

Faster Base64 Encoding and Decoding using AVX2 Instructions

WOJCIECH MUŁA,

DANIEL LEMIRE, Université du Québec (TELUQ)

Web developers use base64 formats to include images, fonts, sounds and other resources directly inside HTML, JavaScript, JSON and XML files. We estimate that billions of base64 messages are decoded every day. We are motivated to improve the efficiency of base64 encoding and decoding. Compared to state-of-the-art implementations, we multiply the speeds of both the encoding ($\approx 10\times$) and the decoding ($\approx 7\times$). We achieve these good results by using the single-instruction-multiple-data (SIMD) instructions available on recent Intel processors (AVX2). Our accelerated software abides by the specification and reports errors when encountering characters outside of the base64 set. It is available online as free software under a liberal license.

CCS Concepts: •**Theory of computation** → **Vector / streaming algorithms**;

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Binary-to-text encoding, Vectorization, Data URI, Web Performance

1. INTRODUCTION

We use base64 formats to represent arbitrary binary data as text. Base64 is part of the MIME email protocol [Linn 1993; Freed and Borenstein 1996], used to encode binary attachments. Base64 is included in the standard libraries of popular programming languages such as Java, C#, Swift, PHP, Python, Rust, JavaScript and Go. Major database systems such as Oracle and MySQL include base64 functions.

On the Web, we often combine binary resources (images, videos, sounds) with text-only documents (XML, JavaScript, HTML). Before a Web page can be displayed, it is often necessary to retrieve not only the HTML document but also all of the separate binary resources it needs. The round-trips needed to retrieve all of the resources are often a performance bottleneck [Everts 2013]. Consequently, major websites—such as Google, Bing, and Baidu—deliver small images within HTML pages using the data URI scheme [Masinter 1998]. A data URI takes the form “data:<content type>;base64,<base64 data>”. For example, consider the `img` element

```

```

where the text “R01GOD1...” is a base64 representation of the binary data of a GIF image. Data URIs are supported by all major browsers [Johansen et al. 2013]. We estimate that billions of pages containing base64 data are loaded every day.

Base64 formats encode arbitrary bytes into a stream of characters chosen from a list of 64 ASCII characters. Three arbitrary bytes can be thus encoded using four ASCII characters. Though base64 encoding increases the number of bytes by 33%, this is alleviated by the commonly used text compression included in the HTTP protocol [Fielding et al. 1999]. The size difference, after compression, can be much smaller than 33% and might even be negligible [Calhoun 2011].

Base64 has many applications on the Web beyond embedding resources within HTML pages as an optimization:

- The recently introduced Web Storage specification allows Web developers to store text data (including base64-encoded resources) persistently within the browser [Hickson

This work is supported by Natural Sciences and Engineering Research Council of Canada, grant 261437. Author’s addresses: D. Lemire, Université du Québec (TELUQ), 5800, Saint-Denis street, Montreal (Quebec) H2S 3L5, Canada.

2016]. With Web Storage, developers can ensure that base64-encoded images and fonts are cached in the browser.

- Similarly, base64 embeds binary data within XML and JSON files generated by web services, as these text-only formats do not otherwise allow binary content. A Web page can retrieve XML and JSON documents and decode the corresponding dynamically-generated binary resources on the fly. Correspondingly, several database systems frequently code and decode base64 strings even though they store binary data as binary:
 - MongoDB normally receives and sends binary data as base64-encoded strings [MongoDB 2017].
 - Elasticsearch accepts binary values as base64-encoded strings [Elastic 2017].
 - SQL Server users can add the BINARY BASE64 qualifier when issuing FOR XML queries, so that the generated XML encodes binary objects using base64 [Microsoft 2017].
 - Amazon SimpleDB automatically encodes data sequences that are not valid in XML using base64 [Amazon 2015].
 - Amazon DynamoDB supports binary attributes, but they are normally exchanged in a base64-encoded form within JSON documents [Amazon 2017]. Crane and Lin report that decoding binary attributes from base64 is slow [Crane and Lin 2017].

Base64 can also be used for security and privacy purposes. Credentials are often stored and transmitted using base64, e.g., in the HTTP Basic authentication method. There are also more advanced applications:

- Many systems allow users to communicate text more freely than binary data. Using this principle, Tierney et al. use base64 to allow users to share encrypted pictures on social networks [Tierney et al. 2013], even when such networks do not natively support this feature.
- Moreover, even when multiple HTTP queries to retrieve resources are efficient, they make it easier for adversaries to track users. Indeed, TCP/IP packet headers cannot be encrypted and they reveal the size of the data, as well as the destination and source addresses. Thus even encrypted Web access may not guarantee anonymity. Tang and Lin show that we can use base64 to better obfuscate Web queries [Tang and Lin 2015].

Encoding and decoding base64 data is fast. We do not expect base64 decoding to be commonly a bottleneck in Web browsers. Yet it can still be much slower to decode data than to copy it: e.g., `memcpy` may use as little as 0.03 cycles per byte while a fast base64 decoder might use 1.8 cycles per byte on the same test (and be $60\times$ slower), see Table VI. Because base64 is ubiquitous and used on a massive scale within servers and database systems, there is industry interest in making it run faster [Char 2014].

Most commodity processors (Intel, AMD, ARM, POWER) benefit from single-instruction-multiple-data (SIMD) instructions. Unlike regular (scalar) instructions, these SIMD instructions operate on several words at once (or “vectors”). For example, recent x64 processors benefit from AVX2 instructions, operating on 256-bit vectors. Though compilers can automatically use these instructions, it may be necessary to design algorithms with SIMD instructions in mind for best speed. Unlike regular (or “scalar”) instructions operating on single words, SIMD instructions operate on several words at once. We refer to these groups of words as vectors. These vectors are implemented as wide registers within the processors. For example, recent x64 processors benefit from AVX2 instructions, operating on 256-bit vectors. We treat such vectors as arrays of 32 bytes, arrays of sixteen 16-bit integers or arrays of eight 32-bit integers.

2. BASE64

Base64 code is made streams of 6-bit words represented as ASCII characters. Blocks of four 6-bit words correspond bijectively to blocks of three 8-bit words (bytes).

- During the encoding of an arbitrary binary stream, each block of three input bytes (or $3 \times 8 = 24$ bits) is unpacked to four 6-bit words ($3 \times 6 = 24$ bits). Each of the four 6-bit words corresponds to an ASCII character. See Algorithm 1. If the length of the input is not divisible by three bytes, then the encoder may use the special padding character ('='). There is one padding character per leftover byte (one or two). The length of a valid base64 string is normally divisible by four. In some applications, it may be acceptable to omit the padding characters ('=') if the size of the binary data is otherwise known.
- Most base64 decoders translate blocks of four ASCII letters into blocks of four 6-bit integer values (in $[0, 63)$). Each of these blocks is then packed into three bytes. See Algorithm 2. When the base64 stream ends with one or two padding characters ('='), two or one final bytes are decoded.

ALGORITHM 1: Base64 encoding

Require: A stream s of n bytes, indexed as $s_0, s_1, \dots, s_{n-1} \in [0, 256)$

Require: A function B mapping values in $[0, 64)$ to ASCII characters (e.g., see Table I)

```
1:  $p \leftarrow$  empty buffer of ASCII characters
2: for  $i$  in  $0, 3, \dots, n - (n \bmod 3) - 3$  do
3:   append  $B(s_i \div 4)$  to  $p$ 
4:   append  $B(((s_i \times 16) \bmod 64) + (s_{i+1} \div 16))$  to  $p$ 
5:   append  $B(((s_{i+1} \times 4) \bmod 64) + (s_{i+2} \div 64))$  to  $p$ 
6:   append  $B((s_{i+2} \bmod 64))$  to  $p$ 
7: end for
8:  $i \leftarrow n - n \bmod 3$ 
9: if  $i < n$  then
10:  append  $B(s_i \div 4)$  to  $p$ 
11:  if  $i = n - 1$  then
12:    append  $B(((s_i \times 16) \bmod 64))$  to  $p$ 
13:    append padding character '=' to  $p$ 
14:  else if  $i = n - 2$  then
15:    append  $B(((s_i \times 16) \bmod 64) + (s_{i+1} \div 16))$  to  $p$ 
16:    append  $B(((s_{i+1} \times 4) \bmod 64))$  to  $p$ 
17:  end if
18:  append padding character '=' to  $p$ 
19: end if
20: return  $p$ 
```

Base64 standards define a lookup table to translate between 6-bit values (in $[0, 63)$) and ASCII characters. We consider the standard [Josefsson 2006] where the following characters are used: A ... Z, a ... z, 0 ... 9, + and /, as in Table I. Unless otherwise specified, the decoder should report an error when characters outside of this set are encountered.

Sometimes, we want to encode binary data within an URL where the '+' and '/' characters have special meaning. Thus we may choose an instance of base64 called base64url [Josefsson 2006]. The sole difference is that value 62 is represented by '-' instead of '+' and the value 63 is represented by '_' instead of '/'. Thus base64url avoids using the characters '+' and '/', and a base64url text can be safely included in an URL. The JSON Web Signature proposal relies on base64url [Jones et al. 2015]. Our

ALGORITHM 2: Base64 decoding

Require: A stream c of n ASCII characters, indexed as C_0, C_1, \dots, C_{n-1} , n must be divisible by 4

Require: A function A mapping ASCII characters to values in $[0, 64)$ (e.g., see Table I), using the conventional that the padding character '=' has value 0, and returning a negative integer if an unsupported ASCII character is found

```
1:  $p \leftarrow$  empty buffer of bytes used to store values in  $[0, 256)$ 
2: for  $i$  in  $0, 4, \dots, n - 4$  do
3:    $a \leftarrow A(C_i)$ 
4:    $b \leftarrow A(C_{i+1})$ 
5:    $c \leftarrow A(C_{i+2})$ 
6:    $d \leftarrow A(C_{i+3})$ 
7:   if any of  $a, b, c, d$  is negative then
8:     report an error as unexpected character was encountered (based on Table I)
9:   end if
10:  append byte value  $(a \times 4) + (b \div 16)$  to  $p$ 
11:  if  $C_{i+2}$  is the padding character ('=') then
12:    return  $p$ 
13:  end if
14:  append byte value  $(b \times 16) \bmod 256 + (c \div 4)$  to  $p$ 
15:  if  $C_{i+3}$  is the padding character ('=') then
16:    return  $p$ 
17:  end if
18:  append byte value  $(c \times 64) \bmod 256 + d$  to  $p$ 
19: end for
20: return  $p$ 
```

Table I: Base64 mapping between 6-bit values and ASCII characters. For each ASCII character, we also provide the code point or byte value as an hexadecimal number. The '=' character pads the end of the stream if the number of bytes is not divisible by 3.

value	ASCII	char	value	ASCII	char	value	ASCII	char	value	ASCII	char
0	0x41	A	16	0x51	Q	32	0x67	g	48	0x77	w
1	0x42	B	17	0x52	R	33	0x68	h	49	0x78	x
2	0x43	C	18	0x53	S	34	0x69	i	50	0x79	y
3	0x44	D	19	0x54	T	35	0x6a	j	51	0x7a	z
4	0x45	E	20	0x55	U	36	0x6b	k	52	0x30	0
5	0x46	F	21	0x56	V	37	0x6c	l	53	0x31	1
6	0x47	G	22	0x57	W	38	0x6d	m	54	0x32	2
7	0x48	H	23	0x58	X	39	0x6e	n	55	0x33	3
8	0x49	I	24	0x59	Y	40	0x6f	o	56	0x34	4
9	0x4a	J	25	0x5a	Z	41	0x70	p	57	0x35	5
10	0x4b	K	26	0x61	a	42	0x71	q	58	0x36	6
11	0x4c	L	27	0x62	b	43	0x72	r	59	0x37	7
12	0x4d	M	28	0x63	c	44	0x73	s	60	0x38	8
13	0x4e	N	29	0x64	d	45	0x74	t	61	0x39	9
14	0x4f	O	30	0x65	e	46	0x75	u	62	0x2b	+
15	0x50	P	31	0x66	f	47	0x76	v	63	0x2f	/

work would be equally applicable to `base64url`, as the difference between `base64` and `base64url` has little impact on encoding and decoding algorithms.

2.1. Character Encodings

Base64 was designed with the ASCII character encoding in mind [Josefsson 2006]. In a document using the ASCII encoding, only seven of the eight bits of each byte is used. By convention, each ASCII character has a corresponding byte value (also called code point) in $[0, 128)$.

There are several supersets to the ASCII character encoding (e.g., UTF-8 or ISO 8859-1): they interpret strings of byte values in $[0, 128)$ as ASCII strings. Only byte values with the most significant bits set are interpreted differently (e.g., as accented characters such as 'é'). In other words, if we need to include an ASCII string within a string that uses a superset of the ASCII character encoding, we only need to copy the byte values. Thus base64 is practical with all ASCII supersets.

Most Web pages are served using the Unicode format UTF-8 [Davis 2012] which supports up to 1 114 112 possible characters. Some programming languages (e.g., Go and Python) also default on UTF-8. XML documents use UTF-8 by default. Conveniently, UTF-8 is an ASCII superset. In UTF-8, only the ASCII characters can be represented using a single byte. All non-ASCII characters in UTF-8 require from two to four bytes.

It might seem like base64 is suboptimal: there are many more than 64 distinct characters. However, there are only 95 printable ASCII characters, and they include the space and the quotes (" and '), the ampersand (&) and the less-than sign (<). Thus there are only about 90 characters that are represented as a single byte in UTF-8 that would be generally usable in HTML and XML. If we restrict the size of our table to a power of two, for simplicity and computational efficiency, then 64 characters is best.

2.2. Efficient Scalar Encoding

Throughout, we consider byte values as unsigned integers in $[0, 256)$. Thus we can think of encoding as mapping a stream of numbers in $[0, 256)$ to ASCII characters.

In the main loop of the encoding Algorithm 1, we combine three-byte values (s_i, s_{i+1}, s_{i+2}) arithmetically into four values in $[0, 64)$, that is

- $s_i \div 4$,
- $((s_i \times 16) \bmod 64) + (s_{i+1} \div 16)$,
- $((s_{i+1} \times 4) \bmod 64) + (s_{i+2} \div 64)$
- and $s_{i+2} \bmod 64$.

Then we pass these four values to the function B which looks up the corresponding ASCII character. In practice, an encoder might implement the function B efficiently as a lookup table, using a 64-byte array.

Thus, given three input bytes, we get four output characters:

- The first character is generated from the six most significant bits of the first byte value.
- The second character is determined from the two least significant bits from the first byte value and from the four most significant bits of the second byte value.
- The third character represents the four least significant bits of the second byte value and the two most significant bits from the third byte value.
- The last character is determined by the six least significant bits from the third byte value.

Thus if we write the byte values starting from the left, with the bits within each byte written from the most significant to the least significant, we have that the first 6 bits

from the left determine the first character, the next 6 bits the second character and so forth.

Example 2.1. Suppose that we need to encode the three byte values 71, 73, 70. We construct four 6-bit values out of these four bytes:

- $71 \div 4 = 17$,
- $((71 \times 16) \bmod 64) + (73 \div 16) = 52$,
- $((73 \times 4) \bmod 64) + (70 \div 64) = 37$
- and $70 \bmod 64 = 6$.

We then look-up the resulting 6-bit values 17, 52, 37, 6 in Table I to get the ASCII characters R, O, l, G. The string R0lG is the base64-encoded version of the three input bytes 71, 73, 70.

One of the fastest encoder [Galbreath 2016] (used by the Google Chrome browser) optimizes the main loop of Algorithm 1 by using several 256-byte arrays. One 256-byte array represents $x \rightarrow B(x \div 4)$, whereas another represents $x \rightarrow B(x \bmod 64)$. This saves a few operations at the expense of a slightly increased memory usage:

- instead of computing $s_i \div 4$ and then looking up the resulting index in a table, one can seek directly s_i in a larger table, saving the cost of the division by 4 (which can be implemented as a shift);
- instead of computing $s_{i+2} \bmod 64$ before looking up the result in a table, we can seek s_{i+2} directly in a larger table, saving the cost of the modulo reduction (which can be implemented with a bitwise AND).

Using additional memory to save a few arithmetic and logical operations may improve the performance.

2.3. Efficient Scalar Decoding

We assume that the input base64 data is encoded in ASCII or some ASCII superset such as UTF-8 or ISO 8859-1. In such cases, any non-ASCII character should trigger an error. If there are white-space characters (e.g., ' ', '\n' or '\r'), then they must be removed prior to decoding (see Appendix B).

Algorithm 2 illustrates a decoding procedure. The main loop of the algorithm consists of two steps, we first map ASCII characters to 6-bit integer values:

- $a \leftarrow A(C_i)$,
- $b \leftarrow A(C_{i+1})$,
- $c \leftarrow A(C_{i+2})$,
- $d \leftarrow A(C_{i+3})$.

And then we compute the three output byte values:

- $(a \times 4) + (b \div 16)$,
- $(b \times 16) \bmod 256 + (c \div 4)$,
- $(c \times 64) \bmod 256 + d$.

The function A can be implemented as a table lookup.

Example 2.2. Let us consider the following base64 code R0lGODlhaQABAIAAAP///wAAACwAAAAAAQABAAACakQBADs=. It represents a small gif image. To decode it, we may look up each character for the base64 code (except the terminating '=') in Table I and map it to its corresponding value in $[0, 64)$: 17, 52, 37, 6, 14, 3, 37, 33, 0, 16, 0, 1, 0, 8, 0, 0, 0, 15, 63, 63, 63, 48, 0, 0, 0, 2, 48, 0, 0, 0, 0, 0, 16, 0, 1, 0, 0, 0, 2, 0, 36, 16, 1, 0, 3, 44. Then we take each block of 4 consecutive values a, b, c, d

and map them to $(a) + (b \div 16), (b \times 16) \bmod 256 + (c \div 4), (c \times 64) \bmod 256 + d$. We must pay attention to the fact that we are missing one value in the last block because of the terminating '=' which means that, in the processing of this last block, we must produce only two values, instead of three. The result is the byte values: 71, 73, 70, 56, 57, 97, 1, 0, 1, 0, 128, 0, 0, 255, 255, 255, 0, 0, 0, 44, 0, 0, 0, 0, 1, 0, 1, 0, 0, 2, 2, 68, 1, 0, 59.

The fast decoder used by Google Chrome browser has a streamlined approach for all but the four last input characters. Instead of a single function A , it uses four distinct lookup tables (A_1, A_2, A_3, A_4) made of 256 32-bit values: $a \leftarrow A_1(C_i), b \leftarrow A_2(C_{i+1}), c \leftarrow A_3(C_{i+2}), d \leftarrow A_4(C_{i+3})$. Normally, only three of four bytes of each of these 32-bit values are used, with the remaining byte set to zero. However, whenever an illegal character is encountered, the extra byte is used as a flag. We compute the bitwise OR of the four 32-bit values (a, b, c, d) : $z = a \vee b \vee c \vee d$. We choose A_1, A_2, A_3, A_4 so that three bytes of the 32-bit value z are the decoded bytes. In effect, this approach decodes 4 input ASCII characters using four lookup and three bitwise OR, not counting a test for illegal characters. The details depend on whether the hardware has a big endian or little endian architecture. This approach uses more memory, but it results in fewer operations and potentially higher speed.

3. ADVANCED X64 INSTRUCTIONS AND INTRINSICS

Commodity x64 processors benefit from several advanced instructions. Recent compilers can automatically make use of them without any intervention from the programmer.

However, our experience is that programmers who tune their code to make explicit use of specific advanced instructions can often see significant performance benefits. When programming in C and C++, many advanced instructions are available through special functions called intrinsics (see Table II). Intrinsics enable programmers to tailor their code to the microarchitecture of their processor without writing assembly code. Intrinsics are supported by most C/C++ compilers on x64 platforms including GNU GCC, the Intel C++ Compiler, Microsoft Visual Studio and LLVM's compilers. We find it convenient to express our algorithms using intrinsics:

- Intrinsics are portable across a wide range of compilers.
- They allow us to write all of our code in a single language (C).
- They leave technical details such as the assignment of registers to the compiler.

Many of the SIMD instructions and intrinsics are straightforward. For example, the `_mm_or_si128` intrinsic (and its corresponding `por` instruction) takes two 128-bit registers and outputs a new 128-bit register made of the bitwise OR of the inputs.

The first vector instructions on x64 processors used 128-bit vectors (starting with the Pentium 4). Recent commodity x64 processors from Intel (Haswell, Broadwell, Skylake, Kaby Lake) and AMD (Carrizo, Ryzen) have 256-bit vectors with the AVX and AVX2 instructions sets.¹ However, these vectors should be regarded as pairs of 128-bit vectors (each of them called a "lane"). Indeed, most AVX/AVX2 instructions cannot move data from one 128-bit lane to another. One exception is the `_mm256_permutevar8x32_epi32` intrinsic which we may use to move or copy 32-bit words from any location in the 32-byte vector to any other.

The `pshufb` instruction shuffles the input bytes into a new vector containing the same byte values. Given an input register v and a control mask m , as vectors of sixteen bytes, it outputs a new vector $(v_{m_0}, v_{m_1}, v_{m_2}, v_{m_3}, \dots, v_{m_{15}})$ (assuming that $0 \leq m_i < 16$ for $i = 0, 1, \dots, 15$).² The AVX2 instruction set contains an upgraded version of this instruction

¹Intel released its first AVX2 processor in 2013 using its Haswell microarchitecture.

²ARM Neon has similar instruction `vtbl`, AltiVec similarly defines `vperm`.

(`vpshufb` with the intrinsic `_mm256_shuffle_epi8`) that does the same shuffling operation inside each of the two 16-byte lanes of a 32-byte register.

Though we may see the `pshufb` and `vpshufb` instructions as “shuffling” instructions (moving the bytes around), another useful interpretation is that of a table lookup. That is, with the function call `_mm256_shuffle_epi8(a,b)`, we can effectively treat `a` as a lookup table, and `b` as 4-bit indexes. Hence, we can use `pshufb` and `vpshufb` to implement maps with the benefit that there are only two registers involved and no other memory access. Moreover, these instructions are inexpensive: they have a throughput of one instruction per cycle on recent Intel processors.

The vectors instructions we use, including `vpshufb`, have low latency (usually 1 CPU cycle) and a high throughput [Fog 2016]. Depending on the compilation settings, especially optimization level and the selected CPU target, a compiler might emit different sequences of instructions to express a given sequence of intrinsics. The intrinsics used to initialize vectors—such as `_mm256_set1_epi16`—can be implemented by the compiler as a load instruction (`vmovdqa`). However, within a loop, vector initializations are often optimized away when the compiler recognizes the resulting vector as a constant. The vector might then be initialized once into a vector register that gets reused. The AVX2 instruction set has access to sixteen 32-byte (YMM) registers.

4. VECTORIZED BASE64

Encoding and decoding base64 data involves mapping blocks of three bytes to blocks of four ASCII characters. We usually decode data by accessing each character, one by one.

For better speed, we may want to decode data using entire vector registers. Depending on the largest available vector length, we may be able to decode base64 data in sets of 16 (e.g., SSE, ARM Neon, Power AltiVec), 32 (e.g., AVX/AVX2) or even 64 (e.g., AVX-512) characters at once. Similarly, we may want to encode base64 data in blocks corresponding to vector registers.

Recent x64 processors have 32-byte vector registers (AVX2/AVX). Thus, for better speed on this popular platform, we may want to read blocks of 32 characters and transform them into 24 decoded bytes in one step, and *vice versa*.

We can vectorize the encoding in the following manner:

- (1) Load 24 new input bytes in a 32-byte register.
- (2) Unpack the 24 bytes into thirty-two 6-bit values.
- (3) Map each of the thirty-two 6-bit values, into its corresponding ASCII character (see Table I).
- (4) Store the resulting 32 bytes.

The decoding proceeds in reverse:

- (1) Load 32 bytes, treating them as ASCII characters.
- (2) Map each of the 32 ASCII characters to its corresponding 6-bit value.
- (3) Pack the thirty-two 6-bit values into 24 bytes (within a 32-byte register).
- (4) Store the 24 new bytes.

This approach assumes that the original data encoded in base64 is divisible by 24 bytes. However, we can process the remaining data using one of the scalar algorithms from § 2.

To our knowledge, the first attempt to vectorize base64 encoding and decoding is due to Klomp [Klomp 2014a]. It used more instructions than the streamlined approach we present, and it lacked a performance evaluation.

ASCII Characters and 6-bit Values. An important step in coding and decoding base64 data is to map 6-bit values (in $[0, 64)$) to ASCII characters. Scalar code (see § 2) often

Table II: Intel intrinsics and instructions on x64 processors.

intrinsic	bits	instruction	description
<code>_mm_set1_epi8</code>	128	—	create a vector containing 16 identical bytes
<code>_mm_or_si128</code>	128	<code>por</code>	bitwise OR
<code>_mm256_or_si256</code>	256	<code>vpor</code>	bitwise OR
<code>_mm_and_si128</code>	128	<code>pand</code>	bitwise AND
<code>_mm256_and_si256</code>	256	<code>vpand</code>	bitwise AND
<code>_mm_cmpeq_epi8</code>	128	<code>pcmpeqb</code>	compare 16 pairs of bytes, outputting 0xFF on equality and 0x00 otherwise
<code>_mm256_cmpeq_epi8</code>	256	<code>vpcmptb</code>	compare 32 pairs of bytes, outputting 0xFF on equality and 0x00 otherwise
<code>_mm_movemask_epi8</code>	128	<code>pmovmskb</code>	construct a 16-bit integer from the most significant bits of 16 bytes
<code>_mm256_shuffle_epi8</code>	256	<code>vpshufb</code>	shuffle two lanes of 16 bytes
<code>_mm_popcnt_u64</code>	64	<code>popcnt</code>	return the number of 1s in a 64-bit word (population count)
<code>_mm_loadu_si128</code>	128	<code>movdqu</code>	load 16 bytes from memory into a vector register
<code>_mm256_loadu_si256</code>	256	<code>vmovdqu</code>	load 32 bytes from memory into a vector register
<code>_mm256_set_epi8</code>	256	—	load 32 specified bytes into a vector register
<code>_mm256_setr_epi8</code>	256	—	load 32 specified bytes into a vector register (listed in reverse order)
<code>_mm256_set1_epi8</code>	256	—	create a vector register with the same 8-bit word repeated 32 times
<code>_mm256_set1_epi32</code>	256	—	create a vector register with the same 32-bit word repeated four times
<code>_mm256_maskload_epi32</code>	256	<code>vpmaskmovd</code>	load 32 bytes from memory into a vector register, omitting values some values according to the provided mask
<code>_mm_storeu_si128</code>	128	<code>movdqu</code>	write a 16-byte vector register to memory
<code>_mm256_storeu_si256</code>	256	<code>vmovdqu</code>	write a 32-byte vector register to memory
<code>_mm256_mulhi_epu16</code>	256	<code>vpmulhw</code>	multiply 16-bit integers, keeping the high 16 bits of the result
<code>_mm256_mullo_epi16</code>	256	<code>vpmullw</code>	multiply 16-bit integers, keeping the low 16 bits of the result
<code>_mm256_subs_epu8</code>	256	<code>vpsubusb</code>	subtract 8-bit unsigned integers
<code>_mm256_add_epi8</code>	256	<code>vpaddb</code>	add 8-bit integers
<code>_mm256_srli_epi32</code>	256	<code>vpsrld</code>	shift 32-bit words right by x bits
<code>_mm256_testz_si256</code>	256	<code>vptest</code>	AND the two inputs and return 1 if the result is zero
<code>_mm256_subs_epu8</code>	256	<code>vpsubusb</code>	Subtract 8-bit unsigned integers, setting zero when the result would be negative
<code>_mm256_maddubs_epi16</code>	256	<code>vpmaddbsw</code>	Vertically multiply pairs of 8-bit integers, producing intermediate 16-bit integers. Horizontally add adjacent pairs of intermediate signed 16-bit integers.
<code>_mm256_madd_epi16</code>	256	<code>vpmaddwd</code>	Multiply pairs of 16-bit integers, producing intermediate signed 32-bit integers. Horizontally add adjacent pairs of intermediate 32-bit integers.
<code>_mm256_permutevar8x32_epi32</code>	256	<code>vpermd</code>	Shuffle 32-bit integers in across lanes.

Table III: Offsets used in translation between 6-bit integers in $[0, 64)$ and ASCII values: e.g., to convert integers in $[0, 25]$ to ASCII code, we must add 65.

6-bit value	ASCII range	offset (6-bit to ASCII)
0 ... 25	A ... Z	65
26 ... 51	a ... z	71
52 ... 61	0 ... 9	-4
62	+	-19
63	/	-16

represents and computes these maps using pre-calculated tables. We could proceed in a similar manner for vectorized code. Given a vector register of 32 bytes, for example, we might have to look up 32 values. For better speed in a vectorized setting, we use the fact that the base64 standard uses ASCII codes spanning five continuous ranges of values corresponding to the upper-case characters ('A' to 'Z'), the lower-case characters ('a' to 'z'), the ten digits ('0' to '9') and the characters '+' and '/'. See Table III. This observation allows us to replace table lookups with some arithmetic and logical operations, and a few `vpshufb` instructions. Because vector instructions can operate on large blocks of bytes, arithmetic and logical instructions are amortized and relatively inexpensive.

4.1. Vectorized Encoding

There are two steps in the vectorized encoding:

- (1) Unpacking the 24 bytes into thirty-two 6-bit values.
- (2) Translating each of the thirty-two 6-bit values, into its corresponding ASCII character.

4.1.1. Unpacking Procedure. During the encoding, we load 24 bytes into a 32-byte AVX2 register. Toward the end of the stream, if there are fewer than 32 bytes left, we can use a masked load (with the `_mm256_maskload_epi32` intrinsic) to avoid exceeding memory bounds.

Unpacking 6-bit values into separate bytes is done in two steps.

- We move each of the eight 3-byte chunks of four 6-bit values into separate 32-bit words.
- Within each 32-bit word, we move the 6-bit values into separate bytes.

We use the byte-shuffling instruction `vpshufb` to move 3-byte chunks into separate 32-bit words. Unfortunately the instruction separates shuffling within halves (*lanes*) of the register which may cause difficulties (see § 3). Indeed, suppose that we were to load the eight 3-byte chunks in a 32-byte vector as `[|----|----|HHHG|GGFF|FEEE|DDDC|CCBB|BAAA]` where A...H denotes bytes from different chunks and the dashes denote unused bytes. We would then need have two 16-byte lanes containing different numbers of 3-byte chunks, with one 3-byte chunk (FFF) overlapping two lanes: `[|FEEE|DDDC|CCBB|BAAA]` and `[|----|----|HHHG|GGFF|]`. To avoid this problem, we load the 24-byte data with an offset 4 bytes, thanks to that a register contains four 3-byte chunks in each lane, as shown on Fig. 1; the first 4 bytes as well as the last 4 bytes (out of 32 bytes) are to be discarded. The code with intrinsics is given in Fig. 3.

There are two main steps.

- As previously stated, we reshuffle each of the 16-byte lanes with the `vpshufb` so that 3-byte chunks go into separate 32-bit words. But we want the bytes to fall in a specific

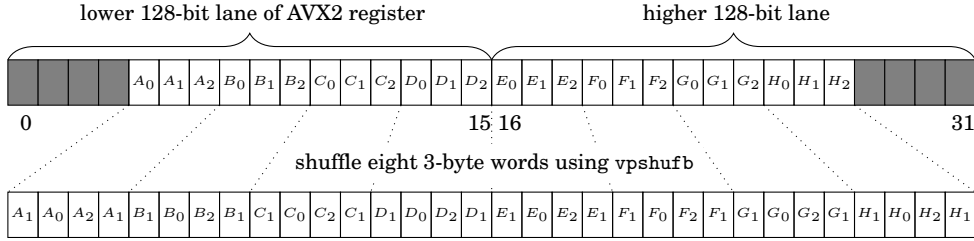


Fig. 1: Encoding: loading and shuffling 24 input bytes within a 32-byte register within two 16-byte lanes

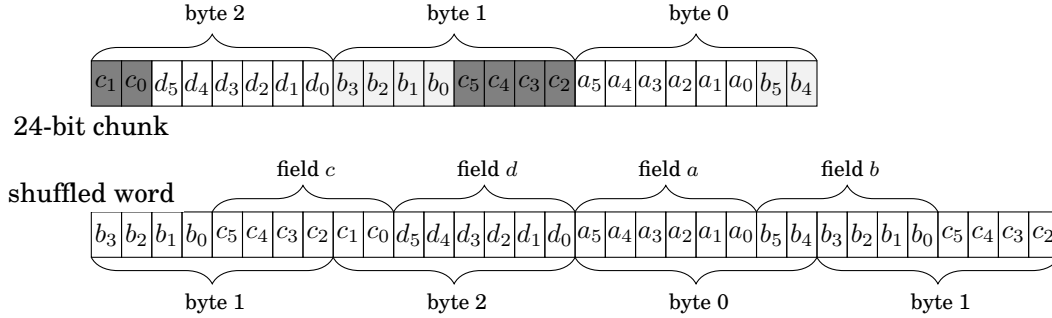


Fig. 2: Encoding: from a packed 24-bit chunk, we generate a shuffled 32-bit word by repeating byte 1.

order. See Fig. 2. This choice enables us to finish the bit shuffling using inexpensive arithmetic and logical operations. This byte-shuffling code is illustrated in Fig. 3.

- It remains to unpack the four 6-bit values a, b, c, d (see Fig. 4). Treating the four bytes as a 32-bit integer, we can isolate the 6-bit values c and a with a bitwise mask (leaving all other values zero). We then need to shift these bits right by 10 and 6 respectively. Because AVX2 lacks a 16-bit variable shift instruction, we use the `vpshufb` instruction (`_mm256_shuffle_epi8`) that multiplies pairs of 16-bit numbers while storing the most significant 16 bits of the result. Likewise, we isolate b and d and shift them left by 4 and 8 respectively. To achieve this shift, we use the multiplication instruction `vmulhw` (`_mm256_mulhi_epi16`), which multiplies pairs of 16-bit numbers while storing the least significant 16 bits of the product.

Finally, we merge partial results using a bitwise OR.

4.1.2. ASCII Translation. Once we have all 6-bit values in separate bytes, we need to convert them into ASCII characters. At a high-level, what we need to do is to take the 6-bit value and add the corresponding “offset” value from Table III. There are five distinct offset values. For example, if our 6-bit value lies in the interval $[0, 25]$, we want to add 65 to it. Yet we want to avoid a potentially expensive lookup.

Instead of using a table lookup, we use a vectorized approach, see Fig. 5. The main ingredient is the `vpshufb` instruction (`_mm256_permute_epi8`), which does a parallel lookup in a destination register using the lower four bits of bytes from a source register. Unlike a conventional table lookup, there is little chance of expensive cache misses since the instruction can take two register values. To make use of the `vpshufb` instruction, we

```

__m256i enc_reshuffle(__m256i input) {
    __m256i in = _mm256_shuffle_epi8(input, _mm256_set_epi8(
        10, 11, 9, 10, 7, 8, 6, 7, 4, 5, 3, 4, 1, 2, 0, 1,
        14, 15, 13, 14, 11, 12, 10, 11, 8, 9, 7, 8, 5, 6, 4, 5
    ));
    __m256i t0 = _mm256_and_si256(in, _mm256_set1_epi32(0x0fc0fc00));
    __m256i t1 = _mm256_mulhi_epu16(t0, _mm256_set1_epi32(0x04000040));
    __m256i t2 = _mm256_and_si256(in, _mm256_set1_epi32(0x003f03f0));
    __m256i t3 = _mm256_mullo_epi16(t2, _mm256_set1_epi32(0x01000010));
    return _mm256_or_si256(t1, t3);
}

```

Fig. 3: Encoding: mapping 24 input bytes into thirty-two 6-bit values stored in distinct bytes

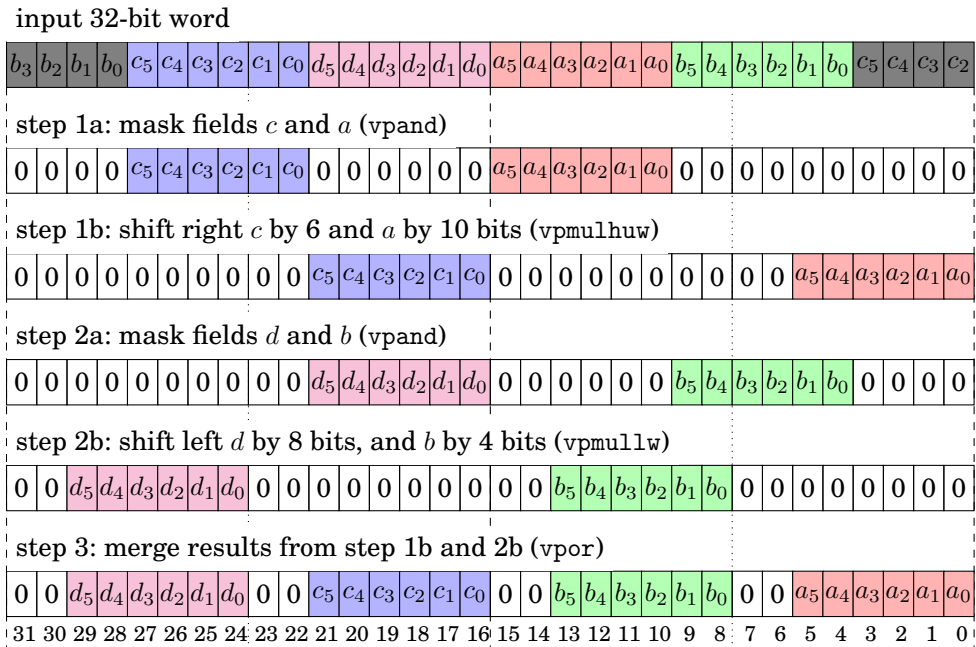


Fig. 4: Encoding: arithmetic and logical operations to unpack four 6-bit values to individual bytes from a shuffled input 32-bit word

Table IV: Input reduction for encoding

6-bit value	reduced	offset
0...25	13	65
26...51	0	71
52...61	1...10	-4
62	11	-19
63	12	-16

```

__m256i toascii(__m256i input) {
    __m256i result = _mm256_subs_epu8(input, b51);
    __m256i less = _mm256_cmpgt_epi8(b26, input);
    result = _mm256_or_si256(result, _mm256_and_si256(less, b13));
    __m256i offsets = _mm256_setr_epi8(
        65, -4, -4, -4, -4, -4, -4, -4,
        -4, -4, -4, -19, -16, 71, 0, 0,
        65, -4, -4, -4, -4, -4, -4, -4,
        -4, -4, -4, -19, -16, 71, 0, 0
    );
    result = _mm256_shuffle_epi8(offsets, result);
    return _mm256_add_epi8(result, input);
}

```

Fig. 5: Encoding: function translating byte values in $[0, 64)$ to ASCII characters according to the base64 standard. We use the convention that `bx` is `_mm256_set1_epi8(x)`.

first have to reduce our 6-bit values to 4-bit values. It suffices to map the 6-bit values to their reduced value in Table IV. For example, we want all values in $[0, 25]$ to be mapped to 13, all values in $[26, 51]$ to be mapped to 0, and so forth.

For our purposes, we use a *saturated subtraction* (`_mm256_subs_epu8`) of unsigned values: it yields 0 whenever the result would be less than zero; it might be expressed as $\max(x - y, 0)$. The hardware implementation is as fast as a regular subtraction.

By using saturated subtract with value 51, we reduce input values from ranges $[0, 25]$ and $[26, 51]$ into a single value 0. The values in $[52, 63]$ are mapped to $[1, 12]$. Then a comparison identifies input values that are less than 26, and adjusts reduced input by assigning code 13 to the range $[0, 25]$. Finally, an invocation of `vpshufb` translates the reduced input into offset values.

4.2. Vectorized Decoding

Our vectorized decoding algorithm is analogous to our encoding algorithm. We decompose decoding in two steps:

- (1) Map each of the 32 ASCII characters to its corresponding 6-bit value.
- (2) Pack the thirty-two 6-bit values into 24 bytes (within a 32-byte register).

Mapping requires both translation from ASCII characters to 6-bit values and verification that all input characters are valid. Treating the ASCII characters as a sequence of byte values, both steps can be efficiently achieved by analyzing the lower and higher nibbles (4-bit halves) of each ASCII character.³ Our approach involves only three lookups in AVX2 vectors—each implemented with a single `vpshufb`—and a few inexpensive

³The *lower nibble* of a byte is made of the least significant 4 bits whereas the *higher nibble* is made of the most significant 4 bits.

vector instructions: a shift, a comparison, a bitwise AND, two additions and a test. To make the input character validation inexpensive, we use a bitset-based approach.⁴

Packing is performed on a vector of 6-bit values passed from the previous step (if no errors were found). This step requires just four instructions. First, we shuffle individual bits within 32-bit words using two instructions, forming 24-bit words. Then we change order of individual bytes within the lanes of an AVX2 register with a call to `vpshufb`. And we finally pack words within lanes into a continuous 24-byte array, which is written to memory.

4.2.1. ASCII Translation. Input bytes are classified to one of five ranges to select the appropriate *offset*. As per the base64 specification, we have to report errors, defined as a character outside Table I. Invalid input characters yield an offset of zero, which indicates errors. Once we have a vector of offsets, a single subtraction of that vector and an input vector yields decoded data.

Decoding ASCII characters to 6-bit values would be easier if we did not need to check for invalid characters. Indeed, if we shift ASCII byte values right by 4 bits, we get

- 2 for characters + and /.
- 3 for characters in 0 . . . 9;
- either 4 or 5 for characters in A . . . Z (4 for A . . . 0 and 5 for P . . . Z);
- either 6 or 7 for characters in a . . . z (6 for a . . . o and 7 for p . . . z);

Thus we can almost derive the offset from the most significant 4 bits. To be able to seek the right offset, we have to distinguish between the characters '+' and '/'. For this purpose, we can check whether the ASCII character is equal to '/' (byte value 0x2F). Table V gives the desired offset given the most significant 4 bits and least significant 4 bits of an ASCII value.

ALGORITHM 3: Base64 decoding for a single ASCII character. The operation AND represents the bitwise logical AND.

Require: an ASCII characters x , with byte value (code point) $\text{ord}(x)$

- 1: $\text{lut_lo} \leftarrow \{21, 17, 17, 17, 17, 17, 17, 17, 17, 17, 19, 26, 27, 27, 27, 26\}$ (constant)
- 2: $\text{lut_hi} \leftarrow \{16, 16, 1, 2, 4, 8, 4, 8, 16, 16, 16, 16, 16, 16, 16, 16\}$ (constant)
- 3: $\text{roll} \leftarrow \{0, 16, 19, 4, -65, -65, -71, -71, 0, 0, 0, 0, 0, 0, 0\}$ (constant)
- 4: $h \leftarrow \lceil \text{ord}(x)/16 \rceil$
- 5: $c \leftarrow -1$ if x is '/' and 0 otherwise
- 6: **if** $(\text{lut_lo}_{\text{ord}(x) \bmod 16} \text{ AND } \text{lut_hi}_h)$ is non-zero **then**
- 7: we have an invalid character
- 8: **end if**
- 9: **return** $\text{ord}(x) + \text{roll}_{h+c}$

Because we need to ensure that only valid ASCII characters are encountered, our algorithm is slightly longer. We give the simplified version that processes a single character in Algorithm 3 and an example of C source code in Fig. 6. The main difficulty in Algorithm 3—and in our vectorized C code—is to check that the character is part of the expected base64 characters. To solve this problem efficiently, we observe that the byte value of an allowed base64 character as per Table I must be such that:

- Its high nibble must be 2, 3, 4, 5, 6 or 7.

⁴A bitset is a standard technique to represent any set made of n possible elements using a word of n bits. If the i^{th} element is present in the set, we set the i^{th} bit to 1, otherwise we set it to 0.

Table V: Mapping between lower/higher nibble of input bytes and offsets: each column corresponds to a valid most significant 4-bit value (nibble) for a base64 ASCII value whereas each row corresponds to a least significant 4-bit. The table provides the corresponding negated offset from Table III. Most significant 4-bit values outside the range [2, 7] do not correspond to valid base64 ASCII characters. For each column, we provide the matching ASCII characters.

ASCII bitset	2 + and / 0x01	3 0...9 0x02	4 A...0 0x04	5 P...Z 0x08	6 a...o 0x04	7 p...z 0x08	{0,1} ∪ [8,15] none 0x10	bitset
0	-	4	-	-65	-	-71	-	0x15
1	-	4	-65	-65	-71	-71	-	0x11
2	-	4	-65	-65	-71	-71	-	0x11
3	-	4	-65	-65	-71	-71	-	0x11
4	-	4	-65	-65	-71	-71	-	0x11
5	-	4	-65	-65	-71	-71	-	0x11
6	-	4	-65	-65	-71	-71	-	0x11
7	-	4	-65	-65	-71	-71	-	0x11
8	-	4	-65	-65	-71	-71	-	0x11
9	-	4	-65	-65	-71	-71	-	0x11
10	-	-	-65	-65	-71	-71	-	0x13
11	19	-	-65	-	-71	-	-	0x1A
12	-	-	-65	-	-71	-	-	0x1B
13	-	-	-65	-	-71	-	-	0x1B
14	-	-	-65	-	-71	-	-	0x1B
15	16	-	-65	-	-71	-	-	0x1A

```

bool fromascii(__m256i str, __m256i * out) {
    __m256i hi_nibbles = _mm256_srli_epi32(str, 4);
    __m256i lo_nibbles = _mm256_and_si256(str, mask_2F);
    __m256i lo = _mm256_shuffle_epi8(lut_lo, lo_nibbles);
    __m256i eq_2F = _mm256_cmpeq_epi8(str, mask_2F);
    hi_nibbles = _mm256_and_si256(hi_nibbles, mask_2F);
    __m256i hi = _mm256_shuffle_epi8(lut_hi, hi_nibbles);
    __m256i roll = _mm256_shuffle_epi8(lut_roll,
        _mm256_add_epi8(eq_2F, hi_nibbles));
    if (!_mm256_testz_si256(lo, hi)) {
        return false;
    }
    *out = _mm256_add_epi8(str, roll);
    return true;
}

```

Fig. 6: Decoding: from ASCII characters to 6-bit values. The function returns false on error. The vectors lut_lo, lut_hi, lut_roll and mask_2F are provided in the main text.

— When its high nibble is 2, the low nibble must be 11 or 15. When its high nibble is 3, the low nibble must be in [0, 9]. When the high nibble is 4 or 6, its low nibble must be non-zero. When the high nibble is 5 or 7, the low nibble must be in [0, 10].

Moreover, these conditions are necessary and sufficient for a base64 character to be valid. See Table V. To check quickly that a character is valid, we use a bitset approach to compare the low and high nibble values. It is implemented in Algorithm 3 with the arrays lut_lo and lut_hi, and a bitwise AND. We refer the reader to Appendix C for a detailed description of the bitset approach and of our vectorized code.

four input indices

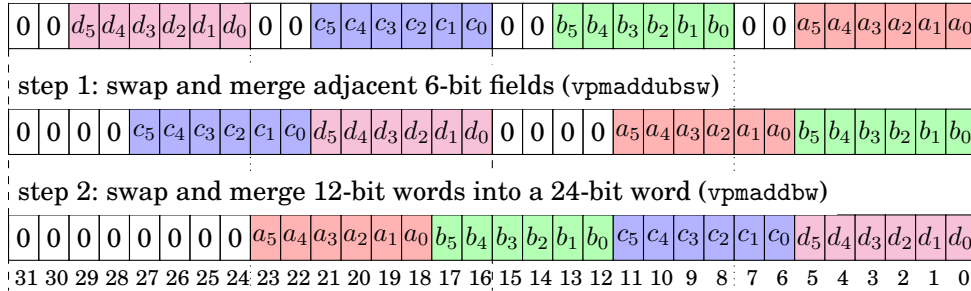


Fig. 7: Decoding: arithmetic operations to pack four 6-bit values stored in separate bytes to a continuous (packed) 24-bit subword of each 32-bit word

4.2.2. *Packing Procedure.* After translation, we have 32 values, that must be saved as a 24-byte array. We pack the 6-bit values using only four instructions.

- (1) We use `vpmaddubsw` to pack the data within 16-bit words.
- (2) We use `vpmaddwd` to pack the data within 32-bit words.
- (3) Then we use `vpshufb` to pack within 128-bit lanes.
- (4) Finally, we pack our 24 bytes within a 256-bit vector using `vpermd`.

We first review how the packing proceeds within 32-bit words (see Fig. 7).

- We initially have four 6-bit values stored in each of the four bytes, with zeros elsewhere $[00d_5d_4d_3d_2d_1d_0|00c_5c_4c_3c_2c_1c_0|00b_5b_4b_3b_2b_1b_0|00a_5a_4a_3a_2a_1a_0]$.
- We call the `vpmaddubsw` (`_mm256_maddubs_epi16`) instruction providing as inputs our data as well as the vector made of the byte values `0x01`, `0x40`, `0x01`, `0x40`, \dots . This instruction multiplies pairs of 8-bit integers from its two input vectors, producing intermediate 16-bit integers, it then adds adjacent pairs of 16-bit integers (the first two, the third and the fourth, \dots). In our case, we multiply alternatively by `0x40` (a shift by 6 bits) and `0x01` (the identity). By inspection, the output is $[0000c_5c_4c_3c_2|c_1c_0d_5d_4d_3d_2d_1d_0|0000a_5a_4a_3a_2|a_1a_0b_5b_4b_3b_2b_1b_0]$.
- We then call the `vpmaddwd` (`_mm256_madd_epi16`) instruction. It is similar to the `vpmaddubsw` instruction: it multiplies pairs of 16-bit integers from its two input vectors, producing intermediate 32-bit integers, it then adds adjacent pairs of 32-bit integers (the first two, the third and the fourth, \dots). We call it with the vector made of the 16-bit values `0x0001`, `0x1000`, \dots so that we alternatively shift by 12 bits or compute the identity. By inspection, the result within each 32-bit word is $[00000000|a_5a_4a_3a_2a_1a_0b_5b_4|b_3b_2b_1b_0c_5c_4c_3c_2|c_1c_0d_5d_4d_3d_2d_1d_0]$.

We have effectively packed four 6-bit value to 3 bytes, within each 32-bit word, using only two instructions.

Within each 128-bit lane, we can then shuffle the bytes to pack the data bytes at the beginning of the lanes using `vpshufb` (`_mm256_shuffle_epi8`). We shall have three 32-bit word filled with data within each lane. Though normally we cannot move data between lanes, the `vpermd` (`_mm256_permutevar8x32_epi32`) instruction is a useful exception. It can shuffle 32-bit words across lanes, allowing us to get the final result. Fig. 8 gives a summary overview of how the data is moved within an AVX2 register. We provide our C implementation in Fig. 9.

AVX2 register: 32 input 6-bit values stored in separate bytes

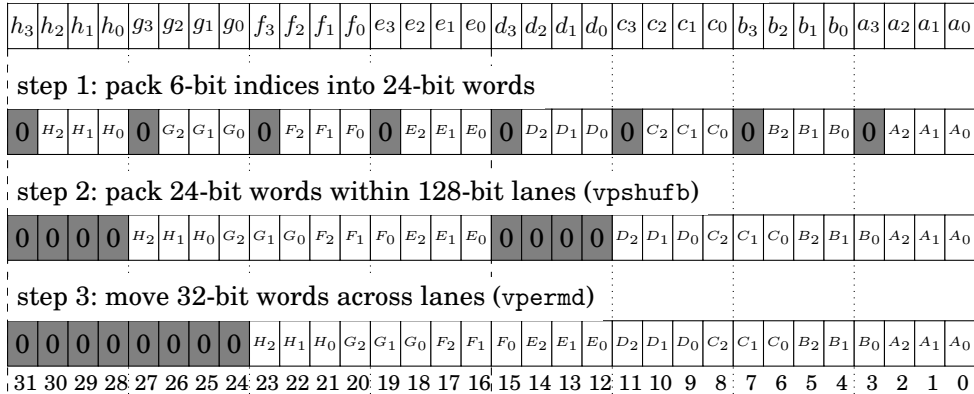


Fig. 8: Decoding: steps required to convert from 32 6-bit indices into a continuous (packed) array of 24 bytes

```

__m256i dec_reshuffle(__m256i in) {
    __m256i merge_ab_and_bc = _mm256_maddubs_epi16(in,
        _mm256_set1_epi32(0x01400140));
    __m256i out = _mm256_madd_epi16(merge_ab_and_bc,
        _mm256_set1_epi32(0x00011000));
    out = _mm256_shuffle_epi8(out, _mm256_setr_epi8(
        2, 1, 0, 6, 5, 4, 10, 9, 8, 14, 13, 12, -1, -1, -1,
        2, 1, 0, 6, 5, 4, 10, 9, 8, 14, 13, 12, -1, -1, -1, -1
    ));
    return _mm256_permutevar8x32_epi32(out, _mm256_setr_epi32(0, 1,
        2, 4, 5, 6, -1, -1));
}

```

Fig. 9: Decoding: packing the 6-bit values into full bytes

5. PERFORMANCE EVALUATION

We implemented our software in C. All tested code has error handling in the sense that unexpected characters are detected. We use a Linux server with an Intel i7-6700 processor running at 3.4 GHz. This Skylake processor has 32 kB of L1 data cache and 256 kB of L2 cache per core with 8 MB of L3 cache. The machine has 32 GB of RAM (DDR4 2133, double-channel). There is no disk access during our tests. We disabled Turbo Boost and set the processor to run at its highest clock speed. We used the processor’s time stamp counter (rdtsc instruction [Paoloni 2010]) to estimate the number of cycles. Our software is freely available (<https://github.com/lemire/fastbase64>) and was compiled using the GNU GCC 5.3 compiler with the “-O3 -march=native” flags.

In addition to our own implementation, we also use the Linux base64 codec (from the 4.10 release), the base64 codec from the QuickTime Streaming Server⁵, as well as the base64 codec used within the Google Chrome browser (part of Chromium release 56.0.2924.87) [Galbreath 2016]. In a cross-browser comparison, Nägele found that the Chrome browser has the best base64 performance [Nägele 2015]. All of the codecs check for invalid characters. All codecs assume that the input is free from white-space

⁵<https://opensource.apple.com/source/QuickTimeStreamingServer/QuickTimeStreamingServer-452/CommonUtilitiesLib/base64.c>

Table VI: Decoding performance in CPU cycles per input Base64 byte. We include the number of cycles per byte required to memcpy the base64-encoded data.

Source	bytes	memcpy	Linux	QuickTime	Chrome	AVX2(Klomp)	AVX2
lena [jpg]	141 020	0.09	20	3.1	1.8	0.43	0.21
peppers [jpg]	12 640	0.03	15.5	3.1	1.8	0.44	0.21
mandril [jpg]	329 632	0.11	20	3.1	2.1	0.43	0.21
Moby Dick [text]	1484	0.04	3.6	3.2	1.8	0.53	0.27
google logo [png]	2357	0.05	4.2	3.1	2.1	0.47	0.23
bing social icons [png]	1355	0.04	3.9	3.2	2.1	0.46	0.23

characters, except that the Linux codec decoder ignores line feed characters (`\n`) within the base64 encoded stream as long as they occur between two blocks of four characters. Klomp produced a library that supports SIMD-accelerated decoding [Klomp 2014b]. The AVX2 code from his library is similar to ours, but it includes more instructions and it is slower ($\approx 2\times$).

To test on realistic data, we included standard images from image processing (Lena, peppers and mandril), a short quote from the novel Moby Dick (“Call me Ishmael (...) the same feelings towards the ocean with me.”), a Google logo found to be base64 encoded in the Google search page and a set of icons found base64 encoded in the Bing search page. To ensure reproducibility, all our test data and benchmarking software is available along with our codecs.

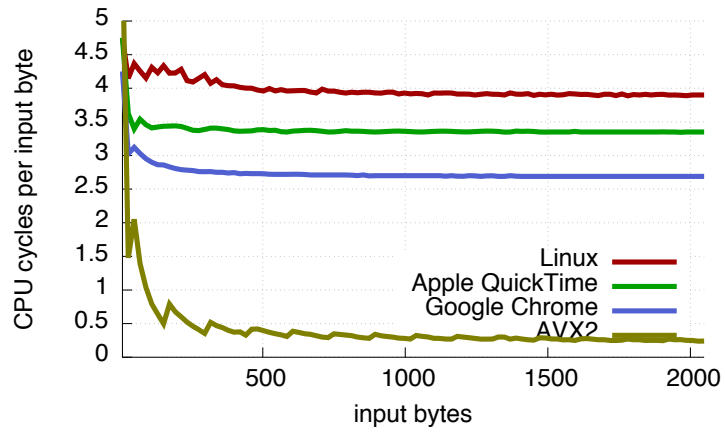
Table VI presents our results in number of cycles per input byte, for single-threaded execution. To make sure our results are reliable, we repeat each test 500 times and check that the minimum and the average cycle counts are within 1% of each other. We report the minimum cycle count divided by the number of bytes in the input. Our AVX2 decoder is nearly an order of magnitude faster than the fastest of the conventional decoder (Chrome).

To appreciate the performance of the encoding and decoding given inputs having various lengths, we generated random binary data and benchmarked the time required to encode and decode it (see Fig. 10). For sufficiently large inputs, the AVX2 codec uses about a quarter of a cycle per input byte during encoding and decoding. The fastest alternative codec (from Google Chrome) uses about 2.7 cycles per input byte for encoding and about 1.8 cycles per input byte for decoding. Thus the AVX2 codec is eleven times faster at encoding and over seven times faster at decoding than the fastest alternative under consideration in our tests for large inputs.

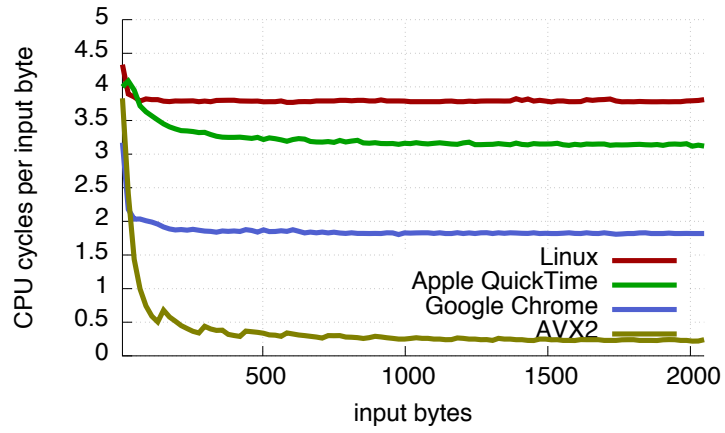
Our AVX2 codec falls back on scalar functions when there are fewer than 28 input characters left to encode, or fewer than 45 bytes left to decode. As long as we process at least ≈ 200 bytes, the AVX2 codec maintains a much higher speed because a sizeable fraction of the data is decoded using SIMD instructions. It is only for short inputs (less than 100 bytes) that the AVX2 codec loses its benefits, achieving a performance no better than scalar functions.

We also analyzed the assembly code produced by our compiler.

- For encoding 24 input bytes into 32 output bytes (to ASCII), our code uses 11 vector instructions (excluding one load and one store): two `vpshufb`, two `vpand`, one `vpor`, one `vpaddb`, one `vpmulhuw`, one `vpmullw`, one `vpsubusb`, one `vpsubb` and one `vpcmpgtb`. For long inputs, we use about 0.25 cycles per input bytes, or about 6 cycles per 24 input bytes. Thus we execute about 11 vector instructions per 6 cycles, not counting a load and a store, as well as a few scalar instructions.
- For decoding 32 input bytes (in ASCII) to 24 output bytes, we use 14 vector instructions (excluding one load and one store): four `vpshufb`, two `vpand`, one `vpsrld`,



(a) Encoding



(b) Decoding

Fig. 10: Performance comparison between base64 codecs using random inputs of various lengths

one `vpaddd`, one `vptest`, one `vpcmpq`, one `vpaddd`, one `vpmaddbsw`, one `vpmaddwd`, and one `vpermd`. For long inputs, we again use about 0.25 cycles per input bytes, or about 8 cycles per 32 input bytes. Thus execute about 14 vector instructions in 8 cycles, not counting a load and a store and some scalar instructions.

Counting the necessary load and stores, we need 13 vector instructions to encode 24 input bytes, and we need 16 vector instructions to decode 32 input bytes. Processors execute complex machine instructions using low-level instructions called μ ops, each of the vector instructions we considered counts for a single μ op [Fog 2016]. We require about 0.5 μ ops per cycle to encode or decode, using our AVX2 codec. As a comparison, we analyzed the assembly code produced by the fast decoder used by Google Chrome. To decode a single block of four bytes, it requires about 20 μ ops or about 5 μ ops per input byte. Thus we can largely explain our good results by the fact that we use fewer instructions per input byte.

6. CONCLUSION

By designing algorithms for vector instructions, we reduced the number of instructions necessary to encode or decode base64 data. Consequently, we showed that commonplace vector instructions (AVX2) can boost encoding and decoding speeds by roughly an order of magnitude. The end product spans a few dozens of lines of code and can be readily integrated into existing systems, such as browsers, runtime libraries, and servers.

A vectorized base64 codec can be written for ARM processors supporting the NEON instructions. We can preserve the fast algorithms, changing the instructions as needed (`vpshufb` becomes `vtbl` and so forth). Klomp reports a 4 \times performance gain on an iPhone [Klomp 2014b] with ARM NEON instructions. However, there is a variety of ARM hardware and software configurations. E.g., Srinivasa et al. [Srinivasa et al. 2017] found 20% performance differences between otherwise identical devices due to process variation in the manufacture of ARM CPUs. We leave a review of base64 codecs on ARM processors to future work.

Upcoming Intel processors will support some of the AVX-512 instruction sets along with 512-bit vectors.⁶ Even wider registers should allow for ever better performance. However, AVX-512 is a family of instruction sets, with the most powerful sets (e.g., AVX-512BW and AVX-512VL) enabling further optimizations. ARM processors are also expected to receive support for wider vectors (e.g., 512-bit or 2048-bit vectors) through the Scalable Vector Extensions [ARM 2017].

Having to design and support functions for several different vectorized instruction sets is cumbersome. We could ask whether we can achieve similar results with higher-level C++ libraries such as the Generic SIMD Library [Wang et al. 2014] or Boost.SIMD [Est erie et al. 2014]. Programming languages like Swift or Rust also have generic SIMD packages that could be put to good use. For comparison, we expect that calling our C implementation from other languages (e.g., Swift and Rust) would be practical and computationally efficient.

For large inputs (e.g., 36 MB), Kopp showed that a GPU implementation could encode base64 data roughly twice as fast as a fast CPU implementation [Kopp 2013]. It could be interesting to further explore the engineering of GPU codecs.

APPENDIX

A. ADVANCED ERROR CHECKING

We find it interesting that even if all encoded characters are part of our table of base64 characters, Algorithm 2 may still decode data that could not possibly have been encoded

⁶<http://0x80.pl/articles/avx512-foundation-base64.html>

in base64 [Char 2014]. Indeed, we would need to add the following additional checks on the last four ASCII characters ($C_{n-4}, C_{n-3}, C_{n-2}, C_{n-1}$) to ensure there was an original binary input:

- The stream must end with zero, one or two padding characters ('=').
- If there are two padding character, then $(A(C_{n-3}) \times 16) \bmod 256 = 0$ where A is as in Algorithm 2.
- If there is one padding character, then $(A(C_{n-2}) \times 64) \bmod 256 = 0$.

However, these checks are only needed on the last four characters so that we can add them to any existing decoder without measurably impacting the performance. The base64 standard [Josefsson 2006] does not require these checks.

B. WHITE SPACE

The base64 specification [Josefsson 2006] requires decoders to reject encoded data with unexpected characters:

Implementations **MUST** reject the encoded data if it contains characters outside the base alphabet when interpreting base-encoded data unless the specification referring to this document explicitly states otherwise.

For some applications, white-space characters may be present and should be removed prior to processing.

We can assume that the input data is encoded in ASCII, UTF-8 or other ASCII supersets (e.g., ISO 8859-1). There are three commonly used white-space characters that we might want to remove:

- the space character (' ', byte value 0x0a or 32 in decimal),
- the line-feed character ('\n', byte value 0x20 or 10 in decimal),
- the carriage-return character ('\r', byte value 0x0D or 13 in decimal).

We can write a scalar function that removes the spaces (Fig. 11). Our function works by copying the bytes in place while advancing the pointer by one byte only when the character is not a white-space character. On our test platform (see § 5), we can remove spaces in place from a small string (1 kB) containing a small fraction of randomly inserted white-space characters (3%) at a rate of 1.6 cycles per byte using this function. We can manually vectorize the removal of white-space characters for better speed (see Fig. 12). In effect, it suffices to compare all input characters with each of the possible white-space characters. We get a resulting mask (`mask16`) that indicates (with a 1-bit) which characters are white space. From this white-space mask, we can lookup a shuffling mask to reorder the input bytes so that white-space characters are omitted. We use the `_mm_popcnt_u64` intrinsic (`popcnt`), which returns the number of 1-bit contained in a 64-bit word. The illustrated code sequence removes `_mm_popcnt_u64(mask16)` white space characters from the input vector, so that the first $16 - _mm_popcnt_u64(mask16)$ bytes of the final vector (`v`) are free from white space. With this faster code, we get a processing rate between 0.3 and 0.4 cycles per input byte. We make the source code of our white-space removal software available online (<https://github.com/lemire/despacer>).

Instead of removing three specific characters (' ', '\n', '\r'), we could remove all characters with a byte value less than 33 to save a few instructions. However, there is no SIMD instruction on x64 processors providing an unsigned comparison. A sequence of at least two instructions are needed to achieve an unsigned comparison (e.g., a vectorized unsigned maximum `pmaxub` followed by an equality check).

```

for(i = 0, p = 0; i < N; i++) {
    uint8_t v = A[i];
    A[p] = v;
    p += table[v];
}

```

Fig. 11: C function to remove selected byte values from an array (A) of N bytes in place: the table array is made of 256 Boolean values (0 and 1) where value 0 indicate that the corresponding byte value is to be removed. The integer p represents the new size of the array.

```

__m128i spaces = _mm_set1_epi8(' ');
__m128i newline = _mm_set1_epi8('\n');
__m128i carriage = _mm_set1_epi8('\r');

__m128i v = ... // v contains 16 input bytes
__m128i is_spaces = _mm_cmpeq_epi8(v, spaces);
__m128i is_newline = _mm_cmpeq_epi8(v, newline);
__m128i is_carriage = _mm_cmpeq_epi8(v, carriage);
__m128i anywhite = _mm_or_si128(
    _mm_or_si128(xspaces, xnewline), xcarriage);

uint64_t mask16 = _mm_movemask_epi8(anywhite);
v = _mm_shuffle_epi8(v, table + mask16);

```

Fig. 12: Vectorized code using Intel intrinsics to remove selected byte values from an array (A) of N bytes from a vector of 16 bytes: the table array is made of 65536 vectors of 16 bytes.

C. COMPLETE DESCRIPTION OF THE VECTORIZED ASCII TRANSLATION

Given ASCII characters, we want to map all of them to a 6-bit value as per Table I, while reporting unexpected characters as errors. For this purpose, we describe the vectorized (using AVX2) version of Algorithm 3, as represented in C code by Fig. 6.

- We shift by 4 bits all of the ASCII byte values, getting values between 2 and 7 (with the `_mm256_srl_epi32` intrinsic), and store the result in `hi_nibbles`.
- Defining `mask_2F` as the vector made of the byte value 0x2F (or '/'), we compare the input data to `mask_2F` (with the `_mm256_cmpeq_epi8` intrinsic). The result is zero when the character value differs from 0x2F, and otherwise, it is 0xFF (or -1 as a signed byte integer). We store it in `eq_2F`.
- Later, we can add `hi_nibbles` and `eq_2F` (with the `_mm256_add_epi8` intrinsic) and then seek the offset (with the `_mm256_shuffle_epi8` intrinsic) from the vector `lut_roll` populated with the offset values
 - 0, 16, 19, 4, -65, -65, -71, -71, 0, 0, 0, 0, 0, 0, 0, 0,
 - 0, 16, 19, 4, -65, -65, -71, -71, 0, 0, 0, 0, 0, 0, 0, 0.
 We store the result in `roll` which contains the decoded offsets. This is enough to derive the offsets assuming that the ASCII character values were in range.
- To check that the ASCII character values are in range, we add a few steps. To be valid, the 4 most significant bits of the value of an ASCII characters should be 2, 3, 4, 5, 6 or 7. Values in the set {0, 1, 8, 9, 10, 11, 12, 13, 14, 15} are always indicative of an invalid character.
 - Otherwise, given the most significant 4 bits (nibble), only some least significant 4 bits are allowed:

- When the most significant 4 bits represent the value 2, then we must have either character '+' (byte value 0x2B) or character '/' (byte value 0x2F) so the least significant 4 bits must represent either 11 or 15 as an unsigned integer.
- When the most significant 4 bits represent the value 3, then we must have characters 0 . . . 9, with a least significant 4 bits ranging in [0, 9].
- When the most significant 4 bits have the value 4 or 6, then least significant 4 bits should be in [1, 15].
- When the most significant 4 bits have value 5 or 7, then the least significant 4 bits should be in [0, 10].

See Table V. We can represent these sets efficiently using bitsets. We start by mapping the most significant 4 bits to an 8-bit word with a single bit set: 2 becomes 0x01, 3 becomes 0x02, 4 and 6 become 0x04, 5 and 7 become 0x08, and all other values become 0x10. We represent this map with the vector `lut_hi` made of the following 32 bytes (repeating twice the same 16-byte subvector) in a function call such as `_mm256_shuffle_epi8(lut_hi, hi_nibbles)` where `hi_nibbles` is a 32-byte vector made of the most significant 4 bits:

- 0x10, 0x10, 0x01, 0x02, 0x04, 0x08, 0x04, 0x08,
0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
- 0x10, 0x10, 0x01, 0x02, 0x04, 0x08, 0x04, 0x08,
0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10.

Similarly, we map the least significant 4 bits to byte values made of the bitwise OR of the byte values corresponding to unallowed most significant 4 bits.

- For example, the value zero is mapped to 0b10101 (or 0x15) which is an indication that the most significant 4 bits are allowed to be 3, 5 and 7. That is we disallow 2 (0x01), 4 and 6 (0x04) as well as all values out of the range [2, 7] (0x10). And we have that $0x15 = 0x10 \text{ OR } 0x04 \text{ OR } 0x01$.
- The least significant 4-bit values in [1, 9] allow the most significant 4-bit values in [3, 7]. So we disallow 2 (0x01) and values outside the range [2, 7] (0x10). Thus we map them to $0x10 \text{ OR } 0x01$ which is 0x11.
- If the least significant 4-bit value is 10, then we disallow 2 and 3 (as well as values outside [2, 7]), so that the bitset is $0x10 \text{ OR } 0x02 \text{ OR } 0x01$ which is 0x13.
- If the least significant 4-bit value is 11 or 15, then we disallow 3, 5, 7 and values outside [2, 7] so that the bitset is $0x02 \text{ OR } 0x08 \text{ OR } 0x10$ which is 0x1A.
- If the least significant 4-bit value is 12, 13 or 14, then we disallow 2, 3, 5, 7 and all values outside [2, 7] so that the bitset is $0x10 \text{ OR } 0x02 \text{ OR } 0x08 \text{ OR } 0x01$ which is 0x1B.

We realize this map with the 32-byte vector `lut_lo` (repeating twice the same 16-byte subvector) and the function call `_mm256_shuffle_epi8(lut_lo, lo_nibbles)` where `lo_nibbles` is a 32-byte vector made of the least significant 4 bits:

- 0x15, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11,
0x11, 0x11, 0x13, 0x1A, 0x1B, 0x1B, 0x1B, 0x1A,
- 0x15, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11,
0x11, 0x11, 0x13, 0x1A, 0x1B, 0x1B, 0x1B, 0x1A.

To check whether a forbidden character has been detected, we need to call the intrinsic `_mm256_testz_si256` (`vptest`) which computes the bitwise AND between two vectors and returns 1 (true) if the value is zero.

Example C.1. Let us consider the following 32 input bytes: `R01G0D1hAQABAIAAAP///wAAACwAAAAA`. The ASCII characters correspond to the following byte values:

0x52, 0x30, 0x6c, 0x47, 0x4f, 0x44, 0x6c, 0x68,
0x41, 0x51, 0x41, 0x42, 0x41, 0x49, 0x41, 0x41,

0x41, 0x50, 0x2f, 0x2f, 0x2f, 0x77, 0x41, 0x41,
0x41, 0x43, 0x77, 0x41, 0x41, 0x41, 0x41, 0x41.

- (1) We can map this array of byte values to one made of the most significant 4 bits:

0x5, 0x3, 0x6, 0x4, 0x4, 0x4, 0x6, 0x6,
0x4, 0x5, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4,
0x4, 0x5, 0x2, 0x2, 0x2, 0x7, 0x4, 0x4,
0x4, 0x4, 0x7, 0x4, 0x4, 0x4, 0x4, 0x4.

We can compare the original input bytes with 0x2F (or '/'), creating a new array that has value 0xFF wherever the input character is equal to '/':

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00.

We can add this value to the most significant 4 bits (modulo 256) to get

0x5, 0x3, 0x6, 0x4, 0x4, 0x4, 0x6, 0x6,
0x4, 0x5, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4,
0x4, 0x5, 0x1, 0x1, 0x1, 0x7, 0x4, 0x4,
0x4, 0x4, 0x7, 0x4, 0x4, 0x4, 0x4, 0x4.

We can then seek these 4-bit values (treating them as indexes) in the table 0, 16, 19, 4, -65, -65, -71, -71, 0, 0, 0, 0, 0, 0, 0, 0 (e.g., index 0 gets value 0, index 1 gets value 16, ...). We get the offsets

-65, 4, -71, -65, -65, -65, -71, -71,
-65, -65, -65, -65, -65, -65, -65, -65,
-65, -65, 16, 16, 16, -71, -65, -65,
-65, -65, -71, -65, -65, -65, -65.

We can add these offsets to the input byte values to get the decoded 6-bit values:

17, 52, 37, 6, 14, 3, 37, 33,
0, 16, 0, 1, 0, 8, 0, 0,
0, 15, 63, 63, 63, 48, 0, 0,
0, 2, 48, 0, 0, 0, 0, 0.

We can compare the original input string (RO1GOD1hAQABAI AAP//wAAACwAAAAA) with Table I and observe that it is, indeed, the expected array. That is, we have successfully mapped the ASCII characters to 6-bit values in [0, 64).

- (2) We also need to check that all of the original ASCII characters are allowed. This requires two lookups.

(a) First, we take the array of the most significant 4-bit values, and, treating them as 4-bit indexes, we look them up in the table

0x10, 0x10, 0x01, 0x02, 0x04, 0x08, 0x04, 0x08,
0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10.

E.g., index 0 and 1 become 0x10, index 2 becomes 0x01, and so forth. From

0x5, 0x3, 0x6, 0x4, 0x4, 0x4, 0x6, 0x6,
0x4, 0x5, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4,
0x4, 0x5, 0x2, 0x2, 0x2, 0x7, 0x4, 0x4,
0x4, 0x4, 0x7, 0x4, 0x4, 0x4, 0x4, 0x4,

we get the array

0x8, 0x2, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4,
0x4, 0x8, 0x4, 0x4, 0x4, 0x4, 0x4, 0x4,
0x4, 0x8, 0x1, 0x1, 0x1, 0x8, 0x4, 0x4,
0x4, 0x4, 0x8, 0x4, 0x4, 0x4, 0x4, 0x4.

All integers in this array have a single bit set (i.e., they are powers of two).

(b) From the original input byte values, we can extract an array made of the least significant 4-bits:

```
0x2, 0x0, 0xC, 0x7, 0xF, 0x4, 0xC, 0x8,  
0x1, 0x1, 0x1, 0x2, 0x1, 0x9, 0x1, 0x1,  
0x1, 0x0, 0xF, 0xF, 0xF, 0x7, 0x1, 0x1,  
0x1, 0x3, 0x7, 0x1, 0x1, 0x1, 0x1, 0x1.
```

Treating these 4-bit values as indexes, we must then look them up in the array
0x15, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11,
0x11, 0x11, 0x13, 0x1A, 0x1B, 0x1B, 0x1B, 0x1A
to get

```
0x11, 0x15, 0x1B, 0x11, 0x1A, 0x11, 0x1B, 0x11,  
0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11,  
0x11, 0x15, 0x1A, 0x1A, 0x1A, 0x11, 0x11, 0x11,  
0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11.
```

It remains to take the bitwise AND of the two generated arrays, and we get the zero vector, verifying that all input characters were allowed.

ACKNOWLEDGMENTS

The authors would like to thank M. Howard, A. Klomp, and N. Kurz for their software contributions.

References

- Amazon 2015. Amazon SimpleDB. <http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/Welcome.html>. (2015). last checked in July 2017.
- Amazon 2017. Amazon DynamoDB. <http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html>. (2017). last checked in July 2017.
- ARM 2017. *The Scalable Vector Extension (SVE), for ARMv8-A*. Technical Report. ARM Holdings, Cambridge, United Kingdom. https://static.docs.arm.com/ddi0584/a/DDI0584A_a_SVE_supp_armv8A.pdf [last checked July 2017].
- David Calhoun. 2011. When to Base64 Encode Images (and When Not To). (2011). <http://davidbcalhoun.com/2011/when-to-base64-encode-images-and-when-not-to/>
- Hanson Char. 2014. A Fast and Correct Base 64 Codec. (2014). <https://aws.amazon.com/blogs/developer/a-fast-and-correct-base-64-codec/>
- Matt Crane and Jimmy Lin. 2017. An Exploration of Serverless Architectures for Information Retrieval. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval (ICTIR '17)*. ACM, New York, NY, USA, 241–244. DOI: <http://dx.doi.org/10.1145/3121050.3121086>
- Mark Davis. 2012. Unicode over 60 percent of the web. (2012). <https://googleblog.blogspot.ca/2012/02/unicode-over-60-percent-of-web.html>
- Elastic 2017. Elasticsearch Reference — Binary datatype. <https://www.elastic.co/guide/en/elasticsearch/reference/current/binary.html>. (2017). last checked in July 2017.
- Pierre Est erie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Laprest e. 2014. Boost.SIMD: Generic Programming for Portable SIMDization. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing (WPMVP '14)*. ACM, New York, NY, USA, 1–8. DOI: <http://dx.doi.org/10.1145/2568058.2568063>
- Tammy Everts. 2013. Rules for Mobile Performance Optimization. *Commun. ACM* 56, 8 (Aug. 2013), 52–59. DOI: <http://dx.doi.org/10.1145/2492007.2492024>
- Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. 1999. Hypertext Transfer Protocol – HTTP/1.1. <https://tools.ietf.org/html/rfc2616>. (1999). Internet Engineering Task Force, Request for Comments: 2616.
- Agner Fog. 2016. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Technical Report. Copenhagen University College of Engineering, Copenhagen, Denmark. http://www.agner.org/optimize/instruction_tables.pdf [last checked July 2017].
- Ned Freed and Nathaniel S. Borenstein. 1996. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. <https://tools.ietf.org/html/rfc2045>. (1996). Internet Engineering Task Force, Request for Comments: 2045.

- Nick Galbreath. 2016. Fast c-string transformations. <https://github.com/client9/stringencoders>. (2016). initially published in 2005, last checked in March 2017.
- Ian Hickson. 2016. Web Storage. <https://www.w3.org/TR/webstorage/>. (2016). W3C.
- Richard Duchatsch Johansen, Talita Cristina Pagani Britto, and Cesar Augusto Cusin. 2013. CSS Browser Selector Plus: A JavaScript Library to Support Cross-browser Responsive Design. In *Proceedings of the 22Nd International Conference on World Wide Web (WWW '13 Companion)*. ACM, New York, NY, USA, 27–30. DOI: <http://dx.doi.org/10.1145/2487788.2487797>
- Michael B. Jones, John Bradley, and Nat Sakimura. 2015. JSON Web Signature (JWS). <https://tools.ietf.org/html/rfc7515>. (2015). Internet Engineering Task Force, Request for Comments: 7515.
- Simon Josefsson. 2006. The Base16, Base32, and Base64 Data Encodings. <https://tools.ietf.org/html/rfc4648>. (2006). Internet Engineering Task Force, Request for Comments: 4648.
- Alfred Klomp. 2014a. Fast Base64 encoding/decoding with SSE vectorization. <http://www.alfredklomp.com/programming/sse-base64/>. (2014). last checked in March 2017.
- Alfred Klomp. 2014b. Fast Base64 stream encoder/decoder in C99, with SIMD acceleration. <https://github.com/aklomp/base64>. (2014). last checked in March 2017.
- Nick Kopp. 2013. Base64 Encoding on a GPU. <https://www.codeproject.com/Articles/276993/Base-Encoding-on-a-GPU>. (2013).
- John Linn. 1993. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. <https://tools.ietf.org/html/rfc1421>. (1993). Internet Engineering Task Force, Request for Comments: 1421.
- Larry Masinter. 1998. The “data” URL scheme. <https://tools.ietf.org/html/rfc2397>. (1998). Internet Engineering Task Force, Request for Comments: 2397.
- Microsoft 2017. XML Data (SQL Server) — Use the BINARY BASE64 Option. <https://www.elastic.co/guide/en/elasticsearch/reference/current/binary.html>. (2017). last checked in July 2017.
- MongoDB 2017. MongoDB Extended JSON. <https://docs.mongodb.com/manual/reference/mongodb-extended-json/>. (2017). last checked in July 2017.
- Thomas Nägele. 2015. *Client-side performance profiling of JavaScript for web applications*. Master’s thesis. Radboud University Nijmegen, the Netherlands.
- Gabriele Paoloni. 2010. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Intel Corporation, Santa Clara, CA.
- Guru Prasad Srinivasa, Rizwana Begum, Scott Haseley, Mark Hempstead, and Geoffrey Challen. 2017. Separated By Birth: Hidden Differences Between Seemingly-Identical Smartphone CPUs. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications (HotMobile '17)*. ACM, New York, NY, USA, 103–108. DOI: <http://dx.doi.org/10.1145/3032970.3032982>
- Yi Tang and Manjia Lin. 2015. EQPO: Obscuring Encrypted Web Traffic with Equal-Sized Pseudo-Objects. In *International Conference on Information Security and Cryptology*. Springer, New York, 227–245.
- Matt Tierney, Ian Spiro, Christoph Bregler, and Lakshminarayanan Subramanian. 2013. Cryptagram: Photo Privacy for Online Social Media. In *Proceedings of the First ACM Conference on Online Social Networks (COSN '13)*. ACM, New York, NY, USA, 75–88. DOI: <http://dx.doi.org/10.1145/2512938.2512939>
- Haichuan Wang, Peng Wu, Ilie Gabriel Tanase, Mauricio J. Serrano, and José E. Moreira. 2014. Simple, Portable and Fast SIMD Intrinsic Programming: Generic SIMD Library. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing (WPMVP '14)*. ACM, New York, NY, USA, 9–16. DOI: <http://dx.doi.org/10.1145/2568058.2568059>