# A Branchless UTF-8 Decoder

October 06, 2017
nullprogram.com/blog/2017/10/06/

This week I took a crack at writing a branchless UTF-8 decoder: a function that decodes a single UTF-8 code point from a byte stream without any `if` statements, loops, short-circuit operators, or other sorts of conditional jumps. You can find the source code here along with a test suite and benchmark:

- https://github.com/skeeto/branchless-utf8

In addition to decoding the next code point, it detects any errors and returns a pointer to the next code point. It's the complete package.

Why branchless? Because high performance CPUs are pipelined. That is, a single instruction is executed over a series of stages, and many instructions are executed in overlapping time intervals, each at a different stage.

The usual analogy is laundry. You can have more than one load of laundry in process at a time because laundry is typically a pipelined process. There's a washing machine stage, dryer stage, and folding stage. One load can be in the washer, a second in the drier, and a third being folded, all at once. This greatly increases throughput because, under ideal circumstances with a full pipeline, an instruction is completed each clock cycle despite any individual instruction taking many clock cycles to complete.

Branches are the enemy of pipelines. The CPU can't begin work on the next instruction if it doesn't know which instruction will be executed next. It must finish computing the branch condition before it can know. To deal with this, pipelined CPUs are also equipped with *branch predictors*. It makes a guess at which branch will be taken and begins executing instructions on that branch. The prediction is initially made using static heuristics, and later those predictions are improved by learning from previous behavior. This even includes predicting the number of iterations of a loop so that the final iteration isn't mispredicted.

A mispredicted branch has two dire consequences. First, all the progress on the incorrect branch will need to be discarded. Second, the pipeline will be flushed, and the CPU will be inefficient until the pipeline fills back up with instructions on the correct branch. With a sufficiently deep pipeline, it can easily be **more efficient to compute and discard an unneeded result than to avoid computing it in the first place**. Eliminating branches means eliminating the hazards of misprediction.

Another hazard for pipelines is *dependencies*. If an instruction depends on the result of a previous instruction, it may have to wait for the previous instruction to make sufficient progress before it can complete one of its stages. This is known as a *pipeline stall*, and it is an important consideration in instruction set architecture (ISA) design.

For example, on the x86-64 architecture, storing a 32-bit result in a 64-bit register will automatically clear the upper 32 bits of that register. Any further use of that destination register cannot depend on prior instructions since all bits have been set. This particular optimization was missed in the design of the i386: Writing a 16-bit result to 32-bit register leaves the upper 16 bits intact, creating false dependencies.

Dependency hazards are mitigated using *out-of-order execution*. Rather than execute two dependent instructions back to back, which would result in a stall, the CPU may instead executing an independent instruction further away in between. A good compiler will also try to spread out dependent instructions in its own instruction scheduling.

The effects of out-of-order execution are typically not visible to a single thread, where everything will appear to have executed in order. However, when multiple processes or threads can access the same memory out-of-order execution can be observed. It's one of the many challenges of writing multi-threaded software.

The focus of my UTF-8 decoder was to be branchless, but there was one interesting dependency hazard that neither GCC nor Clang were able to resolve themselves. More on that later.

**What is UTF-8?**

Without getting into the history of it, you can generally think of UTF-8 as a method for encoding a series of 21-bit integers (*code points*) into a stream of bytes.

- Shorter integers encode to fewer bytes than larger integers. The shortest available encoding must be chosen, meaning there is one canonical encoding for a given sequence of code points.
- Certain code points are off limits: *surrogate halves*. These are code points `U+D800` through `U+DFFF`. Surrogates are used in UTF-16 to represent code points above U+FFFF and serve no purpose in UTF-8. This has interesting consequences for pseudo-Unicode strings, such "wide" strings in the Win32 API, where surrogates may appear unpaired. Such sequences cannot legally be represented in UTF-8.

Keeping in mind these two rules, the entire format is summarized by this table:

```
length byte[0]  byte[1]  byte[2]  byte[3]
1      0xxxxxxx
2      110xxxxx 10xxxxxx
3      1110xxxx 10xxxxxx 10xxxxxx
4      11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

The `x` placeholders are the bits of the encoded code point.

UTF-8 has some really useful properties:

- It's backwards compatible with ASCII, which never used the highest bit.
- Sort order is preserved. Sorting a set of code point sequences has the same result as sorting their UTF-8 encoding.
- No additional zero bytes are introduced. In C we can continue using null terminated `char` buffers, often without even realizing they hold UTF-8 data.
- It's self-synchronizing. A leading byte will never be mistaken for a continuation byte. This allows for byte-wise substring searches, meaning UTF-8 unaware functions like `strstr(3)` continue to work without modification (except for normalization issues). It also allows for unambiguous recovery of a damaged stream.

A straightforward approach to decoding might look something like this:

```
unsigned char *
utf8_simple(unsigned char *s, long *c)
{
    unsigned char *next;
    if (s[0] < 0x80) {
        *c = s[0];
        next = s + 1;
    } else if ((s[0] & 0xe0) == 0xc0) {
        *c = ((long)(s[0] & 0x1f) <<  6) |
             ((long)(s[1] & 0x3f) <<  0);
        next = s + 2;
    } else if ((s[0] & 0xf0) == 0xe0) {
        *c = ((long)(s[0] & 0x0f) << 12) |
             ((long)(s[1] & 0x3f) <<  6) |
             ((long)(s[2] & 0x3f) <<  0);
        next = s + 3;
    } else if ((s[0] & 0xf8) == 0xf0 && (s[0] <= 0xf4)) {
        *c = ((long)(s[0] & 0x07) << 18) |
             ((long)(s[1] & 0x3f) << 12) |
             ((long)(s[2] & 0x3f) <<  6) |
             ((long)(s[3] & 0x3f) <<  0);
        next = s + 4;
    } else {
        *c = -1; // invalid
        next = s + 1; // skip this byte
    }
    if (*c >= 0xd800 && *c <= 0xdfff)
        *c = -1; // surrogate half
    return next;
}
```

It branches off on the highest bits of the leading byte, extracts all of those `x` bits from each byte, concatenates those bits, checks if it's a surrogate half, and

returns a pointer to the next character. (This implementation does *not* check that the highest two bits of each continuation byte are correct.)

The CPU must correctly predict the length of the code point or else it will suffer a hazard. An incorrect guess will stall the pipeline and slow down decoding.

In real world text this is probably not a serious issue. For the English language, the encoded length is nearly always a single byte. However, even for non-English languages, text is usually accompanied by markup from the ASCII range of characters, and, overall, the encoded lengths will still have consistency. As I said, the CPU predicts branches based on the program's previous behavior, so this means it will temporarily learn some of the statistical properties of the language being actively decoded. Pretty cool, eh?

Eliminating branches from the decoder side-steps any issues with mispredicting encoded lengths. Only errors in the stream will cause stalls. Since that's probably the unusual case, the branch predictor will be very successful by continually predicting success. That's one optimistic CPU.

**The branchless decoder**

Here's the interface to my branchless decoder:

```
void *utf8_decode(void *buf, uint32_t *c, int *e);
```

I chose `void *` for the buffer so that it doesn't care what type was actually chosen to represent the buffer. It could be a `uint8_t`, `char`, `unsigned char`, etc. Doesn't matter. The encoder accesses it only as bytes.

On the other hand, with this interface you're forced to use `uint32_t` to represent code points. You could always change the function to suit your own needs, though.

Errors are returned in `e`. It's zero for success and non-zero when an error was detected, without any particular meaning for different values. Error conditions are mixed into this integer, so a zero simply means the absence of error.

This is where you could accuse me of "cheating" a little bit. The caller probably wants to check for errors, and so *they* will have to branch on `e`. It seems I've just smuggled the branches outside of the decoder.

However, as I pointed out, unless you're expecting lots of errors, the real cost is branching on encoded lengths. Furthermore, the caller could instead accumulate the errors: count them, or make the error "sticky" by ORing all `e` values together. Neither of these require a branch. The caller could decode a huge stream and only check for errors at the very end. The only branch would be the main loop ("are we done yet?"), which is trivial to predict with high accuracy.

The first thing the function does is extract the encoded length of the next code point:

```
static const char lengths[] = {
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 3, 3, 4, 0
};

unsigned char *s = buf;
int len = lengths[s[0] >> 3];
```

Looking back to the UTF-8 table above, only the highest 5 bits determine the length. That's 32 possible values. The zeros are for invalid prefixes. This will later cause a bit to be set in `e`.

With the length in hand, it can compute the position of the next code point in the buffer.

```
unsigned char *next = s + len + !len;
```

Originally this expression was the return value, computed at the very end of the function. However, after inspecting the compiler's assembly output, I decided to move it up, and the result was a solid performance boost. That's because it spreads out dependent instructions. With the address of the next code point known so early, the instructions that decode the next code point can get started early.

The reason for the `!len` is so that the pointer is advanced one byte even in the face of an error (length of zero). Adding that `!len` is actually somewhat costly, though I couldn't figure out why.

```
static const int shiftc[] = {0, 18, 12, 6, 0};

*c  = (uint32_t)(s[0] & masks[len]) << 18;
*c |= (uint32_t)(s[1] & 0x3f) << 12;
*c |= (uint32_t)(s[2] & 0x3f) <<  6;
*c |= (uint32_t)(s[3] & 0x3f) <<  0;
*c >>= shiftc[len];
```

This reads four bytes regardless of the actual length. Avoiding doing something is branching, so this can't be helped. The unneeded bits are shifted out based on the length. That's all it takes to decode UTF-8 without branching.

One important consequence of always reading four bytes is that **the caller *must* zero-pad the buffer to at least four bytes**. In practice, this means padding the entire buffer with three bytes in case the last character is a single byte.

The padding must be zero in order to detect errors. Otherwise the padding might look like legal continuation bytes.

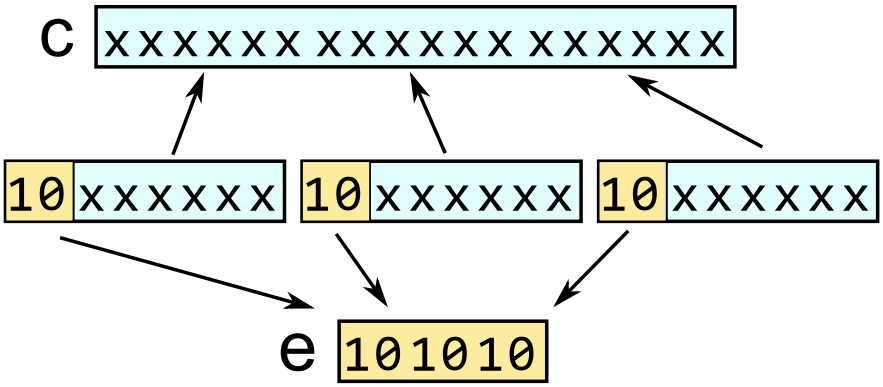
```
static const uint32_t mins[] = {4194304, 0, 128, 2048, 65536};
static const int shifte[] = {0, 6, 4, 2, 0};

*e  = (*c < mins[len]) << 6;
*e |= ((*c >> 11) == 0x1b) << 7;  // surrogate half?
*e |= (s[1] & 0xc0) >> 2;
*e |= (s[2] & 0xc0) >> 4;
*e |= (s[3]       ) >> 6;
*e ^= 0x2a;
*e >>= shifte[len];
```

The first line checks if the shortest encoding was used, setting a bit in `e` if it wasn't. For a length of 0, this always fails.

The second line checks for a surrogate half by checking for a certain prefix.

The next three lines accumulate the highest two bits of each continuation byte into `e`. Each should be the bits `10`. These bits are "compared" to `101010` (`0x2a`) using XOR. The XOR clears these bits as long as they exactly match.




Finally the continuation prefix bits that don't matter are shifted out.

**The goal**

My primary — and totally arbitrary — goal was to beat the performance of Björn Höhrmann's DFA-based decoder. Under favorable (and artificial) benchmark conditions I had moderate success. You can try it out on your own system by cloning the repository and running `make bench`.

With GCC 6.3.0 on an i7-6700, my decoder is about 20% faster than the DFA decoder in the benchmark. With Clang 3.8.1 it's just 1% faster.

*Update*: Björn pointed out that his site includes a faster variant of his DFA decoder. It is only 10% slower than the branchless decoder with GCC, and it's 20% faster than the branchless decoder with Clang. So, in a sense, it's still faster on average, even on a benchmark that favors a branchless decoder.

The benchmark operates very similarly to my PRNG shootout (e.g. `alarm(2)`). First a buffer is filled with random UTF-8 data, then the decoder decodes it

again and again until the alarm fires. The measurement is the number of bytes decoded.

The number of errors is printed at the end (always 0) in order to force errors to actually get checked for each code point. Otherwise the sneaky compiler omits the error checking from the branchless decoder, making it appear much faster than it really is — a serious letdown once I noticed my error. Since the other decoder is a DFA and error checking is built into its graph, the compiler can't really omit its error checking.

I called this "favorable" because the buffer being decoded isn't anything natural. Each time a code point is generated, first a length is chosen uniformly: 1, 2, 3, or 4. Then a code point that encodes to that length is generated. The **even distribution of lengths greatly favors a branchless decoder**. The random distribution inhibits branch prediction. Real text has a far more favorable distribution.

```
uint32_t
randchar(uint64_t *s)
{
    uint32_t r = rand32(s);
    int len = 1 + (r & 0x3);
    r >>= 2;
    switch (len) {
        case 1:
            return r % 128;
        case 2:
            return 128 + r % (2048 - 128);
        case 3:
            return 2048 + r % (65536 - 2048);
        case 4:
            return 65536 + r % (131072 - 65536);
    }
    abort();
}
```

Given the odd input zero-padding requirement and the artificial parameters of the benchmark, despite the supposed 20% speed boost under GCC, my branchless decoder is not really any better than the DFA decoder in practice. It's just a different approach. In practice I'd prefer Björn's DFA decoder.

*Update*: Bryan Donlan has followed up with a SIMD UTF-8 decoder.

*Update 2024*: NRK has followed up with parallel extract decoder.

*Update 2025*: Charles Eckman followed up sharing a branchless encoder, which inspired me to give it a shot.