ORIGINAL ARTICLE

Transcoding Unicode Characters with AVX-512 Instructions

Robert Clausecker[1*] | **Daniel Lemire**[2†]

[1]Zuse Institute Berlin, Germany

[2]DOT-Lab Research Center, Université du Québec (TELUQ), Montréal, Canada

Correspondence
Daniel Lemire, DOT-Lab Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada
Email: daniel.lemire@teluq.ca

Funding information
Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2017-03910

Intel includes in its recent processors a powerful set of instructions capable of processing 512-bit registers with a single instruction (AVX-512). Some of these instructions have no equivalent in earlier instruction sets. We leverage these instructions to efficiently transcode strings between the most common formats: UTF-8 and UTF-16. With our novel algorithms, we are often twice as fast as the previous best solutions. For example, we transcode Chinese text from UTF-8 to UTF-16 at more than 5 GiB s^{-1} using fewer than 2 CPU instructions per character. To ensure reproducibility, we make our software freely available as an open source library. Our library is part of the popular Node.js JavaScript runtime.

KEYWORDS
Vectorization, Unicode, Text Processing, Character Encoding

1 | INTRODUCTION

Computers store strings of text as arrays of bytes. Unicode is a standard for representing text as a sequence of *universal characters* represented by *code points*. Code points are stored as short sequences of bytes according to a given *unicode transformation format* (UTF), the most popular being UTF-8 and UTF-16. Not all sequence of bytes are valid UTF-8 or UTF-16 strings [1]. Validation is required to detect incorrectly encoded or corrupted text before it is processed.

We often need to transcode strings between the two formats. For example, a database might store data in UTF-16 and yet the programmer might need to produce UTF-8 strings for a web site. Thankfully, transcoding is relatively efficient. Conventional transcoders often achieve a throughput between 0.5 GiB s^{-1} to 1.5 GiB s^{-1} on commodity processors [2] (cf. § 8).[1] Yet this falls far below the sequential-read speed of a fast disk (e.g., 5 GiB s^{-1}) or the throughput

[1]The speed is measured by taking the size of the input and dividing by the time elapsed.

of a fast network connection.

IBM mainframes based on z/Architecture provide special-purposes instructions named "CONVERT UTF-8 TO UTF-16" and "CONVERT UTF-16 TO UTF-8" for translation between the two encodings [3]. By virtue of being implemented in hardware, these exceed $10\,\mathrm{GiB\,s^{-1}}$ processing speed for typical inputs. While commodity processors currently lack such dedicated instructions, they can benefit from single-instruction-multiple-data (SIMD) instructions. Unlike conventional instructions which operate on a single machine word (e. g. 64 bits), these SIMD instructions operate on larger registers (128 bits, 256 bits, …) representing vectors of numbers. A single SIMD instruction may add eight pairs of 16-bit words at once. We can transcode gigabytes of text per second [2] by a deliberate use of conventional SIMD instructions (e.g., ARM NEON, SSE, AVX2).

In recent years, Intel introduced new SIMD instruction sets operating over registers as wide as 512 bits. If Intel had merely doubled the width of the registers, there would be little need for further work on our part. However, our experience suggests that to fully benefit from AVX-512 instructions, we need to use adapted algorithms [4]. Indeed, while AVX-512 instructions benefit from wider registers, Intel has also added many more instructions than what is typically found in SIMD instruction sets. There is also a slightly different model: AVX-512 instruction may consume or generate masks in *mask* registers which have no equivalent in prior commodity instruction sets. In AVX-512, a mask is conceptually an array of 8, 16, 32, or 64 bits corresponding to vectors of 8, 16, 32, or 64 elements.

We present novel transcoding functions using AVX-512 instructions. On average, we are roughly twice as fast as the previous fastest functions [2] on commodity processors.

2 | UNICODE AND ITS ENCODINGS

Unicode is a standard based on the *Universal Character Set* (UCS). An extension to ASCII, UCS is a character set whose characters (called *universal characters*) have code points numbered from `U+0`[2] to `U+10FFFF` (decimal 1 114 111). These code points are organized into 17 *planes* of 65 536 characters each, with the first plane `U+0000–U+FFFF` being called the *Basic Multilingual Plane* (BMP). Code points in the range `0xd800–0xdfff` are reserved for *surrogates* used in the UTF-16 encoding and do not represent universal characters.

Unlike simpler character sets like ASCII, universal characters are seldomly stored directly as integers, as such a storage format is wasteful and incompatible to existing byte-oriented environments. Instead, several *Unicode Transformation Formats* (UTF) are employed to store and process universal characters, depending on the use case at hand.

A Unicode Transformation Format transforms each universal character into a sequence of integers, with the size of the integer being dependent on the format. Popular Unicode Transformation Formats include:

UTF-32 representing each universal character as a 32-bit integer. Mainly used as an internal representation.

UTF-16 representing each universal character as one or two 16-bit integers [5]. All code-point values up to `U+FFFF` are stored as 2-byte integer values directly. Otherwise we use surrogate pairs: two consecutive 2-byte values, each storing 10 bits of the codepoint. Used by Java, Windows NT, databases, binary protocols, and others.

UTF-8 representing each universal character as 1–4 bytes [6]. An extension to ASCII, UTF-8 is by far the most popular text encoding on the World Wide Web.

Though our software work covers many cases (from UTF-8 to UTF-16 or UTF-32, from UTF-16 to UTF-8 or UTF-32, and so forth), we study the two most difficult cases: from UTF-8 to UTF-16 and back.

Multi-byte words in computers representing numerical values can be stored in either little-endian format or big-endian format, depending on whether the first byte is the least significant or the most significant. Unicode Transfor-

[2] `U+` followed by a hexadecimal number is notation for a universal character's code point.

case	*UTF-16*	*UTF-8*
ASCII	0000 0000 0GFE DCBA	---- ---- ---- ---- ---- ---- 0GFE DCBA
2-byte	0000 0LKJ HGFE DCBA	---- ---- ---- ---- 110L KJHG 10FE DCBA
3-byte	RQPN MLKJ HGFE DCBA	---- ---- 1110 RQPN 10ML KJHG 10FE DCBA
4-byte	1101 10vu tsRQ PNML	1111 0WVU 10TS RQPN 10ML KJHG 10FE DCBA
	1101 11KJ HGFE DCBA	

(a) Bit-by-bit correspondence between UTF-16 and UTF-8 encodings in the four possible cases. The bits are named A to W starting at the least significant bits with 0vuts = WVUTS − 1.

codepoint	*UTF-16*	*UTF-8*
U+0	0000 0000 0000 0000	---- ---- ---- ---- ---- ---- 0000 0000
U+7F	0000 0000 0111 1111	---- ---- ---- ---- ---- ---- 0111 1111
U+80	0000 0000 1000 0000	---- ---- ---- ---- 1100 0010 1000 0000
U+7FF	0000 0111 1111 1111	---- ---- ---- ---- 1101 1111 1011 1111
U+800	0000 1000 0000 0000	---- ---- 1110 0000 1010 0000 1000 0000
U+FFFF	1111 1111 1111 1111	---- ---- 1110 1111 1011 1111 1011 1111
U+10000	1101 1000 0000 0000	1111 0000 1001 0000 1000 0000 1000 0000
	1101 1100 0000 0000	
U+10FFFF	1101 1011 1111 1111	1111 0100 1000 1111 1011 1111 1011 1111
	1101 1111 1111 1111	

(b) Examples of matched code-point values in UTF-32, UTF-16LE and UTF-8. For U+10000 and U+10FFFF, UTF-16 requires a surrogate pair.

FIGURE 1 Correspondence between UTF-16 and UTF-8. Format-specific prescribed bits (*tag bits*) are underlined.

mation Formats representing characters in units larger than bytes are subject to endianess. If the endianess is not known from the context[3], it can be given by adding a LE or BE suffix to the name of the Unicode Transformation Format, giving e. g. UTF-16BE or UTF-32LE. We can reverse the order of the bytes—between big and little endian—at high speed: e.g., using one instruction per 64 bytes. For simplicity, we present our results on UTF-8 and UTF-16LE.

2.1 | UTF-16

When the Universal Character Set was initially defined, it was meant to be a 16-bit character set with UTF-16 being its natural encoding, representing each universal character in one 16-bit word. It was later realized that 65 536 code points are insufficient to represent the writing systems of the world's many cultures, especially when having to account for over 50 000 Chinese, Japanese, and Korean ideographs. UCS was therefore extended past the Basic Multilingual Plane to code points up to U+10FFFF and UTF-16 retrofitted with a *surrogate* mechanism to permit representation of these newly added characters.

UTF-16 is a versatile Unicode Transformation Format as it permits (absent surrogates) easy processing of text in many popular languages, while not being as memory-hungry as UTF-32. It is widely used in databases and binary file

[3] Big endian is the prescribed default byte order [5], although it is less common.

type	*range*	*pattern*
ASCII lead byte	`0x00–0x7f`	`0XXX XXXX`
continuation byte	`0x80–0xbf`	`10XX XXXX`
2-byte lead byte	`0xc2–0xdf`	`110X XXXX`
3-byte lead byte	`0xe0–0xef`	`1110 XXXX`
4-byte lead byte	`0xf0–0xf4`	`1111 0XXX`

TABLE 1 Types of UTF-8 bytes with tag bits underlined.

formats and is the preferred internal text representation on Windows NT. Nevertheless, with the advent and growing popularity of universal characters outside of the Basic Multilingual Plane, UTF-16 has been steadily declining in use.

Despite big-endian byte order being prescribed for UTF-16, the little-endian variant UTF-16LE is more commonly encountered under the influence of x86's little-endian orientation. A common convention to deal with this ambiguity is to prefix UTF-16 encoded documents with the *byte order mark* (BOM) `U+FEFF`.[4] Its byte-swapped counterpart `U+FFFE` is a reserved "uncharacter" and should not occur in Unicode text. If a UTF-16 encoded document begins with `U+FFFE`, it can thus be assumed to be in wrong byte order, permitting automatic byte-order detection in many situations. Our algorithms do not make use of this convention and strictly assume UTF-16LE throughout. A BOM is neither generated, nor checked for, nor stripped.

As illustrated in the "UTF-16" column of Fig. 1, code points in the Basic Multilingual Plane are represented as themselves. Code points outside of this plane have `0x10000` subtracted from them (the *surrogate plane shift*), yielding a 20-bit number. This number is split into two 10-bit halves. The high half is tagged with `0xd800`, yielding a *high surrogate*. Likewise, the low half is tagged with `0xdc00`, yielding a *low surrogate*. The character is then encoded by giving its high surrogate, directly followed by its low surrogate. It is for this purpose that code points in the range `0xd800–0xdfff` do not represent universal characters.

Decoding UTF-16 is a matter of joining the bits of surrogate pairs, leaving Basic-Multilingual-Plane characters unchanged. Care must be taken to validate that each high surrogate is succeeded by a low surrogate and vice versa. With this sequencing requirement ensured, all UTF-16 sequences are valid and have a 1:1 mapping to code points.

2.2 | UTF-8

The most popular Unicode Transformation Format is UTF-8, representing each universal character as a sequence of 1–4 bytes. Replacing the earlier UTF-1, the format was designed to be backwards-compatible to ASCII while also being safe for use in UNIX file names, and comes with many other desirable features. Under many circumstances, UTF-8 text can be processed as if it was a conventional ASCII-based 8-bit encoding like those of the ISO-8859 family. This includes common applications like concatenation, substring search, field-splitting (with ASCII characters or UTF-8 strings for separators), and collation, rendering it the most popular UTF.

UTF-8 can be seen as an extension to ASCII, where each ASCII character (`U+00–U+7F`) is represented as itself with other characters being represented by sequences of bytes in the range `0x80–0xf4` (cf. Table 1). Such sequences start

[4] `U+FEFF` only has this function as the first character of a document. In other positions, it should be treated as an ordinary universal character and must not be stripped or altered.

expression	*description*
$\neg a$	bitwise complement of a
$\mathrm{ctz}(a)$	number of trailing zeroes in a
$\mathrm{width}(a)$	number of bits needed to represent a
$\mathrm{popcount}(a)$	number of bits set in a
$\mathrm{pext}(a, b)$	the bits given in a extracted from b
$\mathrm{pdep}(a, b)$	b deposited into the bits given in a
$\mathrm{compress}(m, v)$	vector v compressed by mask m
$a + b$	sum of a and b
$a \ll b$	a logically shifted to the left by b places
$a \gg b$	a logically shifted to the right by b places
$a = b$	mask indicating elements of a equal to those of b
$a \wedge b$	bitwise and of a and b
$a \vee b$	bitwise or of a and b
$a \oplus b$	bitwise exclusive-or of a and b
$a \mathbin{?} b : c$	ternary operator; equal to $a \wedge b \vee \neg a \wedge c$

TABLE 2 Summary of notation

with a *lead byte* (`0xc2`[5]`–0xf4`) indicating the length of the sequence in its *tag bits*, followed by 1–3 *continuation bytes* (`0x80–0xbf`), making the encoding stateless, and self-synchronizing.

The details are summarized in the "UTF-8" column of Fig. 1: The bits of the code point are numbered `A–W` starting at the least significant bit. For each of the four possible cases (the ASCII/1-byte case, the 2-byte case, the 3-byte case, and the 4-byte case[6]), the bits of the code point are copied into the lead and continuation bytes as indicated in the figure. Tag bits are applied (underlined in Fig. 1) to distinguish ASCII, lead, and continuation bytes.

For many universal characters, more than one encoding seems to be possible according to the figure. However, only the shortest possible encoding for each character is permitted to ensure uniqueness of the encoding. While 4-byte sequences could encode code points in excess of `U+10FFFF`, such sequences are not legal either. The bytes `0xc0`, `0xc1`, and `0xf5–0xff` are thus not used by UTF-8.

Decoding UTF-8 begins by looking at the tag bits to tell the start and length of each sequence. Then, the code point is assembled from the payload of these bytes. A critical part in decoding UTF-8 is validation, especially against overly-long sequences and illegal code points (surrogates, code points greater than `10FFFF`). In the algorithm presented in § 6 we demonstrate how decoding UTF-8 with comprehensive validation and then reencoding it into UTF-16 can be implemented efficiently, leveraging AVX-512 instructions.

[5] `0xc0` and `0xc1` would introduce 2-byte sequences corresponding to ASCII characters, which are encoded as single bytes instead.

[6] the 1–3-byte cases represent Basic-Multilingual-Plane characters, the 4-byte case corresponds to characters represented as surrogate pairs in UTF-16.

3 | RELATED WORK

There are relatively few academic publications on Unicode string processing using SIMD instructions. Cameron [7] proposed a UTF-8 to UTF-16 transcoder using SIMD instruction using *bit streams*. A bit stream is a transposition on the character inputs. For example, from 128 bytes of data, we produce eight 128-bit registers with the first register containing the most significant bits of each input byte, and the last register containing the least significant bit of each input byte. The transcoding from UTF-8 to UTF-16 is done in this bit stream form with a final phase where unused bytes are removed. Inoue et al. [8] presented a limited UTF-8 to UTF-16 transcoder which lacked validation and could not handle 4-byte UTF-8 characters. They rely on a 105 KiB lookup table.

Lemire and Muła [2] presented a generic approach that does full UTF-8 to UTF-16 and UTF-16 to UTF-8 transcoding, with validation. Their UTF-8 to UTF-16 transcoding function is similar in principle to the strategy used by Inoue et al. [8] in that they rely on the presence of instructions to quickly permute bytes within a register in an arbitrary manner, based on a lookup table. The accelerated UTF-8 to UTF-16 transcoding algorithm processes up to 12 input UTF-8 bytes at a time. Given the input bytes, it finds beginning of each character, forming a 12-bit word which is used as a key in a 1024-entry table. Each entry in the table contains the number of UTF-8 bytes to consume and an index into another table where we find *shuffle masks*. The tables use about 11 KiB. The shuffle masks are applied to the 12 input bytes to form a vector register that can be transformed efficiently. This 12-byte routine works within 64-byte blocks. The 64-byte blocks are validated using a fast technique [1]. Their UTF-16 to UTF-8 algorithm iteratively reads a block of input bytes in a SIMD register. Depending of the values of 16-bit words, the algorithm uses one of several paths. E.g., if all 16-bit words are in the range `U+0000`–`U+07FF`, the 16-bit words are converted to 32-bit words to ultimately produce 1-byte, 2-byte or 3-byte characters. A series of lookup tables allow the efficient permutations, using a total of 8.5 KiB.

Gatilov [9] produced one of the best and most complete software library for Unicode transcoding (utf8lut). It is similar in spirit to the work of Lemire and Muła [2], but utf8lut requires larger tables: 2 MiB for the UTF-8 to UTF-16 transcoder and 16 KiB for the UTF-16 to UTF-8 transcoder.

Unlike this prior work, our proposals do not require lookup tables. This is possible through the use of novel *compression instructions* introduced with AVX-512VBMI2 (see § 4.3, Tbl. 3), allowing us to move bytes to the right places within registers entirely in hardware, without in-memory tables.

4 | NOTATIONAL CONVENTIONS

In the algorithms described below, all logical symbols refer to bitwise logic. Comparisons are performed between corresponding elements of vectors, yielding a bit mask of those elements for which the comparison holds. All arithmetic operations, shifts, and comparisons are performed on unsigned numbers. The width of the number depends on the vector used.

As a general convention, scalars, vectors of bytes, and masks derived from them are indicated with lowercase letters. Vectors of 16- or 32-bit words are indicated with uppercase letters.[7] The symbol n is number of bytes in a vector; for AVX-512 it is $n = 64$. This convention permits us to explain the algorithms in terms of AVX-512 instructions while giving generic formulæ potentially applicable to other future instruction sets.

[7] The convention attempts to underline that byte vectors correspond to UTF-8 whereas word vectors correspond to UTF-16.

The operator precedence follows C precedence rules with

$$a + b \ll c = d \wedge e \vee f$$

being parsed as

$$((((a + b) \ll c) = d) \wedge e) \vee f.$$

Table 2 gives a list of symbols used in decreasing order of precedence.

4.1 | Mask Operations

Masks are conceptually arrays of bits—containing between 8 and 64 bits—meant to be used in conjunction with vectors having the same number of elements. For example, *byte masks* (noted $m_1, m_{234}, \ldots$) may contain 64 bits if they correspond to vectors of 64 bytes. We also have *word masks* (e.g., M_3) containing 16 bits when they corresponding to 512-bit vectors of 32-bit values. See Appendices A and B for detailed lists of our masks and other variables. We operate on masks as if they were unsigned integer values: $m_{+3} = m_4 \ll 3$ means that the whole mask m_4 is shifted to the left by three places to give m_{+3}; the 64 individual mask bits are always either 0 or 1. The logical operations *or* ($\vee$), *and* ($\wedge$) and *not* ($\neg$) are applied bitwise. We have that $m = 0$ sets all bits to zero whereas $m = \neg 0$ sets all bits to one.

In practice, the processor has several instructions dedicated to AVX-512 mask registers (e. g., `kshiftrd`, `kandq`, `korb`). Mask registers can be converted back and forth to general-purpose registers as needed—with the caveat that the conversion from mask registers to general-purpose registers may have a high latency (e. g., 3 cycles).

4.2 | Vector Operations

When operating on vectors, equations have to be read as "SIMD formulæ" applying element-by-element. For example, we write

$$w = m \,?\, a + b : c$$

to mean "each element of w is set to the sum of the corresponding elements in a and b if the corresponding bit is set in m or to c otherwise." With an explicit index $i = 0 \ldots n - 1$, the previous expression could be written as

$$w[i] = m \wedge 1 \ll i \,?\, a[i] + b[i] : c[i] \quad \text{for } i = 0, 1, \ldots, n - 1.$$

We believe that the presentation as "SIMD formulæ" is easier to understand and prefer it where possible. Explicit indices are only used when permutations are involved. For example, we write

$$w[i] = v\big[p[i]\big]$$

to mean "w is v permuted by the index vector p."

Conversions from one element size to another are not explicitly written out; watch the letter case of the variables used to see when this happens. All such conversions are zero-extensions or truncations.

Remark The conventional binary notation presents the least significant bits last. When working with masks and vectors, these least significant bits correspond to the first elements of the vectors. This discrepancy in the order is a source of confusion, but it is difficult to avoid. Intel intrinsic functions reflect this confusion by providing two sets of functions to create new vectors: `_mm512_set_*` and `_mm512_rset_*` depending on the prefered order [10].

4.3 | Special Functions

We use several special bit-manipulation functions corresponding to instructions available on contemporary x86 computers:

ctz The *count trailing zeroes* operation $\mathrm{ctz}(a)$ counts the number of trailing (least significant) zero bits in a, i. e. how often a can be divided by 2 until leaving an odd number. It corresponds to the `bsf`/`tzcnt` instructions of the x86 instruction set. Our algorithms never invoke $\mathrm{ctz}(0)$.

width The *bit width* operation $\mathrm{width}(a)$ counts the number of bits needed to represent a. It is

$$\mathrm{width}(a) = (a \neq 0)\ ?\ \lfloor \log_2 a \rfloor + 1 : 0. \tag{1}$$

This operation is efficiently implemented on many architectures through the *count leading zeroes* operation (x86 instruction `bsr`/`lzcnt`). Our algorithms never invoke $\mathrm{width}(0)$.

popcount The *population count* operation $\mathrm{popcount}(a)$ computes the number of bits set in a. This can also be understood as the sum of the bits of a. It corresponds to the `popcnt` instruction of the x86 instruction set.

pext The *parallel extract* operation $\mathrm{pext}(a, b)$ takes a bit mask a indicating a possibly non-consecutive bit field and extracts those bits from b, packing them into $\mathrm{popcount}(a)$ bits. This corresponds to the `pext` instruction on recent x86 processors. The operation is perhaps best understood with a diagram:

$$\begin{aligned} a &\quad \texttt{1010111011000100} \\ b &\quad \texttt{1000101011110001} \\ \text{bit field} &\quad \texttt{1-0-101-11---0--} \\ \mathrm{pext}(a, b) &\quad \texttt{00000000}\underline{\texttt{10101110}} \end{aligned} \tag{2}$$

pdep The *parallel deposit* operation $\mathrm{pdep}(a, b)$ takes a bit mask a indicating a possibly non-consecutive bit field and deposits the bits from b into this field. It performs the opposite operation to pext and corresponds to the `pdep` instruction on recent x86 processors. We can likewise visualize its operation through a diagram:

$$\begin{aligned} a &\quad \texttt{1010111011000100} \\ b &\quad \texttt{10110100}\underline{\texttt{10101110}} \\ \text{bit field} &\quad \texttt{1-0-101-11---0--} \\ \mathrm{pdep}(a, b) &\quad \texttt{1000101011000000} \end{aligned} \tag{3}$$

compress The *compress vector* operation $\mathrm{compress}(m, v)$ is the only vector operation among our special functions. It performs the same operation as the parallel extract operation pext, but instead of extracting bits from a bit field, it extracts elements from a vector. This corresponds to the `vpcompressb` instruction on recent x86 processors. For the visualization, we have given the mask $m = \texttt{0xcd}$ with the least significant bit on the left to make the operation easier to see. The least significant mask bit decides whether to keep the first vector element and so on until the

most significant mask bit decides whether to keep the last vector element:

$$\begin{array}{rl} m & \texttt{1 0 1 1 0 0 1 1} \\ v & \texttt{12 34 56 78 9a bc de f0} \\ \text{kept elements} & \texttt{12 -- 56 78 -- -- de f0} \\ \text{compress}(m, v) & \underline{\texttt{12 56 78 de f0}}\,\texttt{00 00 00} \end{array} \tag{4}$$

Observe how we reversed the bit order of the mask m to match the natural vector order: its usual binary representation is `11001101`.

5 | AVX-512

Our algorithms are based on the AVX-512 family of instruction-set extensions to the Intel 64[8] instruction-set architecture [11]. An extension to the AVX family of instruction-set extensions, AVX-512 provides a comprehensive set of SIMD instructions for operation on vectors of 16, 32, or 64 bytes organized into bytes or words of 16, 32, or 64 bits. A register file of 32 *vector registers* `zmm0–zmm31` complemented by 8 *mask registers* `k0–k7` is provided.

AVX-512 instructions are generally non-destructive, writing their output into a separate operand from their inputs. In most AVX-512 instructions, one operand is permitted to be a memory operand with the remaining operands being register or immediate operands. This is usually the first input operand, but for some instructions it may also be the output operand.

The AVX-512 instruction set is split into a set of extensions. Each extension adds new instructions to the Intel 64 architecture, enhancing the capabilities of AVX-512. Depending on the microarchitecture used, not all AVX-512 extensions might be available. Table 3 gives a list of AVX-512 instructions used and the extension they hail from. In the following, we list those AVX-512 extensions needed to execute the algorithms described in this paper:

AVX-512F The *foundation* extension implements the basic AVX-512 instruction set on 64-byte vectors. Every AVX-512 implementation must support AVX-512F.

AVX-512BW The *byte/word* extension extends the AVX-512F instructions to vectors of bytes and 16-bit words.

AVX-512DQ The *dword/qword* extension provides additional instructions on 32- and 64-bit words.

AVX-512VBMI The *vector byte manipulation instructions* extension adds instructions to permute and manipulate bytes.

AVX-512VBMI2 The *vector byte manipulation instructions 2* extension adds compress/expand support and double-width shifts for bytes and 16-bit words.

The first generation of Intel 64 processors supporting all required AVX-512 extensions are those code named *Icelake*, based on the microarchitecture code named *Sunny Cove*. By emulating `vpcompressb` through other instructions, it is likely possible to adapt the algorithms to processors as early as the generation code named *Cannon Lake*, albeit at significant reduction in performance.

[8] The 64 bit variant of the x86 (IA-32) instruction-set architecture, also known as AMD64, x86-64, EM64T, and IA-32e.

instruction	*extension*	*description*
`vmovdqu8/16`	BW	move byte/word/dword vector
`vpblendmw/d`	BW/F	blend words/dwords with mask
`vpbroadcastd/q`	F	broadcast dword/qword to vector
`vextracti32x8`	DQ	extract 256-byte word from vector
`vpmovzxbw/wd`	BW/F	zero-extend byte to word or word to dword
`vpaddb/w/d`	BW	add bytes/words/dwords
`vpsubb/w/d`	BW	subtract bytes/words/dwords
`vpcmpub/w`	BW	compare unsigned bytes/words
`vpternlogd`	F	logic on 3 operands by given truth table
`vpandd`	F	bitwise and dwords
`vpandnd`	F	bitwise and-not dwords
`vpsllw/d`	BW/F	logically shift words/dwords left by immediate
`vpsrlw/d`	BW/F	logically shift words/dwords right by immediate
`valignd`	F	right-shift elements between operands
`vpmultishiftqb`	VBMI	shift bytes within qword, see § 7.3
`vpcompressb`	VBMI2	compress byte vector, see § 4.3
`vpermb`	VBMI	permute byte vector by byte index vector
`kmovd/q`	BW	move 32/64-bit mask
`kord/q`	BW	bitwise or 32/64-bit mask
`kandnd/q`	BW	bitwise and-not 32/64-bit mask
`knotd/q`	BW	bitwise complement 32/64-bit mask
`kshiftrd/q`	BW	logically shift 32/64-bit mask right by imm.
`ktestd/q`	BW	test bitwise and/and-not of masks for all-zero
`kortestd/q`	BW	test bitwise or of masks for all-zero/all-one

TABLE 3 Selected AVX-512 instructions.

5.1 | Masking

The output of most vector instructions is subject to *masking*, a novel feature of AVX-512. A mask register `k1`–`k7`[9] is applied to the output operand, specifying either *merge masking* or *zero masking*. With merge masking, only those vector elements indicated by bits set in the mask register are modified in the output operand. The other vector elements remain unchanged. With zero masking, vector elements for which the bits in the mask register are clear are zeroed out.

[9] Mask register `k0` cannot be used for masking, but remains available for logic on masks.

For example, the merge and zero masking instructions

```
vpaddb zmm0{k1}, zmm2, zmm3      (merge masking), and
vpaddb zmm4{k5}{z}, zmm6, zmm7   (zero masking)
```

perform a **p**acked **add**ition of **b**ytes, giving

$$\texttt{zmm0} = \texttt{k1} \mathbin{?} \texttt{zmm2} + \texttt{zmm3} : \texttt{zmm0} \quad \text{and}$$
$$\texttt{zmm4} = \texttt{k5} \mathbin{?} \texttt{zmm6} + \texttt{zmm7} : 0.$$

Masking on register operands is free for most instructions, though merge masking introduces an input dependency on the old value of the output operand.

Masking on memory operands enables *memory fault suppression* for most instructions. This means that the CPU does not signal memory faults for masked-out vector elements, permitting masked out elements to extend into unmapped or non-writable pages. This suppression affects both input and output memory operands.

5.2 | Microarchitectural Details

To simplify the implementation of AVX-512 on microarchitectures designed to execute the older SSE and AVX families of instruction-set extensions, most SIMD instructions operate within *lanes* of 16 bytes. That is, in many ways, it is as if the 64-byte vector registers were made of four nearly independent 16-byte subregisters. Instructions that process data across lanes (such as `vpermb` or `vpcompressb`) exist, but can typically execute on less execution units and take longer to execute in comparison to instructions that do not. We thus want to avoid cross-lane operations if feasible.

On current Intel microarchitectures including Sunny Cove (Icelake), Cypress Cove (Rocket Lake), and Willow Cove (Tiger Lake), most AVX-512 instructions[10] can execute on *execution ports* 0, 1, and 5. Instructions that do not cross lanes usually execute in a single cycle, instructions that do take 3 or more cycles. Some instructions are restricted in the ports they can execute on: shifts can only execute on ports 0/1, permutations and other cross-lane instructions, as well as comparisons into masks can only execute on port 5. Instructions operating on masks (i. e. those whose mnemonics start with `k`) are restricted to one of ports 0 or port 5, depending on the instruction [12, 13].

In addition to these restrictions, ports 0 and 1 support a vector length of only 32 bytes while port 5 supports the whole 64 bytes. Instructions operating on a vector length of 64 bytes are executed either on port 5 or on ports 0/1 joined together, occupying both ports for one cycle simultaneously. Thus, there are effectively only two ports available to execute instructions with a 64-byte vector length. While 32-byte vectors are processed at 3 vectors of 32 bytes (i. e. 6 lanes) per cycle, 64-byte vectors are processed at only 2 vectors of 64 bytes (or 8 lanes) per cycle, leading to a theoretical speedup by a factor of $4/3$ or 33 % of 64-byte vectors over 32-byte vectors for an otherwise identical algorithm. This stands in contrast to the factor 2 or 100 % speedup one would naïvely expect from doubling the vector length.

It is vital for the performance of AVX-512 code to keep track of which ports instructions execute on, rearranging or editing the code such that both port 0/1 and port 5 can execute instructions at the same time [14]. Through the use of microarchitectural simulation [15] in the design of the algorithms, good port utilization has been ensured.

[10] assuming no memory operands

6 | TRANSCODING FROM UTF-8 TO UTF-16

We transcode UTF-8 to UTF-16 by gathering the bytes that make up each character from the last byte of each character to its first byte. This exploits the similarity in bit arrangement between the four cases (ASCII, 2-byte, 3-byte, and 4-byte) highlighted in Fig. 1a. The bytes of each UTF-8 sequence are isolated from the input string, liberated of their tag bits, shifted into position, and finally summed up into a code point.

Using the exact correspondence between 4-byte UTF-8 characters and characters represented as surrogate pairs in UTF-16, we treat 4-byte characters as an overlapping pair of a 3-byte sequences and a 2-byte sequence that is later fixed up into a high and a low surrogate. This saves us extra code for extracting the fourth-last byte of each sequence and avoids the costly use of 32-bit words for intermediate results.

To illustrate this idea, consider the following example, translating the Unicode characters `U+40` (@), `U+A7` (§), `U+2208` (∈), and `U+1D4AA` (𝒪) from UTF-8 to UTF-16:

```
 @    §       ∈          𝒪
40 C2 A7 E2 88 88 F0 9D 92 AA
-----------------------------
40
   C2 A7
         E2 88 88
                  F0 9D 92
                        92 AA
-----------------------------
 0040  00A7  2208  D835  DCAA
```

(5)

These four characters demonstrate the behavior of the algorithm on the four UTF-8 cases, representing ASCII, 2-byte, 3-byte, and 4-byte respectively. Observe especially how the code sequence `F0 9D 92 AA` for 𝒪 is split into two overlapping sequences `F0 9D 92` and `92 AA`. The first of these two is translated into the high surrogate `D835` with the second one becoming the low surrogate `DCAA`.

The algorithm can be roughly described with the following *plan of attack*:

1. Read a vector of 64 bytes.
2. Classify each byte according to whether it is an ASCII byte, continuation byte, 2-byte lead byte, 3-byte lead byte, or 4-byte lead byte.
3. Construct a mask indicating the last byte of each UTF-8 sequence. For 4-byte characters, the third byte is indicated, too, treating them as a 3-byte sequence for the high surrogate and a 2-byte sequence for the low surrogate.
4. Use the mask to gather the last, 2nd last, and 3rd last byte of each sequence.
5. Strip tag bits, shift bits into place and or them into UTF-16 words.
6. Postprocess surrogates by shifting their bits into place, and applying tag bits and surrogate plane shift.
7. Write the resulting bytes to the output, incrementing the input and output pointers by the number of bytes consumed/generated.
8. Repeat until the end of input or an encoding error are encountered.

Apart from this general plan, there are also fast paths for the cases (a) ASCII characters only, (b) ASCII, and 2-byte sequences only, and (c) 1–3-byte sequences only.

Validation is performed throughout the transcoding process, as explained in § 6.4. In comparison to previous algorithms, it is simplified by advancing the input only by complete UTF-8 sequences; if the input is correct UTF-8,

each vector of input thus begins with a complete sequence.

6.1 | Classification and Masks

After reading a vector of bytes from the input buffer, the characters in it are classified according to the range they fall into. Various masks are then built from this classification. In the following explanations, we follow the convention from § 4 where names of the form $m_{\ldots}$ refer to masks about the input vector w_{in} while names of the form $M_{\ldots}$ refer to masks about the output vector.

These two kinds of masks are connected through the pext and pdep operations, relating the end bytes of the decoded UTF-8 sequences to the UTF-16 words they correspond to and vice versa.

The first set of masks is derived directly from w_{in}, classifying the input into ASCII

$$m_1 = (w_{\text{in}} < \texttt{0x80}), \tag{6}$$

2/3/4-byte sequence lead bytes

$$m_{234} = (\texttt{0xc0} \leq w_{\text{in}}), \tag{7}$$

3/4-byte sequence lead bytes

$$m_{34} = (\texttt{0xe0} \leq w_{\text{in}}) \tag{8}$$

and 4-byte sequence lead bytes

$$m_4 = (\texttt{0xf0} \leq w_{\text{in}}). \tag{9}$$

From these we then derive a mask

$$m_{1234} = m_1 \vee m_{234} \tag{10}$$

indicating the presence of any kind of lead byte. All other bytes ($\neg m_{1234}$) are continuation bytes.

Then we construct the important mask m_{end} identifying the last bytes of each sequence to be decoded. These are the last bytes of each UTF-8 sequence as well as the third byte of each 4-byte sequence. Working backwards from these last bytes, we later use this mask to gather the last, second-last and third-last bytes of each sequence.

The key insight in constructing m_{end} is that as each UTF-8 sequence is followed by another UTF-8 sequence, we can find the positions of the last bytes as those preceding the lead bytes of the next sequence ($m_{1234} \gg 1$). The third byte of each 4-byte sequence is added by first computing the fourth byte of each sequence

$$m_{+3} = m_4 \ll 3 \tag{11}$$

and then shifting the result to the right to obtain the last third bytes

$$m_{\text{end}} = (m_{+3} \vee m_{1234}) \gg 1 \vee m_{+3}. \tag{12}$$

An unfortunate consequence of defining m_{end} by going backwards from the lead bytes of the next characters is that we only catch the last character of the vector when it is followed by an incomplete character whose lead byte we can shift to the right. For 4-byte sequences right at the end of the vector, this leads to us only detecting the third last byte in the character. Oring in m_{+3} at the end fixes this problem for the 4-byte case.

For the other cases, the only effect of this process is that if w_{in} does not end in a partial character, decodes to no more than 32 words of UTF-16, and the last character is not a 4-byte sequence, we process one less character in the current iteration than possible. However, the minor performance impact of hitting this edge case is more than outweighed by not spending extra time computing the mask correctly.[11]

To visualize the various masks, consider the strings "x∇𝔓" and "$\varepsilon \le \pm 1$" with a vector length of $n = 8$ bytes:

	x	∇			𝔓				ε		≤			±		1
w_{in}	78	e2	88	87	f0	9d	94	93	ce	b5	e2	89	a4	c2	b1	31
$m_1 = w_{\text{in}} < \texttt{0x80}$	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
$m_{234} = \texttt{0xc0} \le w_{\text{in}}$	0	1	0	0	1	0	0	0	1	0	1	0	0	1	0	0
$m_{34} = \texttt{0xe0} \le w_{\text{in}}$	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0
$m_4 = \texttt{0xf0} \le w_{\text{in}}$	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
$m_{1234} = m_1 \vee m_{234}$	1	1	0	0	1	0	0	0	1	0	1	0	0	1	0	1
$m_{+3} = m_4 \ll 3$	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
$m_{\text{end}} = (m_{+3} \vee m_{1234}) \gg 1 \vee m_{+3}$	1	0	0	1	0	0	1	1	0	1	0	0	1	0	1	0

(13)

In this example, masks are arrays of eight bits corresponding to eight-byte sequences: in our actual implementation, we use 64-bit masks. Note in particular how m_{end} accounts for the last character in the left string (being a 4-byte character), but not in the right string, where it is an ASCII character. Also note how the character 𝔓 has two end bits, being treated as a 3-byte sequence overlapping a 2-byte sequence.

6.2 | Assembling Characters

With these masks in hand, we can strip off the tag bits and assemble characters. The UTF-8 tag bits are stripped off by clearing the most significant two bit of each non-ASCII byte in w_{in}, giving

$$w_{\text{stripped}} = m_1 \; ? \; w_{\text{in}} : w_{\text{in}} \wedge \texttt{0x3f}. \tag{14}$$

The tag bits of 3/4-byte lead bytes are not completely removed by this step; this is sufficient for our purposes as these tag bits get shifted out later on.

Characters are assembled by selecting from w_{stripped} the last (W_{end}), second-last (W_{-1}) and third-last bytes (W_{-2}) of each sequence, zero-extending them to 16 bits and joining their bits into a UTF-16 word. We do this by first preparing a permutation vector P that holds for each word in the output vector, the index of the last byte of the

[11] If a perfect mask is desired, you can instead use

$$m'_{\text{end}} = (m_1 \vee m_2 \ll 1 \vee m_{34} \ll 2 \vee m_4 \ll 3) \wedge \neg(m_4 \ll 2 \wedge 1 \ll (n-1)),$$

where $m_2 = m_{234} \wedge \neg m_{34}$ indicates 2-byte sequence lead bytes. The first byte of each sequence is shifted to the position of its last byte (and the third byte of a 4-byte sequence). The mask is then post-processed by clearing the third byte of a 4-byte sequence starting in the third-last byte of w_{in}, as only complete sequences can be processed.

corresponding sequence. This vector is prepared by compressing (`vpcompressb`) a byte vector holding an identity permutation $(0, 1, \ldots, 63)$ subject to m_{end}. The compressed vector is then zero-extended (`vpmovzxbw`) to 16-bit words, keeping its first $n/2$ elements:

$$P = \text{compress}\big(m_{\text{end}}, (0, 1, \ldots, n-1)\big). \tag{15}$$

We only generate one vector of UTF-16 words per iteration representing at most 32 characters. When the input contains ASCII characters, it might be possible for m_{end} to contain more than 32 set bits. Bits set in m_{end} past the 32nd bit are discarded during the processing: P contains only $n/2$ (or 32) elements. With P in hand, we can load the last byte of each sequence

$$W_{\text{end}}[i] = w_{\text{stripped}}\big[P[i]\big] \tag{16}$$

with a single permutation instruction (`vpermb`).[12]

By decrementing the entries of P, we produce index vectors corresponding to the second-last and third-last bytes of each sequence. To avoid loading the third-last byte of a 1/2-byte sequence or the second-last byte of an ASCII sequence, we mask w_{stripped} with masks

$$m_{-1} = \neg m_1 \gg 1 \quad \text{and} \tag{17}$$

$$m_{-2} = m_{34} \wedge \neg 0 \gg 2 \tag{18}$$

to clear out bytes before ASCII characters resp. those that do not start a 3/4-byte sequence, accounting for possible wrap around.[13] We then obtain our vectors

$$W_{-1}[i] = \big(m_{-1}\ ?\ w_{\text{stripped}} : 0\big)\big[P[i] - 1\big] \quad \text{and} \tag{19}$$

$$W_{-2}[i] = \big(m_{-2}\ ?\ w_{\text{stripped}} : 0\big)\big[P[i] - 2\big] \tag{20}$$

as desired. The last, second-last, and third-last bytes are shifted into place and ored such that the bits `A`–`W` are contiguous, giving

$$W_{\text{sum}} = W_{-2} \ll 12 \vee W_{-1} \ll 6 \vee W_{\text{end}}. \tag{21}$$

[12] as `vpermb` permutes each byte, we zero-mask its result with `0x5555555555555555` to only permute into the less significant byte of each 16-bit word, zero-extending for free.

[13] $P[i] - 1$ and $P[i] - 2$ may yield negative numbers; we assume that in a permutation, such indices either wrap around to the end of the vector or produce 0 as an output. If negative permutation indices yield zeroes, the term $\neg 0 \gg 2$, serving as wraparound protection, can be omitted from m_{-2}.

The value of W_{sum} depending on the case taken can be visualized as follows:

case	*byte sequence*	W_{sum}
ASCII	0GFE DCBA	0000 0000 0GFE DCBA
2 byte	110L KJHG 10FE DCBA	0000 0LKJ HGFE DCBA
3 byte	1110 RQPN 10ML KJHG 10FE DCBA	RQPN MLKJ HGFE DCBA
hi surr	1111 0WVU 10TS RQPN 10ML KJHG	0WVU TSRQ PNML KJHG
lo surr	10ML KJHG 10FE DCBA	0000 MLKJ HGFE DCBA

(22)

This representation is close to UTF-16LE format with only the surrogate cases diverging. To address this difference, we first identify the locations of surrogates in W_{out}. Sequences in w_{in} corresponding to low surrogates end at the fourth bytes of 4-byte sequences. By extracting the locations of these through m_{end} into the space of W_{out}, we obtain the locations of low surrogates

$$M_{\text{lo}} = \text{pext}(m_{\text{end}}, m_{+3}) \tag{23}$$

in W_{out}. High surrogates

$$M_{\text{hi}} = M_{\text{lo}} \gg 1 \tag{24}$$

always precede low surrogates.

Surrogates are fixed up by shifting high surrogates into position and applying surrogate plane shift and tag bits,[14] giving

$$W_{\text{out}} = \begin{cases} (W_{\text{sum}} \gg 4) + \texttt{0xd7c0} & \text{if } M_{\text{hi}} \\ W_{\text{sum}} \vee \texttt{0xdc00} & \text{if } M_{\text{lo}} \\ W_{\text{sum}} & \text{otherwise.} \end{cases} \tag{25}$$

The operation of Eq. 25 can be visualized as follows, where $\texttt{0vuts} = \texttt{WVUTS} - 1$:

case	W_{sum}	W_{out}
high surrogate	0WVU TSRQ PNML KJHG	1101 10vu tsRQ PNML
low surrogate	0000 MLKJ HGFE DCBA	1101 11KJ HGFE DCBA
other	RQPN MLKJ HGFE DCBA	RQPN MLKJ HGFE DCBA

(26)

For illustration purposes, we provide C code implementing equation Eq. 25 using Intel intrinsic functions: see Fig. 2.

The vector W_{out} holds the UTF-16LE encoded characters we want to write out. There is a final issue: the 64 bytes of UTF-8 data in the input may correspond to anywhere from 21 to 64 words of output, of which the first up to 32 words are processed.[15] If a surrogate pair happened to straddle the end of W_{out}, we would discard the corresponding low surrogate and produce an incorrect result. So once again, special care must be taken to omit the 32nd word of

[14] Adding $\texttt{0xd7c0} = \texttt{0xd800} - \texttt{0x0020}$ applies the tag bits and the surrogate plane shift in one step.

[15] Input data corresponding to the remaining words (if any) is reprocessed in the next iteration.

```
__m512i mask_d7c0d7c0 = _mm512_set1_epi32(0xd7c0d7c0);
__m512i mask_dc00dc00 = _mm512_set1_epi32(0xdc00dc00);
//...
// Mlo, Mhi and Wsum have been computed, we compute Wout.
__m512i lo_surr_mask = _mm512_maskz_mov_epi16(Mlo, mask_dc00dc00);
__m512i shifted4_Wsum = _mm512_srli_epi16(Wsum, 4);
__m512i tagged_lo_surrogates = _mm512_or_si512(Wsum, lo_surr_mask);
__m512i Wout = _mm512_mask_add_epi16(tagged_lo_surrogates, Mhi,
               shifted4_Wsum, mask_d7c0d7c0);
```

FIGURE 2 C code using Intel intrinsic functions equivalent to Eq. 25.

output if it is a high surrogate. We do so by computing a mask

$$M_{\text{out}} = \neg\big(M_{\text{hi}} \wedge 1 \ll (n/2 - 1)\big) \tag{27}$$

of the elements of W_{out} excluding the last element if it happens to be a high surrogate. We introduce a variable b which is set to all ones ($b = \neg 0$) except at the end of the input (cf. § 6.3). By depositing the mask M_{out} into the last bytes of each sequence, we obtain a mask

$$m_{\text{processed}} = \mathsf{pdep}(b \wedge m_{\text{end}}, M_{\text{out}}) \tag{28}$$

holding the locations of the last byte of each sequence that has been processed into a word in W_{out}.

With this mask, we can compute the number of bytes of input processed

$$n_{\text{in}} = \mathsf{width}(m_{\text{processed}}) \tag{29}$$

and the number of words of output produced

$$n_{\text{out}} = \mathsf{popcount}(m_{\text{processed}}). \tag{30}$$

The first n_{out} bytes of the output vector are then deposited into the output buffer, input and output buffers are advanced by n_{in} and n_{out} and we continue with the next iteration.

To visualize the generation of $m_{\text{processed}}$, consider the example string "±1=O" with a vector length of $n = 8$ bytes:

$$\begin{array}{rcccccccc} & \pm & 1 & = & & & & O \\ w_{\text{in}} & \texttt{C2} & \texttt{B1} & \texttt{31} & \texttt{3D} & \texttt{F0} & \texttt{9D} & \texttt{92} & \texttt{AA} \\ m_{\text{end}} & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \end{array} \tag{31}$$

As the character `D835 DCAA` straddles the end of the vector, it cannot be processed in the current iteration:

$$\begin{array}{rcccc} & \pm & 1 & = & O \\ W_{\text{out}} & \texttt{00B1} & \texttt{0031} & \texttt{003D} & \texttt{D835 (DCAA)} \\ M_{\text{hi}} & 0 & 0 & 0 & 1 \\ 1 \ll (n/2-1) & 0 & 0 & 0 & 1 \\ M_{\text{out}} & 1 & 1 & 1 & 0 \end{array} \tag{32}$$

Depositing the bits of M_{out} through m_{end}, we then obtain

$$\begin{array}{rcccccccc} m_{\text{processed}} & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ \text{bytes processed} & \texttt{C2} & \texttt{B1} & \texttt{31} & \texttt{3D} & \texttt{--} & \texttt{--} & \texttt{--} & \texttt{--} \\ \text{words produced} & \multicolumn{4}{c}{\texttt{00B1}} & \multicolumn{2}{c}{\texttt{0031}} & \multicolumn{2}{c}{\texttt{003D}} \end{array} \tag{33}$$

and advance buffers by $n_{\text{in}} = 4$ bytes and $n_{\text{out}} = 3$ words respectively. The bytes corresponding to O will be processed again in the next iteration.

6.3 | Processing the Tail

The final bit of input with less than 64 characters remaining (tail) is handled through the variable b. This variable holds a mask of those bytes in w_{in} we are permitted to process. Initially we set $b = \neg 0$, permitting all bytes to be processed. When the end of the input with $\ell < n$ bytes remaining to be processed is reached, we set b to a mask of the first ℓ bytes of w_{in}, giving

$$b = (1 \ll \ell) - 1. \tag{34}$$

The tail of input is read zero-masked by b, padding it with NUL bytes. Then, a final iteration of the main loop is performed, processing only the bytes accounted for in b.

6.4 | Input Validation

Throughout the transcoding process, we check the input for encoding errors and abort transcoding if any such error occurs. Aborting is done by determining the location of the encoding error and setting the remaining input length ℓ to the number of bytes preceding the first error. We then clear all input bytes starting at the first erroneous byte and jump to the tail-handling code from § 6.3, effectively restarting the current iteration as "final" iteration.

Having talked about how to continue after an error has occurred, we shall now direct our attention to the kinds of errors we have to check for. A UTF-8 encoded document must conform to the following rules:

1. Bytes `0xf5–0xff` must not occur.
2. Lead and continuation bytes must match: each byte in `0xc0` to `0xdf` must be followed by one continuation byte, each byte from `0xe0` to `0xef` by two continuation bytes and each byte from `0xf0` to `0xf4` by three continuation bytes.
3. Continuation bytes may not otherwise occur.
4. The decoded character must be larger than `U+7F` for 2-byte sequences, larger than `U+7FF` for 3-byte sequences, and larger than `U+FFFF` for 4-byte sequences.
5. The character must be no greater than `U+10FFFF`.
6. The character must not be in the range `U+D800–U+DFFF`.

We check for these rules throughout the algorithm, mostly reusing masks we already have to compute for other steps of the code. Three checks are performed in total:

Overlong 2-byte sequences

Right at the beginning, we check whether any of the bytes `0xc0` or `0xc1` occur. Presence of these bytes indicates a 2-byte sequence that encodes a code point below `U+80`, violating condition 4. The first invalid input byte is the first `0xc0` or `0xc1` byte found:

$$\text{valid if} \quad \big(m_{234} \wedge (w_{\text{in}} < \texttt{0xc2})\big) = 0. \tag{35}$$

Mismatched continuation bytes

After computing the various classification masks, we check if conditions 2 and 3 hold. As each byte of UTF-8 is either a lead or continuation byte, we check this by computing where continuation bytes should be (m_c) and comparing this with where lead bytes are not:

$$\text{valid if} \quad m_c = \neg m_{1234}. \tag{36}$$

We compute m_c from the location of the second (m_{+1}), third (m_{+2}), and fourth byte (m_{+3}, see Eq. 11) of each sequence:

$$m_{+1} = m_{234} \ll 1, \tag{37}$$

$$m_{+2} = m_{34} \ll 2, \tag{38}$$

$$m_c = m_{+1} \vee m_{+2} \vee m_{+3}. \tag{39}$$

Conveniently, this check also fails on the input if it starts with continuation bytes, violating the invariant established earlier. We do not catch a UTF-8 sequence straddling the end of the vector; such a sequence is checked properly in the next iteration once additional bytes have been fed in.

If this check fails, we must distinguish two cases to determine the location of the first encoding error: If the first mismatch of m_c and m_{1234} is due to a continuation byte present where there should not be one, the first invalid byte is that byte, giving

$$\ell = \text{ctz}(m_c \oplus \neg m_{1234}). \tag{40}$$

Otherwise a continuation byte is missing where there should be one and the corresponding lead byte is the first invalid byte. This byte can be found by masking m_{1234} to all bits preceding the mismatch

$$m_{\text{pre}} = \big(1 \ll \text{ctz}(m_c \oplus \neg m_{1234})\big) - 1 \tag{41}$$

and then finding the last (most significant) bit in it, corresponding to the lead byte that is missing a continuation byte. This gives

$$\ell = \text{width}(m_{1234} \wedge m_{\text{pre}}) - 1. \tag{42}$$

Encodings out of range

Finally, we check if the codepoints encoded by 3- and 4-byte sequences are in range (conditions 4 and 5) and that 3-byte sequences do not encode surrogates (condition 6). The algorithm treats input bytes in the range `0xf5–0xff` as lead bytes of 4-byte sequences. Such sequences encode code points well in excess of `U+110000`, allowing us to verify

condition 1 as a side effect with no extra code.

We augment our existing mask set with a mask

$$m_3 = m_{34} \wedge \neg m_4 \tag{43}$$

indicating the location of 3-byte sequence start bytes in w_{in}. Shifting the mask to indicate the last byte of each 3-byte sequence, extracting through m_{end}, and truncating to $n/2$ bits, we obtain a mask

$$M_3 = \text{pext}(m_{\text{end}}, m_3 \ll 2) \tag{44}$$

indicating which words in W_{out} correspond to 3-byte sequences. We then use M_3 to check if any 3-byte sequences encode codepoints below `U+800`,

$$M_{<\texttt{U+800}} = M_3 \wedge (W_{\text{out}} < \texttt{0x800}) \tag{45}$$

indicating violations of condition 4.

Then we check for surrogates: words in M_3 must not encode surrogates, words in M_{hi} must encode high surrogates (condition 6).[16] A word in M_{hi} produces a high surrogate if and only if the code point it encodes is in range `U+10000–U+10FFFF` (conditions 1, 4, and 5). The masks

$$\begin{aligned} M_{3s} &= M_3 \wedge (\texttt{0xd800} \le W_{\text{out}} < \texttt{0xe000}) \\ &= M_3 \wedge (W_{\text{out}} - \texttt{0xd800} < \texttt{0x0800}) \end{aligned} \tag{46}$$

and

$$\begin{aligned} M_{4s} &= M_{\text{hi}} \wedge \neg(\texttt{0xd800} \le W_{\text{out}} < \texttt{0xdc00}) \\ &= M_{\text{hi}} \wedge (W_{\text{out}} - \texttt{0xd800} \ge \texttt{0x0400}) \end{aligned} \tag{47}$$

indicate violations of these conditions.[17] The check succeeds if no offending words are found:

$$\text{valid if} \quad M_{<\texttt{U+800}} \vee M_{3s} \vee M_{4s} = 0. \tag{48}$$

If an offending word is found, the first invalid byte is the start byte of the corresponding sequence. As the error can never occur in a low surrogate, we can find its location by projecting its location back onto the locations of the first and fourth bytes of every sequence:

$$\ell = \text{ctz}(\text{pdep}(m_{+3} \vee m_{1234},\ M_{<\texttt{U+800}} \vee M_{3s} \vee M_{4s})). \tag{49}$$

[16] by construction, words produced from 1- and 2-byte sequences never produce surrogates and M_{lo} always produces low surrogates; we do not need to validate these.

[17] As all comparisons are unsigned (`vpcmpltuw`), one comparison for each range check suffices.

6.5 | Fast Paths

Three fast paths are provided, speeding up common cases. The first two are programmed such that they cannot be triggered in the "final" iterations for the tail or in case of an encoding error, allowing us to omit the handling of b in their length computations for a further performance increase.

ASCII only

If the first 32 bytes of input are all ASCII bytes, we process these by zero-extension (`vpmovzxbw`) of the first 32 bytes to 16-bit words. The number of processed bytes is always 32, the number of words written out always 32, shortening the dependency chain to the next iteration. No validation is needed in this case as ASCII bytes are always valid.

Only the first 32 bytes are considered before embarking on the fast path as the default path does not process more than 32 characters in any case. Hence, while checking for all 64 bytes to be ASCII would allow for slightly faster processing in the all-ASCII case, performance for documents with short runs of ASCII characters amidst other characters (e. g. HTML documents) suffers significantly, outweighing the benefits of the other case.

1/2 byte only

In the absence of 3- and 4-byte sequences ($m_{34} = 0$), we employ a simplified variant of the algorithm. While following the same operating principles as the main algorithm, we can take some shortcuts in the proven absence of 3- and 4-byte sequences. First, the computation of some masks is greatly simplified, with most masks being entirely irrelevant for this path:

$$m_2 = m_{234}, \tag{50}$$

$$m_{\text{end}} = \neg m_2, \quad \text{and} \tag{51}$$

$$M_{\text{out}} = \text{pdep}\big(m_{\text{end}}, (1 \ll n/2) - 1\big). \tag{52}$$

We then employ a simplified scheme to compute W_{out}: Instead of masking out tag bits, we subtract `0xc2` from the lead byte of each two-byte sequence to cancel out the tag bits of both lead and continuation byte, giving

$$w_{-\texttt{0xc2}} = m_1 \ ?\ 0 : w_{\text{in}} - \texttt{0xc2}. \tag{53}$$

Instead of first building a permutation vector P and then using it to permute the input bytes into place, we directly compress the bytes into position (`vpcompressb`) and then zero extend to 16-bit words (`vpmovzxbw`), giving

$$W_{\text{end}} = \text{compress}(m_{\text{end}}, w_{\text{in}}) \quad \text{and} \tag{54}$$

$$W_{-1} = \text{compress}(m_{1234}, w_{-\texttt{0xc2}}). \tag{55}$$

Vectors W_{end} and W_{-1} must be merged by addition instead of bitwise or to correctly cancel out tag bits, giving

$$W_{\text{out}} = (W_{-1} \ll 6) + W_{\text{end}}. \tag{56}$$

The operation on 2-byte characters can be visualized as follows; `0xc2` is subtracted separately to illustrate the idea:

$$\begin{array}{lll} & W_{\text{end}} & 0000\ 0000\ \underline{10\text{FE}\ \text{DCBA}} \\ + & W_{-1} \ll 6 & 00\underline{11\ 0\text{LKJ}\ \text{HG}}00\ 0000 \\ - & \texttt{0xc2} \ll 6 & 00\underline{11\ 0000\ 10}00\ 0000 \\ \hline = & W_{\text{out}} & 0000\ \text{0LKJ}\ \text{HGFE}\ \text{DCBA} \end{array} \tag{57}$$

We want to increment the input pointer quickly—without a long chain of operations. We find it advantageous to always process half a vector (32 bytes or 33 bytes to include a final continuation byte) of input data per iteration like in the ASCII-only fast path. While this approach usually processes less data than first determining the maximum number of input bytes we can process, being able to load the next data quicker is more important. We avoid accessing the SIMD masks to determine whether we advance by 32 bytes or 33 bytes.

Thus we have

$$n_{\text{in}} = \begin{cases} n/2 + 1 & \text{if } \texttt{0x80} \le w_{\text{in}}[n/2] < \texttt{0xc0} \\ n/2 & \text{otherwise,} \end{cases} \tag{58}$$

processing 32 bytes per iteration unless a 2-byte sequence straddles the middle of the vector[18], in which case we process that extra byte, too.

The output buffer is advanced by the number of characters starting in the first 32 bytes, giving

$$n_{\text{out}} = \text{popcount}\big(m_{1234} \wedge (1 \ll n/2) - 1\big). \tag{59}$$

As for validation, the checks for "encodings out of range" are omitted. The check for "mismatched continuation bytes" is simplified to

$$\text{valid if} \quad m_2 \ll 1 = \neg m_{1234} \tag{60}$$

as continuation bytes must always directly follow 2-byte sequence lead bytes. The combination of all these simplifications yields a code path of roughly half the latency of the standard code path.

1/2/3 byte only

In the absence of 4-byte sequences ($m_4 = 0$), all characters are in the Basic Multilingual Plane. In this common case, we can slightly simplify the main routine. We have that $m_{+3} = m_4 \ll 3$ is zero. Consequently, we can simplify the definitions of m_c and m_{end} to

$$m_c = m_{+1} \vee m_{+2} \quad \text{and} \tag{61}$$

$$m_{\text{end}} = m_{1234} \gg 1. \tag{62}$$

[18] i. e. unless the byte at position $n/2$ is a continuation byte

The computation of W_{out} and M_{out} is eliminated. As no surrogates are present, we can omit the surrogate post-processing and don't need to account for surrogate pairs straddling the end of the vector. Instead, we directly get

$$W_{\text{out}} = W_{\text{sum}} \quad \text{and} \tag{63}$$

$$M_{\text{out}} = \neg 0. \tag{64}$$

Finally, the validation check for out-of-range encoding is slightly simpler: as surrogates cannot occur, we can drop the M_{4s} term off Eq. 48.

7 | TRANSCODING FROM UTF-16 TO UTF-8

As explained in § 2.1, UTF-16 encodes characters in the Basic Multilingual Plane (`U+0000–U+FFFF`) in one 16-bit word and all others in two words as *surrogate pairs*. To encode a code point as a surrogate pair, `0x10000` is subtracted from the character code to obtain a 20-bit binary number. The most significant 10 bits are added to `0xD800` to form a *high surrogate*, which is followed by the less significant 10 bits added to `0xDC00`, producing the corresponding *low surrogate*.

UTF-8 encodes Unicode characters in the range `U+0000–U+007F` in one byte, characters in the range `U+0080–U+07FF` in two bytes, characters in the range `U+0800–U+FFFF` in three bytes and the other characters in four bytes. Characters encoded in one UTF-16 word thus correspond to characters encoded in 1–3 bytes of UTF-8 and characters encoded in two UTF-16 words correspond to characters encoded in 4 bytes of UTF-8. This suggests the following *plan of attack* for transcoding UTF-16 to UTF-8:

1. Read a vector of 16-bit words.
2. Classify the input words into ASCII (`0x0000–0x007F`), 2-byte (`0x0080–0x07FF`), high surrogate (`0xD800–0xDBFF`), low surrogate (`0xDC00–0xDFFF`), and 3-byte (`0x0800–0xFFFF`).
3. *Zero extend* each 16-bit word to a 32-bit word and join low and high surrogates.
4. Shuffle the bits within each 32-bit word into the right positions and apply tag bits according to the type of character, producing UTF-8 sequences padded with null bytes.
5. *Compress* this vector, squeezing out the padding bytes.
6. Write the byte string to the output buffer and proceed to the next iteration.

Apart from this general plan, we also have fast code paths for the three cases of (a) ASCII characters only, (b) all in `U+0000–U+07FF`, and (c) no surrogates, complementing the default code path (d) surrogates present. Which code path to take is decided based on the characters in the current 62-byte chunk of input. We expect that most text inputs would consistently rely on the same code paths. Thus branches corresponding to the various fast paths are easy to predict, and we expect that they may provide a significant performance boost.

We would now like to explain the steps in the *plan of attack* in detail. The steps are interlinked with information produced in each step being reused for the subsequent steps. Additionally, the classification masks are reused for input validation.

First, 32 words (i. e. 64 bytes) of input are loaded from memory into W_{in}. Of these words, 31 words are encoded in the iteration with the last word serving as a *look ahead* for surrogate processing (§ 7.2). The mask

$$L = 1 \ll n/2 - 1 \tag{65}$$

indicates the position of the lookahead word in W_{in}.

7.1 | Classification and Fast Paths

We first need to find out what UTF-8 cases the characters in our input correspond to. Comparing the 16-bit words in the input vector with `0x0080` and `0x0800`, we produce the masks

$$M_{234} = (\texttt{0x0080} \leq W_{\text{in}}) \wedge \neg L \quad \text{and} \tag{66}$$

$$M_{12} = (\texttt{0x0800} > W_{\text{in}}) \tag{67}$$

telling us if non-ASCII (i. e. 2-, 3-, or 4-byte) characters and ASCII or 2-byte characters are present. ASCII characters in the lookahead are ignored to simplify some later bits of the algorithm. Based on this information, we can then embark on a code path suitable for this chunk of input.

ASCII only

If all input words represent ASCII characters ($M_{234} = 0$), we handle the input in an ASCII-only fast path: the vector is truncated to bytes (`vpmovwb`) and deposited into the output buffer, advancing it by 31 bytes. Though we could advance by 32 bytes, we want the the algorithm to proceed with a constant stride through memory irrespective of the content.

Default path

If some 3- or 4-byte characters are present ($M_{12} \vee L \neq \neg 0$), we check for surrogates. We do this by masking the words with `0xfc00` and then checking if the result is equal to `0xd800` (high surrogate, M_{hi}) or `0xdc00` (low surrogate, M_{lo}), giving

$$\begin{aligned} M_{\text{hi}} &= (\texttt{0xd800} \leq W_{\text{in}} < \texttt{0xdc00}) \wedge \neg L \\ &= (W_{\text{in}} \wedge \texttt{0xfc00} = \texttt{0xd800}) \wedge \neg L \quad \text{and} \end{aligned} \tag{68}$$

$$\begin{aligned} M_{\text{lo}} &= (\texttt{0xdc00} \leq W_{\text{in}} < \texttt{0xe000}) \\ &= (W_{\text{in}} \wedge \texttt{0xfc00} = \texttt{0xdc00}). \end{aligned} \tag{69}$$

If surrogates are found to be present ($M_{\text{hi}} \vee M_{\text{lo}} \neq 0$), we proceed to § 7.2 to handle them.[19] Otherwise we skip that step, set $W_{\text{joined}} = W_{\text{in}}$ zero-extended from 16-bit to 32-bit (`vpmovzxwd`), and directly go to § 7.3.

1/2 byte only

In the third and final case, we know that the input is a mix of ASCII and 2-byte characters. We process this case by shuffling the bits of two-byte characters into position.[20] The most significant two bits of each byte are cleared and tag bits are applied. Through this whole process, ASCII characters are left unchanged, giving us

$$W_{\text{out}} = M_{234} \; ? \; (W_{\text{in}} \ll 8 \vee W_{\text{in}} \gg 6) \wedge \texttt{0x3f3f} \vee \texttt{0x80c0} : W_{\text{in}}. \tag{70}$$

[19] Low surrogates in the lookahead are registered to permit detection of sequencing errors.

[20] As we are on a little-endian architecture, the lead byte is the less-significant of the two.

We illustrate this equation in the 2-byte case:

$$\begin{aligned} W_{\text{in}} &\quad \texttt{0000 0LKJ HGFE DCBA} \\ W_{\text{in}} \ll 8 \vee W_{\text{in}} \gg 6 &\quad \texttt{HGFE DCBA 000L KJHG} \\ W_{\text{out}} &\quad \texttt{10FE DCBA 110L KJHG} \end{aligned} \tag{71}$$

The words of W_{out} before the lookahead are then bytewise compared with `0x0800`[21] producing a mask

$$m_{\text{keep}} = W_{\text{out}} \geq_{\text{byte}} (L \; ? \; \texttt{0xffff} : \texttt{0x0800}) \tag{72}$$

holding binary 01 for ASCII characters, 11 for 2-byte characters, and 00 for the lookahead. With this mask, we finally compress W_{out} into a UTF-8 stream

$$w_{\text{out}} = \text{compress}(m_{\text{keep}}, W_{\text{out}}) \tag{73}$$

and write it to the output.

The output buffer pointer is advanced by the number of bytes of output produced, which is one byte for each word of input (sans lookahead) and another byte for each 2-byte character.

$$n_{\text{out}} = \text{popcount}(M_{234}) + n/2 - 1. \tag{74}$$

7.2 | Surrogates

When surrogates are present in the input, the bits of low surrogate have to be merged into those of the corresponding high surrogate, yielding the code point of the character to be encoded.

First, W_{in} is zero extended to 32 bits per element.[22] A vector W_{lo}, holding for each high surrogate in W_{in} its corresponding low surrogate, is produced by rotating W_{in} to the right by one element.

Then, the surrogates are joined by subtracting the tag bits (`0xd800` for the high surrogate, `0xdc00` for the low surrogate), undoing the surrogate plane shift for the high surrogate, shifting the bits of the high surrogate into place and then adding the two together. By pulling out the constants representing the tag bits and the plane shift, these additions and subtractions can be combined into one using 32-bit unsigned arithmetic. This gives us

$$\begin{aligned} W_{\text{joined}} &= M_{\text{hi}} \; ? \; \big((W_{\text{in}} - \texttt{0xd800} + \texttt{0x0040}) \ll 10\big) + (W_{\text{lo}} - \texttt{0xdc00}) : W_{\text{in}} \\ &= M_{\text{hi}} \; ? \; \big((W_{\text{in}} \ll 10) - \texttt{0x35f000}\big) + (W_{\text{lo}} - \texttt{0xdc00}) : W_{\text{in}} \\ &= M_{\text{hi}} \; ? \; (W_{\text{in}} \ll 10) + W_{\text{lo}} + \texttt{0xfca02400} : W_{\text{in}}. \end{aligned} \tag{75}$$

With the surrogate pairs decoded, we can then proceed to § 7.3 to encode into UTF-8. The vector elements corresponding to low surrogates are ignored for the rest of the algorithm.

[21] a constant we have already loaded into a register; any other constant with high-byte in range `0x01–0x7f` and low byte 0 works.

[22] From here on, each vector holds $2n$ bytes of data. These can be implemented as pairs of n-byte vectors.

7.3 | Encoding into UTF-8

When we reach this step, we have transformed W_{in} into a vector W_{joined} of 32-bit integers, holding the code points of the characters in the input.[23] We would now like to encode these code points into UTF-8, producing 1–4 bytes of output per code point.

Consider Fig. 1a: for the 2-, 3- and 4-byte case, the bits `A–W` making up the code point always appear in the same position. This suggests using the same encoding procedure for the 2-, 3-, and 4-byte case with merely different tag bits applied at the end. ASCII characters are handled with a shift into position.

The encoding procedure is based on the `vpmultishiftqb` instruction introduced with the VBMI instruction set extension. Given a vector of 64-bit words and for each such word a vector of eight bytes, the instruction uses the byte vectors as indices to pick eight 8-bit chunks of data (8 consecutive bits) from the corresponding source words. By choosing these indices such that they do not cross a 32-bit boundary, we can effectively use the instruction to select four 8-bit chunks out of each 32-bit word.

Applying the index vector $(18, 12, 6, 0)$ to each 32-bit word[24] of W_{joined}, we obtain W_{shifted} with each bit shifted into the right position with some bits left over:

W_{joined}	0000 0000 000w vuts RQPN MLKJ HGFE DCBA
index 18	00 000w vu
index 12	vuts RQPN
index 6	PN MLKJ HG
index 0	HGFE DCBA
W_{shifted}	HGFE DCBA PNML KJHG vuts RQPN 0000 0wvu

(76)

To fix up the left-over bits, we mask with `0x3f3f3f3f`, reusing the mask from the 2-byte fast path. Then, appropriate tag bits W_{tag} are applied:

case	W_{shifted} *masked with* `0x3f3f3f3f`	*tag bits*
2-byte	00FE DCBA 000L KJHG 0000 0000 0000 0000	0x80c00000
3-byte	00FE DCBA 00ML KJHG 0000 RQPN 0000 0000	0x8080e000
4-byte	00FE DCBA 00ML KJHG 00ts RQPN 0000 0wvu	0x808080f0

(77)

Finally, the ASCII case is handled by just shifting the ASCII words into position and merging these shifted characters into the output of the other cases, giving us

$$W_{\text{out}} = M_{234} \; ? \; W_{\text{shifted}} \wedge \texttt{0x3f3f3f3f} \vee W_{\text{tag}} : W_{\text{in}} \ll 24. \tag{78}$$

We end up with UTF-8 encoded characters in W_{out}. Each character occupies a 32-bit word and is padded with `0x00` bytes to 4 bytes. Input words corresponding to low surrogates have been passed through, being decoded into junk content. We get rid of the padding and the low surrogate junk by preparing a mask of bytes we want to keep and compressing out the unwanted bytes using the `vpcompressb` instruction.

In the mask, we want to keep the most significant byte of each 32-bit word and all non-zero bytes—except for

[23] In the presence of surrogates, some of these elements are ignored.

[24] i. e. the index vector $(18, 12, 6, 0, 50, 44, 38, 32)$ applied to each 64-bit word

processed low surrogates. These seemingly complex requirements can be negotiated in two steps by first building a comparison mask and then taking all bytes that are not lower than the mask. For low surrogate bytes and the lookahead, the mask is `0xff` which cannot occur in w_{out}.[25] For the most significant byte of all other words, it is `0x00` which admits every byte. For other bytes, it is `0x01`, admitting only nonzero bytes. Thus we have

$$W_{\text{keep}} = M_{\text{lo}} \vee L \;?\; \texttt{0xffffffff} : \texttt{0x00010101}, \quad \text{building} \tag{79}$$

$$m_{\text{keep}} = (W_{\text{out}} \geq_{\text{byte}} W_{\text{keep}}). \tag{80}$$

With this mask, we compress W_{out} into

$$w_{\text{out}} = \text{compress}(m_{\text{keep}}, W_{\text{out}}), \tag{81}$$

write it to the output buffer and advance the output by

$$n_{\text{out}} = \text{popcount}(m_{\text{keep}}) \tag{82}$$

bytes. Due to the little-endian orientation of the x64 architecture, the bytes of each UTF-8 sequence end up in the right order: within each 32-bit word, they are written from the least significant byte to the most significant byte.

7.4 | Validation

In contrast to the UTF-8 to UTF-16 procedure, validation of UTF-16 input is less involved. We merely have to check for the correct sequencing of surrogates: every high surrogate must be followed by a low surrogate and vice versa. As this validation only pertains surrogates, it is skipped in their absence, i. e. in all fast paths; input strings without surrogates are always valid.

To aid in this process, we only process 31 words of input in each iteration, permitting a "look ahead" into the first word of the next iteration. We also keep track of a *surrogate carry* c indicating if the first word in W_{in} was preceded by a high surrogate. This carry allows us to decide if a low surrogate in $W[0]$ is to be ignored ($c = 1$) or is a sequencing error ($c = 0$).[26]

Correct sequencing is checked for by concatenating M_{hi} with c and shifting it to the position of the corresponding low surrogates M_{lo}. The input is valid if each high surrogate corresponds to a low surrogate:

$$\text{valid if} \quad (M_{\text{hi}} \ll 1 \vee c) = M_{\text{lo}}. \tag{83}$$

The carry for the next iteration is computed as the presence of a high surrogate in the vector element right before the lookahead, giving

$$c_{\text{out}} = M_{\text{hi}} \gg (n/2 - 2) \wedge 1. \tag{84}$$

In the absence of surrogates, i. e. in the fast paths, the carry is cleared ($c_{\text{out}} = 0$).

[25] The byte value `0xff` is not possible in valid UTF-8 (cf. Table 1). It could be generated through invalid sequencing of surrogates, but we catch such an error when validating the content.

[26] When $W[0]$ is not a low surrogate, c is guaranteed to be clear.

If validation fails, we find the location of the first mismatched surrogate to transcode the words preceding the encoding error and then terminate. This is done by setting the number of remaining input words ℓ to the number of words preceding the encoding error and then jumping to the tail handling code § 7.5.

Computing the location requires more work than Eq. 83; for a high surrogate not followed by a low surrogate, that equation indicates the missing low surrogate as the first erroneous word when it should really be the unmatched high surrogate. So we proceed more carefully and first compute the sets of high surrogates not followed by low surrogates

$$M_{\text{hi}-\text{lo}} = M_{\text{hi}} \wedge \neg(M_{\text{lo}} \gg 1) \tag{85}$$

and the set of low surrogates not preceded by high surrogates

$$M_{\text{lo}-\text{hi}} = M_{\text{lo}} \wedge \neg(M_{\text{hi}} \ll 1 \vee c). \tag{86}$$

The number of valid bytes is then the longest prefix not found in either of these masks:

$$\ell = \text{ctz}(M_{\text{lo}-\text{hi}} \vee M_{\text{hi}-\text{lo}}). \tag{87}$$

7.5 | Decoding Failure and Tail Handling

At the end of the input, there might be some UTF-16 words left to process, but not enough to load a whole 64-byte vector. We deal with this remaining input in a manner similar to the UTF-8 to UTF-16 case, cf. § 6.3.

When $\ell < n/2$ words of input remain to be processed, we compute a mask

$$B = (1 \ll \ell) - 1 \tag{88}$$

of input words left to be processed. We then load the remaining input zero-masked with B,[27] giving us the input tail padded with `U+0000`. This remaining input is then processed in a final iteration of the main loop. As each null byte translates into a single byte of output, this leads to an output that is precisely $n/2 - 1 - \ell$ bytes longer than the true output length. We compensate for this by adjusting the output length reported to the caller accordingly.

In contrast to the other direction, this approach may write past the end of the output buffer if it is just long enough to hold the decoded string. We avoid this problem by performing masked stores instead of potentially storing null bytes past the end of the output.

8 | EXPERIMENTS

Our initial implementation of the algorithms was written in Intel 64 assembly for systems following the System V ABI [16]. Hand-tuned assembly can slightly surpass optimizing compilers due to better instruction scheduling and register allocation. For the measurements and comparison with competitive libraries, we have translated the code to C++ using *intrinsic functions* [10] to access AVX-512 instructions and integrated it into our *simdutf* library[28] as the *AVX-512* kernel. This library is freely available. Despite a slight loss of performance in comparison to the assembly

[27] Or in case of an encoding error, mask the already loaded vector.

[28] `https://github.com/simdutf/simdutf`

implementation, we believe that this approach facilitates better portability and integration into existing software.

Our library is organized in different kernels that are automatically selected at runtime based on the features of the CPU, a process sometimes called runtime dispatching. During benchmarking, we can manually select the different kernels. As the names suggest, the AVX2 kernel relies on AVX2 instructions (32-byte vector length) while the AVX-512 kernel using our new functions relies on AVX-512 instructions with a 64-byte vector length. Our new functions are part of the AVX-512 kernel, and the AVX2 kernel represents results presented by Lemire and Muła [2].

For benchmarking, we use Ubuntu 22.04 on a non-virtual (*metal*) server from Amazon Web Services (`c6i.metal`). These servers have 32-core Intel Xeon 8375C (Ice Lake) processors with 41 MiB of L3 memory, with 48 kB of L1 data cache memory and 1.25 MiB of L2 cache memory per core. The base clock frequency is 2.9 GHz, with a maximal frequency of 3.5 GHz. They have 256 GiB of main memory (DDR4, 3200 MHz). The benchmarks are single-threaded and we exclude disk and network accesses from our tests. The software is written in C++ and compiled with the Clang 14 C++ compiler from the LLVM project using the default `cmake` setting for a release build: `-O3 -DNDEBUG`.

We could use several threads. For example, we could split the input into segments, and compute the expected transcoded size of the segments, before transcoding each segment in its own thread. However, merely joining a thread under Linux can require tens of microseconds of waiting from the main thread. With the high speed of our functions, this penalty is equivalent to the time required to process hundreds of kilobytes of data. We could use faster synchronization techniques (e.g., spin locks and thread pools), but at the expense of complexity and power efficiency. We expect that multicore parallelism is only warranted for large inputs, in the megabytes or gigabytes range. Future work might consider such cases.

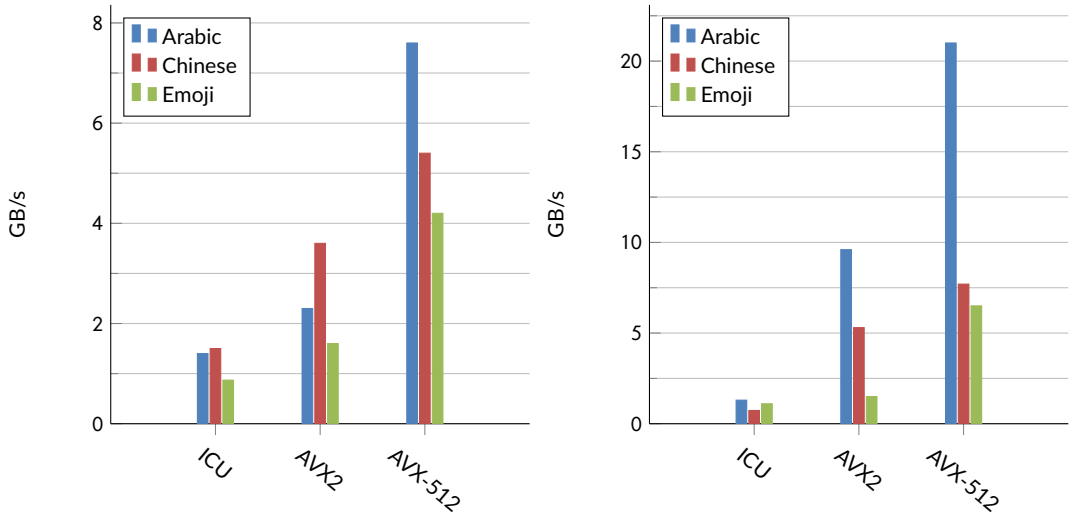
8.1 | Setup

We benchmark the transcoding of data files between UTF-8 and UTF-16 in memory. We repeat the task 10 000 times, measuring the time of each conversion: the C++ library reports a precision of 1 ns for the `std::chrono::steady_clock` measures on our test system [17]. The distribution of timings has a long tail akin to a log-normal distribution: most values are close to the minimum. We verify automatically that the difference between the minimum and the average timing is small (less than 1 %).

AVX-512 capable Intel processors prior to the Ice Lake and Rocket Lake families would systematically reduce their frequency when using 512-bit instructions, a process that Intel referred to as *licensing*. Such 512-bit licensing is no longer present in the more recent processors [18]. However, the processor frequency may fluctuate based on power consumption and heat production as is generally the case with Intel processors. We expect 512-bit instructions to use more power, and thus to run at a slightly lower frequency. Irrespective of power usage, Intel processors execute 512-bit instructions at a reduced speed initially (e.g., 4× slower)—for a few microseconds. We assume that our functions with 512-bit instructions are part of a binary executable compiled with optimizations for 512-bit capable processors so that this temporary effect is uncommon, maybe occurring only once.

We are interested in the steady-state performance of our functions: we therefore always benchmark our functions twice: once to intuitively *warm* the processor so that 512-bit instructions always execute at full speed and so that the processor has had a chance to decode the instructions. Furthermore, we may sometimes benchmark a function relying on 512-bit instructions, followed by a conventional function: to ensure that the latter is not penalized by the power usage of the first function, we pause for a millisecond when switching the benchmarked function.

We report performance results in characters per second. A given string has the same number of characters irrespective of the format (UTF-8, UTF-16). We also report speeds in gigabytes per second by taking the size of the input and dividing by the time elapsed. We focus on little-endian UTF-16, but our software supports big-endian



(a) UTF-8 to UTF-16 transcoding

(b) UTF-16 to UTF-8 transcoding

FIGURE 3 Transcoding speeds in gigabytes of input data per second for various test files. We compare against the ICU library. The simdutf library provides both AVX2 and AVX-512 functions.

UTF-16, at little cost.

We compare our work with the following competitors:

- We use the `u8u16` library [7] (last released in 2007).
- We use the `utf8lut` library [9] (last modified April 19, 2020). We require full validation of input (`cmValidate`).
- We use the C++ component from International Components for Unicode (ICU) [19].
- We use the transcoding functions of the LLVM project, they were originally produced by the Unicode Consortium.
- We use the `iconv` library, which is part of the C library (GNU C Library 2.35).

We use automatically generated (lipsum) text in Arabic, Chinese, Hebrew, Hindi, Japanese, Korean, Latin and Russian, as well as a list of emojis (henceforth Emoji).[29] When formatted as UTF-16, they range in size between 11 KiB and 170 KiB. The Chinese, Hindi, Japanese and Korean files have a high fraction of 3-byte UTF-8 characters. The Arabic, Hebrew and Russian files have a high fraction of 2-byte UTF-8 characters. Except for the Emoji file, none of the file contain 4-byte UTF-8 characters. We make our files freely available.[30]

8.2 | Results

We present speed results in gigacharacters per second regarding the validating UTF-8 to UTF-16 transcoding functions on the lipsum files in Table 4. The AVX2 and AVX-512 columns correspond to our own code, in the simdutf library (version 2.0.9, git tag `v2.0.9`). As the AVX2 kernels are implemented with a 32-byte vector length compared to the 64-byte vector length of the AVX-512 kernels, we expect a 33 % higher throughput just from using longer vectors (see § 5.2).

[29] `https://github.com/rusticstuff/simdutf8`

[30] `https://github.com/lemire/unicode_lipsum`

(a) UTF-8 to UTF-16

	llvm	*iconv*	*ICU*	*u8u16*	*utf8lut*	*AVX2*	*AVX-512*
Arabic	0.17	0.39	0.80	0.87	0.92	1.3	4.3
Chinese	0.22	0.26	0.50	0.45	0.63	1.2	1.8
Emoji	0.18	0.19	0.22	0.31	0.18	0.40	1.0
Hebrew	0.17	0.39	0.80	0.87	0.92	1.3	4.3
Hindi	0.16	0.21	0.43	0.49	0.72	0.84	1.7
Japanese	0.21	0.26	0.51	0.46	0.64	1.2	1.7
Korean	0.13	0.30	0.62	0.54	0.72	0.89	1.8
Latin	0.35	0.56	1.5	13.	1.0	22.	20.
Russian	0.17	0.29	0.46	0.86	0.92	1.3	4.2
harm. mean	0.18	0.29	0.50	0.60	0.57	1.0	2.3

(b) UTF-16 to UTF-8

	llvm	*iconv*	*ICU*	*utf8lut*	*AVX2*	*AVX-512*
Arabic	0.38	0.30	0.67	2.4	4.8	11.
Chinese	0.38	0.28	0.36	2.4	2.6	3.9
Emoji	0.29	0.20	0.27	0.37	0.38	1.6
Hebrew	0.48	0.32	0.68	2.3	4.8	11.
Hindi	0.31	0.21	0.21	2.4	2.6	3.8
Japanese	0.38	0.26	0.37	2.3	2.7	3.8
Korean	0.43	0.30	0.37	2.3	2.7	3.8
Latin	0.58	0.56	0.91	2.3	18.	20.
Russian	0.28	0.23	0.23	2.4	4.8	11.
harm. mean	0.37	0.27	0.36	1.5	1.9	4.5

TABLE 4 Validating transcoding speeds (gigacharacters per second) over the lipsum datasets, last row is the harmonic mean of the column. The last column (AVX-512) presents the results from our new algorithms.

On the Latin file, the new AVX-512 kernel fails to improve on the earlier AVX2 kernel: the Latin dataset is made almost entirely of ASCII inputs which the AVX-512 kernel processes 32 bytes at a time, just like the AVX2 kernel. On the Emoji file, the new AVX-512 kernel achieves one gigacharacter per second when transcoding from UTF-8, which is more than twice as fast as any competitor. When transcoding from UTF-16, the new kernel transcode the Emoji file at 1.6 gigacharacters per second, which is more than four times as fast as any competitor. Whether transcoding from UTF-8 or UTF-16, the new kernel does well when transcoding inputs dominated by 2-byte UTF-8 sequences (Arabic, Hebrew and Russian): it is twice as fast as any competitor. For inputs dominated by 3-byte UTF-8 sequences (Chinese, Hindi, Japanese and Korean), the gain compared to the earlier AVX2 kernel is of the order of 50%.

Fig. 3 and Table 5 present the speed results in gigabytes of inputs per second. Whereas the speeds of the AVX-

(a) UTF-8 to UTF-16

	llvm	*iconv*	*ICU*	*u8u16*	*utf8lut*	*AVX2*	*AVX-512*
Arabic	0.31	0.70	1.4	1.5	1.6	2.3	7.6
Chinese	0.66	0.77	1.5	1.3	1.9	3.7	5.4
Emoji	0.72	0.78	0.87	1.2	0.70	1.6	4.2
Hebrew	0.31	0.70	1.4	1.6	1.6	2.3	7.7
Hindi	0.43	0.56	1.2	1.3	1.9	2.3	4.6
Japanese	0.60	0.74	1.5	1.3	1.9	3.5	5.0
Korean	0.32	0.73	1.5	1.3	1.8	2.2	4.5
Latin	0.35	0.56	1.5	13.	1.0	22.	20.
Russian	0.31	0.52	0.83	1.6	1.7	2.4	7.7
harm. mean	0.40	0.66	1.2	1.5	1.4	2.6	6.0

(b) UTF-16 to UTF-8

	llvm	*iconv*	*ICU*	*utf8lut*	*AVX2*	*AVX-512*
Arabic	0.76	0.61	1.3	4.7	9.6	21.
Chinese	0.76	0.57	0.73	4.7	5.3	7.7
Emoji	1.2	0.82	1.1	1.5	1.5	6.5
Hebrew	0.96	0.63	1.3	4.7	9.6	21.
Hindi	0.63	0.42	0.41	4.7	5.3	7.7
Japanese	0.77	0.52	0.74	4.7	5.3	7.6
Korean	0.86	0.60	0.73	4.7	5.4	7.6
Latin	1.2	1.1	1.8	4.7	37.	40.
Russian	0.56	0.45	0.46	4.7	9.6	21.
harm. mean	0.80	0.59	0.77	3.8	5.1	11.

TABLE 5 Validating transcoding speeds in gigabytes of input per second over the lipsum datasets, last row is the harmonic mean of the column. The last column (AVX-512) presents the results from our new algorithms.

512 function in gigacharacters per second vary by multiples (from 4.3 with Arabic to 1.0 with Emoji), the gaps are much less significant in gigabytes per second (from 7.6 with Arabic to 4.2 with Emoji).

Table 6 presents the number of instructions retired per character, measured using the hardware performance counters provided by Intel.[31] In the worst case (for the Emoji files), the new AVX-512 kernel still requires fewer than 6 instructions per character to transcode in either direction. Except for the Latin files, the new AVX-512 kernel requires far fewer than half the number of instructions than the AVX2 kernel when transcoding from UTF-8. For example, we reduce the number of instructions by a factor of three for the Arabic file. Table 7 provides the number of instructions

[31] Under Linux, the performance counters are made available by the operating system. A C program can query them using the functions defined in the `linux/perf_event.h` header.

per cycle. We find that the AVX-512 kernel is associated with a lower number of instructions retired per cycle—especially so when transcoding from UTF-8. Correspondingly, we expect a lower number of 64-byte instructions being retired per cycle compared to 32-byte instructions due to the microarchitectures of the Intel CPUs (§ 5.2).

The `utf8lut` library, when transcoding from UTF-8, requires fewer instructions than our AVX-2 kernel, but it is associated with few instructions per cycle. Hence, the `utf8lut` library is generally slower than our AVX-2 kernel despite relying on the same instruction set. The `utf8lut` library relies on a 2 MiB table for UTF-8 to UTF-16 transcoding as opposed to a small table (11 KiB) for our AVX-2 kernel, and no table at all for our AVX-512 kernel. A large table may cause the CPU to wait for loads to complete and increases overall cache pressure.

(a) UTF-8 to UTF-16

	llvm	*iconv*	*ICU*	*u8u16*	*utf8lut*	*AVX2*	*AVX-512*
Arabic	65.	52.	27.	15.	5.3	7.4	2.3
Chinese	82.	78.	38.	31.	8.4	11.	3.2
Emoji	100	100	93.	45.	95.	29.	5.4
Hebrew	65.	51.	27.	15.	5.3	7.4	2.3
Hindi	80.	71.	34.	28.	7.3	12.	3.0
Japanese	81.	76.	37.	30.	8.2	11.	3.2
Korean	75.	66.	31.	25.	6.9	12.	2.7
Latin	56.	35.	11.	0.65	5.3	0.35	0.24
Russian	66.	52.	27.	16.	5.3	7.2	2.3

(b) UTF-16 to UTF-8

	llvm	*iconv*	*ICU*	*utf8lut*	*AVX2*	*AVX-512*
Arabic	37.	53.	27.	6.3	2.6	1.1
Chinese	48.	67.	41.	6.3	4.5	2.2
Emoji	62.	90.	67.	51.	48.	5.4
Hebrew	37.	53.	27.	6.3	2.6	1.1
Hindi	45.	62.	38.	6.3	4.5	2.2
Japanese	47.	65.	40.	6.3	4.5	2.2
Korean	43.	58.	35.	6.3	4.5	2.2
Latin	31.	34.	19.	6.3	0.69	0.55
Russian	37.	53.	27.	6.3	2.6	1.1

TABLE 6 CPU instructions retired per character when transcoding with validation. The last column (AVX-512) presents the results from our new algorithms.

In Fig. 4, we present the measured transcoding speed for various small prefixes of the Arabic files. We find that as long as the input has hundreds of characters, we can reach and exceed a billion characters decoded per second.

(a) UTF-8 to UTF-16

	llvm	*iconv*	*ICU*	*u8u16*	*utf8lut*	*AVX2*	*AVX-512*
Arabic	3.3	5.8	6.1	3.8	1.4	2.7	2.9
Chinese	5.2	5.7	5.3	4.0	1.5	3.9	1.7
Emoji	5.1	5.8	5.8	3.9	4.8	3.3	1.6
Hebrew	3.2	5.8	6.1	3.8	1.4	2.7	2.9
Hindi	3.7	4.3	4.2	3.9	1.5	2.9	1.5
Japanese	4.8	5.5	5.4	3.9	1.5	3.9	1.6
Korean	2.8	5.6	5.4	3.9	1.4	3.1	1.5
Latin	5.6	5.6	4.6	2.4	1.6	2.2	1.4
Russian	3.2	4.3	3.6	3.9	1.4	2.7	2.9
harm. mean	3.9	5.3	5.0	3.6	1.6	3.0	1.8

(b) UTF-16 to UTF-8

	llvm	*iconv*	*ICU*	*utf8lut*	*AVX2*	*AVX-512*
Arabic	4.0	4.6	5.2	4.2	3.5	3.3
Chinese	5.2	5.4	4.8	4.2	3.4	2.5
Emoji	5.2	5.3	5.2	5.3	5.1	2.6
Hebrew	5.1	4.8	5.2	4.2	3.5	3.3
Hindi	4.1	3.7	2.6	4.2	3.4	2.5
Japanese	5.2	4.9	4.7	4.2	3.4	2.5
Korean	5.3	5.0	4.5	4.2	3.4	2.5
Latin	5.2	5.5	5.0	4.2	3.6	3.2
Russian	3.0	3.4	2.1	4.2	3.5	3.3
harm. mean	4.5	4.6	3.9	4.3	3.6	2.8

TABLE 7 CPU instructions retired per cycle when transcoding with validation. The last column (AVX-512) presents the results from our new algorithms.

Historically, some processors could only read and write data when the memory address was a multiple of the data size. Older Intel processors could read and write at any address, but with a severe penalty for unaligned memory addresses. On recent processors (e.g., Intel's Sandy Bridge microarchitecture launched in 2011), there is reportedly no measurable performance penalty for reading or writing misaligned memory operands [14]. However, there might be indirect penalties (e.g., accessing more cache lines). In the hope of achieving better performance, we could require that our memory buffers start at an address divisible by 512 bits. However, we expect that the performance of the transcoding functions is generally unaffected by memory alignment on our test system. Thus our benchmarking code does not align the memory in any particular manner, relying instead on the default behavior of the memory allocator. To test the effect of the memory alignment of the input, we transcoded the same data, but shifted by 0 to 512 bytes

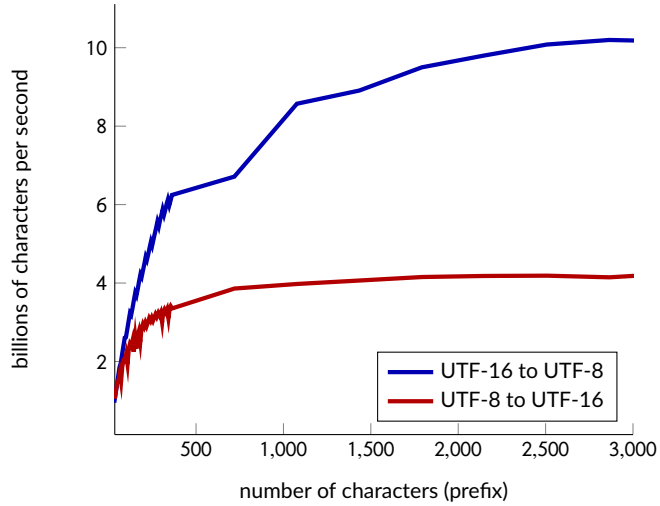


FIGURE 4 Validating transcoding speed in billions of characters per second for prefixes of various lengths of the Arabic files using our techniques.

inside a buffer. Using one of our UTF-8 file (Arabic), we measured a difference of 2% between the fastest and slowest alignment when using our fast (AVX-512) transcoder. We get a similar result if we transcode from a fixed input to offsetted locations inside a destination buffer. Our results suggest that memory alignment is likely not a significant factor.

9 | CONCLUSION

It is not *a priori* obvious that character transcoding is amenable to SIMD processing. Earlier work achieved high speeds but it required kilobytes of lookup tables [2]. Our work indicates that the AVX-512 instruction-set extensions enables high speed for tasks such as character transcoding—without lookup tables and using few instructions. It suggests that some features of the AVX-512 instruction-set extensions might serve as a reference for future instruction-set extensions. In particular, we find masked SIMD instructions (move, load, store, compress) with byte-level granularity useful.

Both Intel and AMD support AVX-512 instructions. They also both offer specialized compilers, tuned for their processors. Future work could compare the performance of our routines on more varied Intel and AMD processors (e.g., Intel Rocket Lake and Sapphire Rapids, AMD Zen 4), using specialized compilers (e.g., from Intel and AMD) and hand-tuned assembly. We could extend our benchmarks to cover a wider range of string.

Acknowledgements

We thank W. Muła who produced an early UTF-8 to UTF-16 transcoder using AVX-512 instructions. The version presented in this manuscript follows a different design but Muła's work provided a crucial motivation. We thank N. Boyer for his technical work on our software library, benchmarks, and tests.

references

[1] Keiser J, Lemire D. Validating UTF-8 in less than one instruction per byte. Software: Practice and Experience 2021;51(5). doi:10.1002/spe.2920.

[2] Lemire D, Muła W. Transcoding billions of Unicode characters per second with SIMD instructions. Software: Practice and Experience 2022;52(2):555–575. doi:10.1002/spe.3036.

[3] z/Architecture Principles of Operation. International Business Machines Corporation, fourteenth ed.; 2022, document SA22-7832-13.

[4] Muła W, Lemire D. Base64 encoding and decoding at almost the speed of a memory copy. Software: Practice and Experience 2020;50(2):89–97. doi:10.1002/spe.3036.

[5] Hoffman P, Yergeau F, UTF-16, an encoding of ISO 10646; 2000. Internet Engineering Task Force, Request for Comments: 3629. `https://tools.ietf.org/html/rfc2781` [last checked July 2021].

[6] Yergeau F, UTF-8, a transformation format of ISO 10646; 2003. Internet Engineering Task Force, Request for Comments: 3629. `https://tools.ietf.org/html/rfc3629` [last checked July 2021].

[7] Cameron RD. A case study in SIMD text processing with parallel bit streams: UTF-8 to UTF-16 transcoding. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming ACM; 2008. p. 91–98. doi:10.1145/1345206.1345222.

[8] Inoue H, Komatsu H, Nakatani T. Accelerating UTF-8 Decoding Using SIMD Instructions (in Japanese). Information Processing Society of Japan Transactions on Programming 2008;1(2):1–8.

[9] Gatilov S, utf8lut: Vectorized UTF-8 converter; 2012. `https://bit.ly/3qI7BVQ` [last checked October 2022].

[10] Intel Intrinsics Guide; 2023. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html` [last checked June 2023].

[11] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2. Intel Corporation; 2022, order number 325383-077US.

[12] Fog A. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. In: Optimization Manuals, vol. 4 published online at `https://agner.org/optimize`; 2022.

[13] Abel A, Reineke J. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In: ASPLOS ASPLOS '19, New York, NY, USA: ACM; 2019. p. 673–686. doi:10.1145/3297858.3304062.

[14] Fog A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. In: Optimization Manuals, vol. 3 published online at `https://agner.org/optimize`; 2022.

[15] Abel A, Reineke J. uiCA: Accurate Throughput Prediction of Basic Blocks on Recent Intel Microarchitectures. In: Rauchwerger L, Cameron K, Nikolopoulos DS, Pnevmatikatos D, editors. ICS '22: 2022 International Conference on Supercomputing, Virtual Event, USA, June 27-30, 2022 ICS '22, ACM; 2022. p. 1–12. doi:10.1145/3524059.3532396.

[16] Matz M, Hubička J, Jaeger A, Mitchell M, System V Application Binary Interface AMD64 Architecture Processor Supplement; 2012. Draft Version 0.99.6.

[17] Najafi A, Tai A, Wei M. Systems Research is Running out of Time. In: Proceedings of the Workshop on Hot Topics in Operating Systems HotOS '21, New York, NY, USA: Association for Computing Machinery; 2021. p. 65–71. `https://doi.org/10.1145/3458336.3465293`, doi:10.1145/3458336.3465293.

[18] Downs T, Ice Lake AVX-512 Downclocking; 2020. `https://travisdowns.github.io/blog/2020/08/19/icl-avx512-freq.html` [last checked July 2022].

[19] International Components for Unicode (ICU); 2010. `http://site.icu-project.org` [last checked July 2021].

A | UTF-8 TO UTF-16: SUMMARY OF VARIABLES

symbol	*type*	*description*	*Eq.*
w_{in}	byte vector	input vector	—
m_1	byte mask	1-byte sequence lead bytes in w_{in}	(6)
m_{234}	byte mask	2/3/4-byte sequence lead bytes in w_{in}	(7)
m_2	byte mask	2-byte sequence lead bytes in w_{in}	—
m_{34}	byte mask	3/4-byte sequence lead bytes in w_{in}	(8)
m_3	byte mask	3-byte sequence lead bytes in w_{in}	(43)
m_4	byte mask	4-byte sequence lead bytes in w_{in}	(9)
m_{1234}	byte mask	lead bytes in w_{in}	(10)
m_{+1}	byte mask	second byte of each sequence in w_{in}	(37)
m_{+2}	byte mask	third byte of each sequence in w_{in}	(38)
m_{+3}	byte mask	fourth byte of each sequence in w_{in}	(11)
m_{end}	byte mask	last byte of each sequence in w_{in}	(12)
M_3	word mask	3-byte characters in W_{out}	(44)
M_{hi}	word mask	high surrogates in W_{out}	(24)
M_{lo}	word mask	low surrogates in W_{out}	(23)
w_{stripped}	byte vector	w_{in} with tag bits stripped off	(14)
P	word vector	indices of last-in-sequence bytes in w_{in}	(15)
m_{-1}	byte mask	mask to admit only 2nd-last bytes of w_{in}	(17)
m_{-2}	byte mask	mask to admit only 3rd-last bytes of w_{in}	(18)
W_{end}	word vector	last byte of each sequence in w_{in}	(16)
W_{-1}	word vector	second-last byte of each sequence in w_{in}	(19)
W_{-2}	word vector	third-last byte of each sequence in w_{in}	(20)
W_{sum}	word vector	W_{end}, W_{-1}, and W_{-2} bits assembled	(21)
W_{out}	word vector	W_{end} with surrogates fixed up	(25)
M_{out}	word mask	valid words in W_{out}	(27)
$m_{\text{processed}}$	byte mask	last byte of each sequence in M_{out}	(28)
n_{in}	integer	number of w_{in} bytes processed	(29)
n_{out}	integer	number of words written out	(30)
ℓ	integer	number of input bytes left to process	—
b	byte mask	input bytes left to process	(34)
m_c	byte mask	where continuation bytes should be in w_{in}	(39)
m_{pre}	byte mask	w_{in} bytes preceding first mismatch	(41)
$M_{<\text{U+800}}$	word mask	overlong 3-byte characters in W_{out}	(45)
M_{3s}	word mask	3-byte characters encoding surr. in W_{out}	(46)
M_{4s}	word mask	surrogates not encoding surr. in W_{out}	(47)

Variables pertaining to the fast paths are not listed.

B | UTF-16 TO UTF-8: SUMMARY OF VARIABLES

symbol	*type*	*description*	*Eq.*
W_{in}	word vector	input vector	—
L	word mask	position of lookahead word	(65)
M_{234}	word mask	words that are 2–4-byte characters	(66)
M_{12}	word mask	words that are 1- and 2-byte characters	(67)
M_{hi}	word mask	words that are high surrogates	(68)
M_{lo}	word mask	words that are low surrogates	(69)
W_{lo}	dword vector	W_{in} shifted to the right by one element	—
W_{joined}	dword vector	W_{in} with high and low surrogates joined	(75)
W_{shifted}	dword vector	W_{joined} bits shifted with `vpmultishiftqb`	(76)
W_{tag}	dword vector	UTF-8 tag bits for W_{out}	(77)
W_{out}	dword vector	W_{in} transcoded to UTF-8 with padding	(78)
W_{keep}	dword vector	magic constant for which bytes to keep	(79)
m_{keep}	byte mask	mask of W_{out} bytes we want to keep	(80)
w_{out}	byte vector	output string without padding	(81)
n_{out}	integer	length of w_{out}	(82)
c	integer	surrogate carry (in)	—
c_{out}	integer	surrogate carry out	(84)
$M_{\text{hi}-\text{lo}}$	word mask	high surrogates not followed by low surr.	(85)
$M_{\text{lo}-\text{hi}}$	word mask	low surrogates not preceded by high surr.	(86)
ℓ	integer	number of input words left to process	(87)
B	word mask	input words left to process	(88)