edx
    call   ebp
    jnc    .readoffset
.readlength:
    clc
    call   GetNumber
    push   esi
    mov    esi, edi
    add    esi, dword[ebx+shrinkler_ctx.ofs]
```

```
    rep    movsb
    pop    esi

    ; After reference
    call   GetKind
    jnc   .lit
.readoffset:
    stc
    call   GetNumber
    neg    ecx
    inc    ecx
    inc    ecx
    mov    [ebx+shrinkler_ctx.ofs], ecx
    jne   .readlength

    ; return depacked length
    mov    esp, [ebx+shrinkler_ctx.esp]
    sub    edi, [esp+32+4]
    mov    [esp+pushad_t.eax], edi
    popad
    ret
```

12. C/x86 assembly

The following algorithms were translated from C to x86 assembly or were already implemented in x86 assembly and optimized for size.

12.1 Lempel-Ziv Ross Williams (LZRW)

Designed by Ross Williams and described in An Extremely Fast Ziv-Lempel Data Compression Algorithm published in 1991. The compression ratio is only slightly worse than LZ77 but is much faster at compression.

```
lzrw1_depack:
_lzrw1_depack:
    pushad
    lea    esi, [esp+32+4]
    lodsd
    xchg   edi, eax        ; edi = outbuf
    lodsd
    xchg   ebp, eax        ; ebp = inlen
    lodsd
    xchg   esi, eax        ; esi = inbuf
    add    ebp, esi        ; ebp = inbuf + inlen
L0:
    push   16 + 1          ; bits = 16
    pop    edx
    lodsw                  ; ctrl = *in++, ctrl |= (*in++) << 8
    xchg   ebx, eax
L1:
    ; while(in != end) {
    cmp    esi, ebp
    je     L4
    ; if(--bits == 0) goto L0
    dec    edx
    jz     L0
L2:
    ; if(ctrl & 1) {
    shr    ebx, 1
    jc     L3
    movsb                  ; *out++ = *in++;
    jmp    L1
L3:
    lodsb                  ; ofs = (*in & 0xF0) << 4
    aam    16
    cwde
    movzx  ecx, al
    inc    ecx
    lodsb                  ; ofs |= *in++ & 0xFF;
    push   esi             ; save pointer to in
    mov    esi, edi        ; ptr  = out - ofs;
    sub    esi, eax
    rep    movsb           ; while(len--) *out++ = *ptr++;
    pop    esi             ; restore pointer to in
    jmp    L1
L4:
    sub    edi, [esp+32+4] ; edi = out - outbuf
    mov    [esp+28], edi   ; esp+_eax = edi
    popad
    ret
```

12.2 Ultra-fast LZ (ULZ)

Ultra-fast LZ was first published by Ilya “encode” Muravyov in 2010 and then appears to have been open sourced in 2019. The following code is a straightforward translation of the C decoder to x86 assembly.

```
static uint32_t add_mod(uint32_t x, uint8_t** p);

uint32_t ulz_depack(
  void *outbuf,
  uint32_t inlen,
  const void *inbuf)
{
    uint8_t  *ptr, *in, *end, *out;
    uint32_t dist, len;
    uint8_t  token;

    out = (uint8_t*)outbuf;
    in  = (uint8_t*)inbuf;
    end = in + inlen;

    while(in < end) {
      token = *in++;
      if(token >= 32) {
        len = token >> 5;
        if(len == 7)
          len = add_mod(len, &in);
        while(len--) *out++ = *in++;
        if(in >= end) break;
      }
      len = (token & 15) + 4;
      if(len == (15 + 4))
        len = add_mod(len, &in);
      dist = ((token & 16) << 12) + *(uint16_t*)in;
      in += 2;
      ptr = out - dist;
      while(len--) *out++ = *ptr++;
    }
    return (uint32_t)(out - (uint8_t*)outbuf);
}

static uint32_t add_mod(uint32_t x, uint8_t** p) {
    uint8_t c, i;

    for(i=0; i<=21; i+=7) {
      c = *(*p)++;
      x += (c << i);
      if(c < 128) break;
    }
    return x;
}
```

```
ulz_depack:
_ulz_depack:
    pushad
    lea    esi, [esp+32+4]
    lodsd
    xchg   edi, eax          ; edi = outbuf
    lodsd
    xchg   ebx, eax
    lodsd
    xchg   esi, eax          ; esi = inbuf
    add    ebx, esi          ; ebx += inbuf
ulz_main:
    xor    ecx, ecx
    mul    ecx
    ; while (in < end) {
    cmp    esi, ebx
    jnb    ulz_exit
    ; token = *in++;
    lodsb
    ; if(token >= 32) {
    cmp    al, 32
    jb     ulz_copy2
    ; len = token >> 5
    mov    cl, al
    shr    cl, 5
    ; if(len == 7)
    cmp    cl, 7
    jne    ulz_copy1
    ; len = add_mod(len, &in);
    call   add_mod
ulz_copy1:
    ; while(len--) *out++ = *in++;
    rep    movsb
    ; if(in >= end) break;
    cmp    esi, ebx
    jae    ulz_exit
ulz_copy2:
    ; len = (token & 15) + 4;
    mov    cl, al
    and    cl, 15
    add    cl, 4
    ; if(len == (15 + 4))
    cmp    cl, 15 + 4
    jne    ulz_copy3
    ; len = add_mod(len, &in);
    call   add_mod
ulz_copy3:
    ; dist = ((token & 16) << 12) + *(uint16_t*)in;
    and    al, 16
    shl    eax, 12
    xchg   eax, edx
    ; eax = *(uint16_t*)in;
    ; in += 2;
    lodsw
    add    edx, eax
```

```
    ; p = out - dist
    push   esi
    mov    esi, edi
    sub    esi, edx
    ; while(len--) *out++ = *p++;
    rep    movsb
    pop    esi
    jmp    ulz_main
    ; }
ulz_exit:
    ; return (uint32_t)(out - (uint8_t*)outbuf);
    sub    edi, [esp+32+8]
    mov    [esp+28], edi
    popad
    ret

; static uint32_t add_mod(uint32_t x, uint8_t** p);
add_mod:
    push   eax                ; save eax
    xchg   eax, ecx           ; eax = len
    xor    ecx, ecx           ; i = 0
am_loop:
    mov    dl, byte[esi]      ; c = *(*p)++
    inc    esi
    push   edx                ; save c
    shl    edx, cl            ; x += (c << i)
    add    eax, edx
    pop    edx                ; restore c
    cmp    dl, 128            ; if(c < 128) break;
    jb     am_exit
    add    cl, 7              ; i+=7
    cmp    cl, 21             ; i<=21
    jbe    am_loop
am_exit:
    xchg   eax, ecx           ; ecx = len
    pop    eax                ; restore eax
    ret
```

12.3 BriefLZ

Designed by Jørgen Ibsen and published in 2015. BriefLZ combines fast encoding and decoding with a good compression ratio. Ibsen uses 16-Bit tags instead of 8-Bit to improve performance on 16-bit architectures. It encodes the match reference length and offset using Elias gamma coding. The following size-optimized decoder in x86 assembly is only 92 bytes.

```
blz_depack:
_blz_depack:
    pushad
    lea    esi, [esp+32+4]   ;
    lodsd
    xchg   edi, eax          ; bs.dst = outbuf
    lodsd
    lea    ebx, [edi+eax]    ; end = bs.dst + outlen
    lodsd
    xchg   esi, eax          ; bs.src = inbuf
    call   blz_init_getbit
blz_getbit:
    add    ax, ax            ; tag <<= 1
    jnz    blz_exit_getbit   ; continue for all bits
    lodsw                    ; read 16-bit tag
    adc    ax, ax            ; carry over previous bit
blz_exit_getbit:
    ret
blz_init_getbit:
    pop    ebp               ; ebp = blz_getbit
    mov    ax, 8000h         ;
blz_literal:
    movsb                    ; *out++ = *bs.src++
blz_main:
    cmp    edi, ebx          ; while(out < end)
    jnb    blz_exit

    call   ebp               ; cf = blz_getbit
    jnc    blz_literal       ; if(cf==0) goto blz_literal
                             ;
blz_getgamma:
    pushfd                   ; save cf
    cdq                      ; result = 1
    inc    edx
blz_gamma_loop:
    call   ebp               ; cf = blz_getbit()
    adc    edx, edx          ; result = (result << 1) + cf
    call   ebp               ; cf = blz_getbit()
    jc     blz_gamma_loop    ; while(cf == 1)

    popfd                    ; restore cf
    cmovc  ecx, edx          ; ecx = cf ? edx : ecx
    cmc                      ; complement carry
    jnc    blz_getgamma      ; loop twice

    ; ofs = blz_getgamma(&bs) - 2;
    dec    edx
    dec    edx

    ; len = blz_getgamma(&bs) + 2;
    inc    ecx
    inc    ecx

    ; ofs = (ofs << 8) + (uint32_t)*bs.src++ + 1;
    shl    edx, 8
```

```
    mov    dl, [esi]
    inc    esi
    inc    edx

    ; ptr = out - ofs;
    push   esi
    mov    esi, edi
    sub    esi, edx
    rep    movsb
    pop    esi
    jmp    blz_main
blz_exit:
    ; return (out - (uint8_t*)outbuf);
    sub    edi, [esp+32+4]
    mov    [esp+28], edi
    popad
    ret
```

12.4 Not Really Vanished (NRV)

Designed by Markus F.X.J. Oberhumer and used in the famous Ultimate Packer for eXecutables (UPX). NRV uses an LZ77 format with Elias gamma coding for the reference match offset and length. The following x86 assembly derived from n2b_d_s1.asm in the UCL library is currently 115 bytes.

```
nrv2b_depack:
_nrv2b_depack:
    pushad
    mov    edi, [esp+32+4]   ; output
    mov    esi, [esp+32+8]   ; input

    xor    ecx, ecx
    mul    ecx
    dec    edx
    mov    al, 0x80

    call   init_get_bit
    ; read next bit from input
    add    al, al
    jnz    exit_get_bit

    lodsb
    adc    al, al
exit_get_bit:
    ret
init_get_bit:
    pop    ebp
    jmp    nrv2b_main
    ; copy literal
nrv2b_copy_byte:
    movsb
nrv2b_main:
    call   ebp
    jc     nrv2b_copy_byte

    push   1
    pop    ebx
nrv2b_match:
    call   ebp
    adc    ebx, ebx

    call   ebp
    jnc    nrv2b_match

    sub    ebx, 3
    jb     nrv2b_offset

    shl    ebx, 8
    mov    bl, [esi]
    inc    esi
    xor    ebx, -1
    jz     nrv2b_exit

    xchg   edx, ebx
nrv2b_offset:
    call   ebp
    adc    ecx, ecx

    call   ebp
    adc    ecx, ecx
```

```
    jnz    nrv2b_copy_bytes

    inc    ecx
nrv2b_len:
    call   ebp
    adc    ecx, ecx

    call   ebp
    jnc    nrv2b_len

    inc    ecx
    inc    ecx
nrv2b_copy_bytes:
    cmp    edx, -0xD00
    adc    ecx, 1
    push   esi
    lea    esi, [edi + edx]
    rep    movsb
    pop    esi
    jmp    nrv2b_main
nrv2b_exit:
    ; return depacked length
    sub    edi, [esp+32+4]
    mov    [esp+28], edi
    popad
    ret
```

12.5 Lempel-Ziv-Markov chain Algorithm (LZMA)

Designed by Igor Pavlov and published in 1998 with the 7zip archiver. It's an LZ77 variant with features similar to LZX used for Microsoft CAB files and compressed help (CHM) files. LZMA uses an arithmetic coder to store compressed data as a stream of bits resulting in high compression ratios that inspired the development of Packfire, KKrunchy, and LZOMA, to name a few. There's a description by Charles Bloom in De-obfuscating LZMA and by Matt Mahoney in Data Compression Explained. Alex Ionescu has also published a minimal implementation with very detailed and helpful comments included in the source. Another size-optimized version is available from the UPX LZMA SDK. The arithmetic coder for LZMA usually requires 16KB of RAM and may not be suitable for devices with limited resources. mudlord's Win32 executable packer called mupack has an x86 implementation.

Although the compression ratio is excellent, and the speed is acceptable for small files. The complexity of the decompressor for only a few additional percents more in the compression ratio didn't merit an implementation in x86 assembly. I'd be willing to implement it on a better architecture like ARM64, but not x86. Shrinkler, KKrunchy, and LZOMA all offer ~55% ratios with much smaller RAM and ROM requirements that seem more suitable for executable compression.

12.6 Lempel–Ziv–Oberhumer-Markov Algorithm (LZOMA)

Designed by Alexandr Efimov and published in 2015. LZOMA is specifically for decompression of the Linux Kernel but is also suitable for decompression of PE or ELF files too. It's primarily based on ideas used by LZMA and LZO. It provides fast decompression like LZO, and a simplified LZMA format provides a high compression ratio. The trade-off is slow compression requiring a lot of memory. It's possible to improve the compression ratio by using a real entropy encoder, but at the expense of decompression speed. While it's still only an experimental algorithm and probably needs more testing, the following is a decoder in C and handwritten x86 assembly.

```
typedef struct _lzoma_ctx {
    uint32_t w;
    uint8_t  *src;
} lzoma_ctx;

static uint8_t get_bit(lzoma_ctx *c) {
    uint32_t cy, x;

    x = c->w;
    c->w <<= 1;

    // no bits left?
    if(c->w == 0) {
      // read 32-bit word
      x = *(uint32_t*)c->src;
      // advance input
      c->src += 4;
      // double with carry
      c->w = (x << 1) | 1;
    }
    // return carry bit
    return (x >> 31);
}

void lzoma_depack(
  void *outbuf,
  uint32_t inlen,
  const void *inbuf)
{
    uint8_t   *out, *ptr, *end;
    uint32_t  cf, top, total, len, ofs, x, res;
    lzoma_ctx c;

    c.w    = 1 << 31;
    c.src  = (uint8_t*)inbuf;
    out    = (uint8_t*)outbuf;
    end    = out + inlen;

    // copy first byte
    *out++ = *c.src++;
    len    = 0;
    ofs    = -1;

    while(out < end) {
      for(;;) {
        // if bit carried, break
        if(cf = get_bit(&c)) break;
        // copy byte
        *out++ = *c.src++;
        len = 2;
      }
      // unpack lz
      if(len) {
        cf = get_bit(&c);
      }
```

```
// carry?
if(cf) {
  len   = 3;
  total = out - (uint8_t*)outbuf;
  top   = ((total <= 400000) ? 60 : 50);
  ofs   = 0;
  x     = 256;
  res   = *c.src++;

  for(;;) {
    x += x;
    if(x >= (total + top)) {
      x -= total;
      if(res >= x) {
        cf = get_bit(&c);
        res = (res << 1) + cf;
        res -= x;
      }
      break;
    }
    // magic?
    if(x & (0x002FFE00 << 1)) {
      top = (((top << 3) + top) >> 3);
    }
    if(res < top) break;

    ofs -= top;
    total += top;
    top <<= 1;
    cf = get_bit(&c);
    res = (res << 1) + cf;
  }
  ofs += res + 1;
  // long length?
  if(ofs >= 5400) len++;
  // huge length?
  if(ofs >= 0x060000) len++;
  // negate
  ofs =- ofs;
}

if(get_bit(&c)) {
  len += 2;
  res = 0;
  for(;;) {
    cf = get_bit(&c);
    res = (res << 1) + cf;
    if(!get_bit(&c)) break;
    res++;
  }
  len += res;
} else {
  cf = get_bit(&c);
  len += cf;
}
```

```
      ptr = out + ofs;
      while(--len) *out++ = *ptr++;
    }
}
```

The assembly code doesn't transfer that well on to x86. It does, however, avoid having to use lots of RAM, which is a plus.

```
lzoma_depack:
_lzoma_depack:
    pushad                   ; save all registers
    lea    esi, [esp+32+4]
    lodsd
    xchg   edi, eax          ; edi = outbuf
    lodsd
    xchg   ebp, eax          ; ebp = inlen
    add    ebp, edi          ; ebp += out
    lodsd
    xchg   esi, eax          ; esi = inbuf
    pushad                   ; save esi, edi and ebp
    call   init_getbit
get_bit:
    add    eax, eax          ; c->w <<= 1
    jnz    exit_getbit       ; if(c->w == 0)
    lodsd                    ; x = *(uint32_t*)c->src;
    adc    eax, eax          ; c->w = (x << 1) | 1;
exit_getbit:
    ret                      ; return x >> 31;
init_getbit:
    pop    ebp               ; ebp = &get_bit
    mov    eax, 1 << 31      ; c->w = 1 << 31
    cdq                      ; ofs = -1
    movsb                    ; *out++ = *src++;
    xor    ecx, ecx          ; len = 0
    jmp    main_loop
copy_byte:
    movsb                    ; *out++ = *c.src++;
    mov    cl, 2             ; len = 2
main_loop:
    xor    ebx, ebx          ; res = 0

    ; while(out < end)
    cmp    edi, [esp+pushad_t._ebp]
    jnb    lzoma_exit

    ; for(;;) {
    call   ebp               ; cf = get_bit(&c);
    jnc    copy_byte         ; if(cf) break;

    ; unpack lz
    jecxz  skip_lz           ; if(len) {
    call   ebp               ;   cf = get_bit(&c);
skip_lz:                     ; }
    ; carry?
    jnc    use_last_offset   ; if(cf) {
    mov    cl, 3+2           ;   len = 3
    pushad                   ;
    ; total = out - (uint8_t*)outbuf
    sub    edi, [esp+32+pushad_t._edi]
    ; top = ((total <= 400000) ? 60 : 50;
    mov    cl, 50
    cmp    edi, 400000
    ja     skip_upd
```

```
    add    cl, 10
skip_upd:
    xor    ebp, ebp          ; ofs = 0
    xor    edx, edx          ; x = 256
    inc    dh
    mov    bl, byte[esi]     ; res = *c.src++
    inc    esi
find_loop:                   ; for(;;) {
    add    edx, edx          ;   x += x;
    ; if(x >= (total + top)) {
    push   edi               ; save total
    add    edi, ecx          ; edi = total + top
    cmp    edx, edi          ; cf = (x - (total + top))
    pop    edi               ; restore total
    jb     upd_len3          ; jump if x is < (total + top)

    sub    edx, edi          ; x -= total;
    cmp    ebx, edx          ; if(res >= x) {
    jb     upd_len2          ; jump if res < x

    ; cf = get_bit(&c);
    call   dword[esp+pushad_t._ebp]
    adc    ebx, ebx          ; res = (res << 1) + cf;
    sub    ebx, edx          ; res -= x;
    jmp    upd_len2
upd_len3:
    ; magic?
    ; if(x & (0x002FFE00 << 1)) {
    test   edx, (0x002FFE00 << 1)
    jz     upd_len4

    ; top = (((top << 3) + top) >> 3);
    lea    ecx, [ecx+ecx*8]
    shr    ecx, 3
upd_len4:
    cmp    ebx, ecx          ; if(res < top) break;
    jb     upd_len2

    sub    ebp, ecx          ; ofs -= top
    add    edi, ecx          ; total += top
    add    ecx, ecx          ; top <<= 1

    ; cf = get_bit(&c);
    call   dword[esp+pushad_t._ebp]

    ; res = (res << 1) + cf;
    adc    ebx, ebx
    jmp    find_loop
upd_len2:
    ; ofs = (ofs + res + 1);
    lea    ebp, [ebp + ebx + 1]

    ; if(ofs >= 5400) len++;
    cmp    ebp, 5400
    sbb    dword[esp+pushad_t._ecx], 0
```

```
    ; if(ofs >= 0x060000) len++;
    cmp    ebp, 0x060000
    sbb    dword[esp+pushad_t._ecx], 0

    neg    ebp                ; ofs = -ofs;

    mov    [esp+pushad_t._edx], ebp ; save ofs in edx
    mov    [esp+pushad_t._esi], esi
    mov    [esp+pushad_t._eax], eax
    popad                     ; restore registers
use_last_offset:
    call   ebp                ; if(get_bit(&c)) {
    jnc    check_two

    add    ecx, 2             ; len += 2
upd_len:                      ; for(res=0;;res++) {
    call   ebp                ; cf = get_bit(&c);
    adc    ebx, ebx           ; res = (res << 1) + cf;

    call   ebp                ; if(!get_bit(&c)) break;
    jnc    upd_lenx

    inc    ebx                ; res++;
    jmp    upd_len
upd_lenx:
    add    ecx, ebx           ; len += res
    jmp    copy_bytes
check_two:                    ; } else {
    call   ebp                ;   cf = get_bit();
    adc    ecx, ebx           ;   len += cf
copy_bytes:                   ; }
    push   esi                ; save c.src pointer
    lea    esi, [edi + edx]   ; ptr = out + ofs
    dec    ecx
    ; while(--len) *out++ = *ptr++;
    rep    movsb
    pop    esi                ; restore c.src
    jmp    main_loop
lzoma_exit:
    popad                     ; free()
    popad                     ; restore registers
    ret
```

12.7 KKrunchy

Designed by Fabian Giesen for the demo group, Farbrausch, KKrunchy comprises two algorithms. The first, developed between 2003 and 2005, is an LZ77 variant with an arithmetic coder published in 2006. The second algorithm developed between 2006 and 2008, borrows ideas from PAQ7 and was published in 2011. Both are slow at compression but acceptable for demo productions and are compact for decompression. Fabian describes both in more detail here, including the "secret ingredient" that can improve ratios of 64K intros by up to 10%. In 2011, Farbrausch members published source code for their demo productions made between 2001-2011, including both compressors. A 32-Bit x86 decoder is already available from Fabian. There appears to be a buffer overflow in the compressor that goes unnoticed without address sanitizer. Here's an alternate version of the simple depacker used as a reference.

```
#ifdef linux
// gcc
#define REV(x) __builtin_bswap32(x)
#else
// msvc
#define REV(x) _byteswap_ulong(x)
#endif

typedef struct _fr_state {
    const uint8_t *src;
    // range decoder values
    uint32_t val, len, pbs[803];
} fr_state;

// decode a bit using range decoder
static int DB(
  fr_state *s, int idx, uint32_t flag)
{
    uint32_t a, b, c, d, e;

    a = s->pbs[idx];
    b = (s->len >> 11) * a;
    c = (s->val >= b);
    d = -c; e = c-1;
    s->len = (d & s->len) | (e & b);
    a = (d & a) | (e & -a + 2048);
    a >>= (5 - flag);
    s->pbs[idx] += (a ^ d) + c;
    d &= b;
    s->val -= d; s->len -= d;
    a = (s->len >> 24);
    a = a == 0 ? -1 : 0;
    b = (a & 0xFF) & *s->src;
    d = -a;
    s->src += d;
    s->val = (s->val << (d << 3)) | b;
    s->len = (s->len << (d << 3));
    return c;
}

// decode tree
static int DT(
  fr_state *s, int p, int bits)
{
    int c;

    for(c=1; c<bits;) {
      c = (c+c) + DB(s, p + c, bits==256);
    }
    return c - bits;
}

// decode gamma
static int DG(fr_state *s, int flag) {
    int     v, x = 1;
```

```
    uint8_t c = 1;

    v = (-flag & (547 - 291)) + 291;

    do {
      c = (c+c) + DB(s, v+c, 0);
      x = (x+x) + DB(s, v+c, 0);
      c = (c+c) + (x & 1);
    } while(c & 2);

    return x;
}

uint32_t fr_depack(
  void *outbuf,
  const void *inbuf)
{
    int      tmp, i, ofs, len, LWM;
    uint8_t  *ptr, *out = (uint8_t*)outbuf;
    fr_state s;

    s.src  = (const uint8_t*)inbuf;
    s.len  = ~0;
    s.val  = REV(*(uint32_t*)s.src);
    s.src += 4;

    for(i=0; i<803; i++) s.pbs[i] = 1024;

    for(;;) {
      LWM = 0;
      // decode literal
      *out++ = DT(&s, 35, 256);
    fr_read_bit:
      if(!DB(&s, LWM, 0)) continue;
      // decode match
      len = 0;
      // use previous offset?
      if(LWM || !DB(&s, 2, 0)) {
        ofs = DG(&s, 0);
        if(!ofs) break;

        len  = 2;
        ofs  = ((ofs - 2) << 4);
        tmp  = ((ofs != 0 ? -1 : 0) & 16) + 3;
        ofs += DT(&s, tmp, 16) + 1;

        len -= (ofs < 2048);
        len -= (ofs < 96);
      }
      LWM  = 1;
      len += DG(&s, 1);
      ptr  = out - ofs;

      while(len--) *out++ = *ptr++;
      goto fr_read_bit;
```

```
    }
    return out - (uint8_t*)outbuf;
}
```

13. Results

The following table, while ordered by ratio, is NOT a rank order and shouldn't be interpreted that way. It wouldn't be fair to judge the algorithms based on my criteria, that is: lightweight decompressor, high compression ratio, open source. The ratios are based on compressing a 1MB PE file for Windows without any additional trickery.

Algorithm	RAM (Bytes)	ROM (Bytes)	Ratio
LZ77	0	54	32%
ZX7 Mini	0	67	36%
LZSS	0	69	40%
LZ4	0	80	43%
ULZ	0	124	44%
LZE	0	97	45%
ZX7	0	81	46%
MegaLZ	0	117	46%
BriefLZ	0	92	46%
LZSA1	0	96	46%
LZSA2	0	187	50%
NRV2b	0	115	51%
LZOMA	0	238	54%
Shrinkler	4096	235	55%
KKrunchy	3212	639 (compiler generated)	55%
LZMA	16384	1265 (compiler generated)	58%

14. Summary

One could surely write a book about compression algorithms used by the Demoscene. And it's safe to say I've only scraped the surface on this subject. For example, there is no analysis of compression and decompression speed of implementations for the x86 or other architectures. My primary concern at the moment is in the compression ratio and code size.

15. Acknowledgements

A number of people helped directly or indirectly with this post.

- Tim Bell for LZB and information about the Stac Electronics lawsuit.
- Blueberry for optimization tips and fixing my initial 68K translation of Shrinkler.
- Qkumba for fixing x86 translation, translation of Exomizer and 6502 depackers.
- Trixter for 8088 depackers.
- Introspec for Z80 depackers and impressive knowledge of LZ variations.
- Emmanuel Marty for aPUltra, LZSA, and helping with x86 decoder for aPLib.

16. Further Research

To save you time locating information about some of the topics discussed in this post, I've included some links to get you started.

16.1 Documentaries and Interviews

16.2 Websites, Blogs and Forums

16.3 Demoscene Productions

This is not a "best of" list or what my favorites are. It's mainly from some youtube recommendations and please don't take offense If I didn't include your demo. Contact me if you feel I've missed any.

16.4 Tools

16.5 Other Compression Algorithms

The following table, while ordered by ratio, is NOT a rank order and shouldn't be interpreted that way. It wouldn't be fair to judge the algorithms based on my criteria, which is a lightweight decompressor, high compression ratio, open-source. The compression ratios are from compressing a 1MB PE file for Windows.

OK/Good (~25-39%)

Library / API / Algorithm	Ratio	Link

zpack	24%	https://github.com/zerkman/zpacker
PPP	27%	https://tools.ietf.org/html/rfc1978
JQCoding	27%	https://encode.su/threads/2157-Looking-for-a-super-simple-decompressor?p=43099&viewfull=1#post43099
LZJB	28%	https://github.com/nemequ/lzjb
LZRW1	31%	http://ross.net/compression/lzrw1.html
LZ48	31%	http://www.cpcwiki.eu/forum/programming/lz48-cruncherdecruncher/
LZ77	32%	https://github.com/andyherbert/lz1
LZW	33%	https://github.com/vapier/ncompress
LZP1	34%	http://www.hugi.scene.org/online/coding/hugi%2012%20-%20colzp.htm
Kitty	34%	https://encode.su/threads/2174-Kitty-file-compressor-(Super-small-compressor)
LZ49	35%	http://www.cpcwiki.eu/forum/programming/lz48-cruncherdecruncher/
LZ4X	36%	https://github.com/encode84/lz4x
QuickLZ	36%	http://www.quicklz.com/
ZX7mini	36%	https://github.com/antoniovillena/zx7mini
RtlDecompressBuffer (LZNT1)	36%	Windows OS
Decompress (Xpress)	37%	Windows OS.

Very Good (40-49%)

Library / API / Algorithm	Ratio	Link
LZSS	40%	https://github.com/kieselsteini/lzss
LZF	40%	https://encode.su/threads/1819-LZF-Optimized-LZF-compressor
LZM	41%	https://github.com/r-lyeh/stdarc.c

RtlDecompressBuffer (Xpress)	43%	Windows OS
BLZ4	43%	https://github.com/jibsen/blz4
LZ4Ultra	43%	https://github.com/emmanuel-marty/lz4ultra
ULZ	44%	https://github.com/encode84/ulz
BitBuster	44%	https://www.teambomba.net/bombaman/downloadd26a.html
LZE	45%	http://gorry.haun.org/pw/?lze
Decompress (Xpress Huffman)	45%	Windows OS
ZX7	45%	http://www.worldofspectrum.org/infoseekid.cgi?id=0027996
LZMAT	45%	http://www.matcode.com/lzmat.htm
CRUSH	45%	https://sourceforge.net/projects/crush/
Hrust	46%	https://github.com/specke/ohc
MegaLZ	46%	http://os4depot.net/index.php?function=showfile&file=development/cross/megalz.lha
LZSA1	46%	https://github.com/emmanuel-marty/lzsa
BriefLZ	46%	https://github.com/jibsen/brieflz
apUltra	47%	https://github.com/emmanuel-marty/apultra
Pletter5	47%	http://www.xl2s.tk/
Pucrunch	48%	https://github.com/mist64/pucrunch
SR2	48%	http://mattmahoney.net/dc/#sr2

Excellent (50% >)

Library / API / Algorithm	Ratio	Link
BCRUSH	50%	https://github.com/jibsen/bcrush
LZSA2	50%	https://github.com/emmanuel-marty/lzsa
RtlDecompressBufferEx (Xpress Huffman)	50%	Windows OS

Decompress (MSZip)	51%	Windows OS
Exomizer	51%	https://bitbucket.org/magli143/exomizer/wiki/Home
aPLib	51%	http://ibsensoftware.com/products_aPLib.html
JCALG1	52%	https://bitsum.com/portfolio/jcalg1/
NRV2B	52%	http://www.oberhumer.com/opensource/ucl/
BALZ	53%	https://sourceforge.net/projects/balz/
Decompress (LZMS)	54%	Windows OS
LZOMA	54%	https://github.com/alef78/lzoma
KKrunchy	55%	https://github.com/farbrausch/fr_public
Shrinkler	55%	https://github.com/askeksa/Shrinkler
NLZM	55%	https://github.com/nauful/NLZM
BCM	55%	https://github.com/encode84/bcm
D3DDecompressShaders (DXT/BC)	57%	Windows OS
Packfire	57%	http://neural.untergrund.net/
LZMA	58%	https://www.7-zip.org/sdk.html
PAQ8F	70%	http://mattmahoney.net/dc/paq.html