SIMD-enhanced libc string functions

how it's done

Robert Clausecker <fuz@FreeBSD.org>
Getz Mikalsen <getz@FreeBSD.org>

Common Tasks

- copying strings (*strcpy*, *memcpy*, ...)
- finding string length (*strlen*, *strnlen*, ...)
- finding characters (*strchr*, *memchr*, ...)
- comparing strings (*strcmp*, *memcmp*, ...)
- finding substrings (*strstr*, *memmem*, ...)
- splitting at delimiters (*strspn*, *strcspn*, ...)

Common Tasks

- copying strings (*read* then *write*)
- finding string length (*read* then *compare*)
- finding characters (*read* then *compare*)
- comparing strings (*read* then *compare*)
- ~~finding substrings~~ (complicated)
- splitting at delimiters (*read* then *set match*)

What does that mean?

read

- char by char until end of string
- one load/compare/conditional branch per character

write

- char by char until end of string
- one write per character

compare

- char by char until match or end of string
- one compare/conditional branch per character

What does that mean?

read

- char by char until end of string
- one load/compare/conditional branch per character (**slow**)

write

- char by char until end of string
- one write per character (**slow**)

compare

- char by char until match or end of string
- one compare/conditional branch per character (**slow**)

Conclusion

Conclusion

Strings suck

What can we do about that?

- Get rid of strings (oof...)
- special-purpose instructions (arch dependent)
  - speed varies depending on CPU model
  - often only *memcpy*(), *memset*() supported
- strange hacks (hmm...)

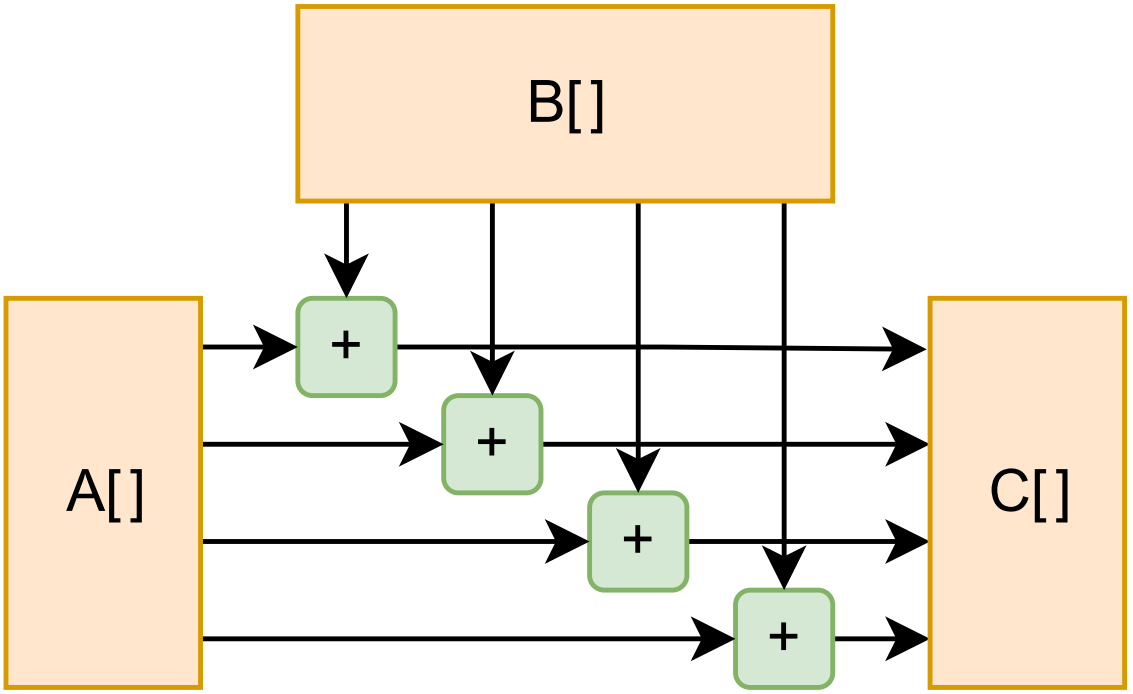
SIMD

Your new best friend

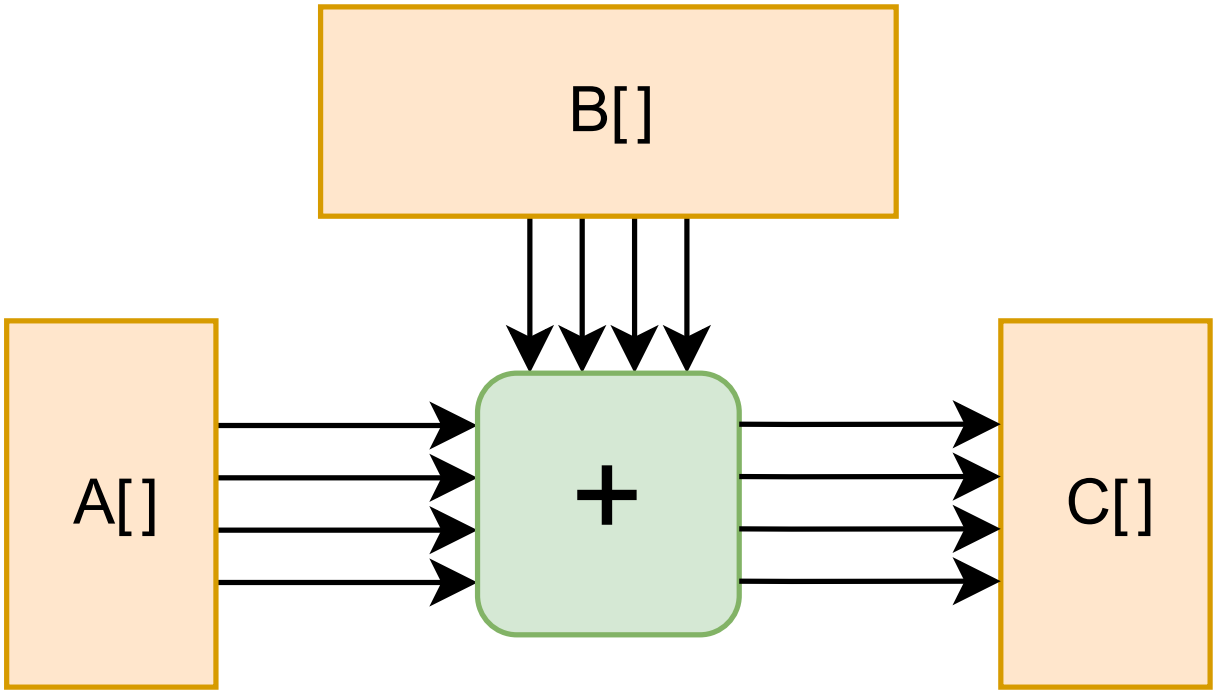
SIMD

- **S**ingle **I**nstruction **M**ultiple **D**ata
- *SIMD register*: short arrays of numbers
- common lengths: 16, 32, 64 bytes
- *same operation* on all elements
- but as fast as scalar operations
- SIMD with 16 bytes: 16x scalar performance

Scalar vs. SIMD



Scalar vs. SIMD



typical SIMD operations

Arithmetic (integer/FP)

- addition, subtraction, multiplication, ...

Logic

- element-wise comparison, and, or, xor, ...

Data transfer

- read, write, extract masks, ...

... many more

Strings and SIMD

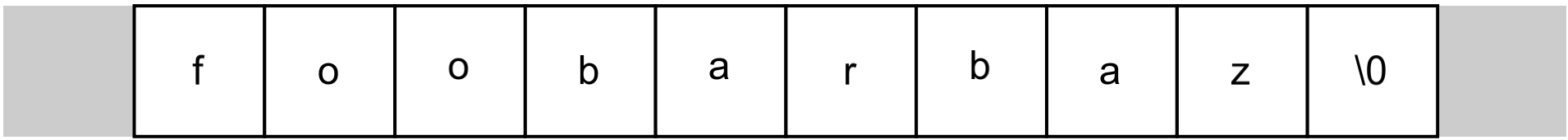
How can this help us with string processing?

Strings and SIMD

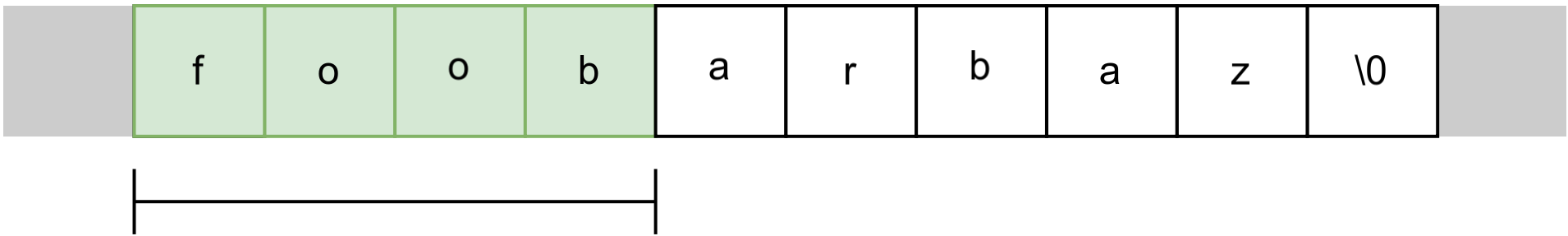
How can this help us with string processing?

- load multiple characters at once
- process them simultaneously
- ...
- profit?

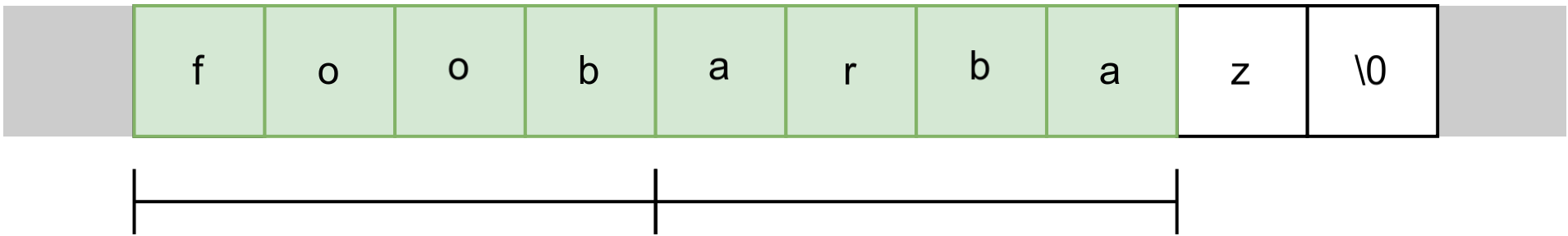
Difficulties



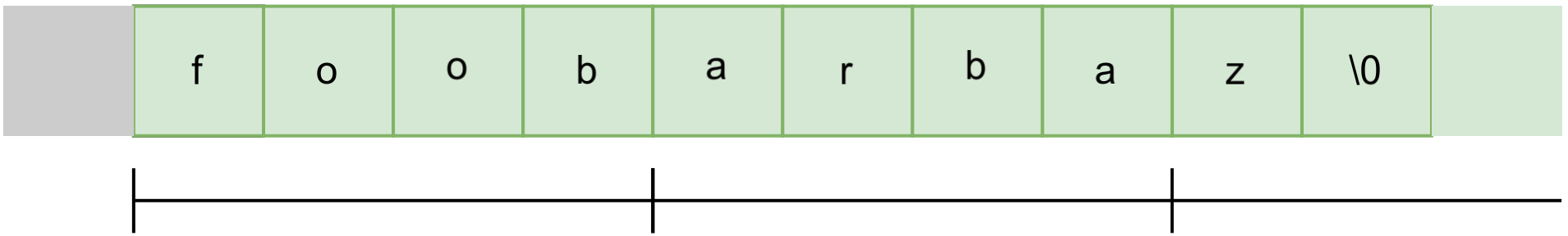
Difficulties



Difficulties



Difficulties



Difficulties

- We can easily overshoot the string's end
- For nul-terminated strings, we won't know where that is until we see the nul byte
- Do we have to iterate char-by-char after all?

What can we do?

What can we do?

String bounds are fictious

What can we do?

String bounds are fictious
Let's overcome them!

Overcoming array bounds

- the computer does not know what an array is
- it only knows that there's memory at some addresses but not at others

Overcoming array bounds

- the computer does not know what an array is
- it only knows that there's memory at some addresses but not at others

thus:

- if we don't go too far out of bounds, it'll be fine!
- C doesn't let us, so let's use assembly

How far is too far?

- Memory is organised in *pages*
- size: arch dependent, usually 4096 bytes
- pages are either accessible entirely or not at all
- there is no more fine-grained memory protection
  - (check out CHERI, it's cool)

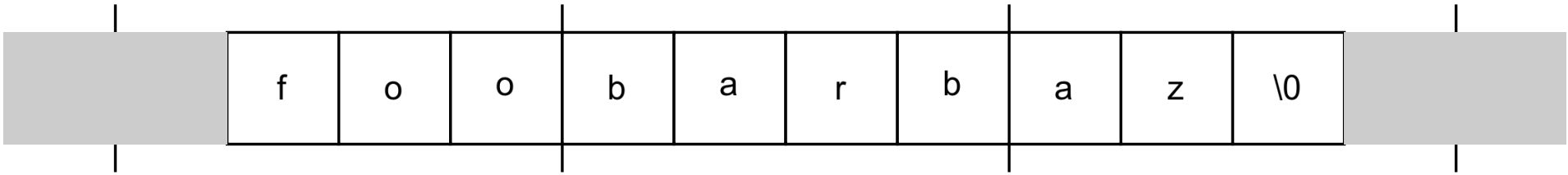
How far is too far?

- if at least one byte of a string is on a page, the whole page is accessible
- aligned accesses never cross page boundaries

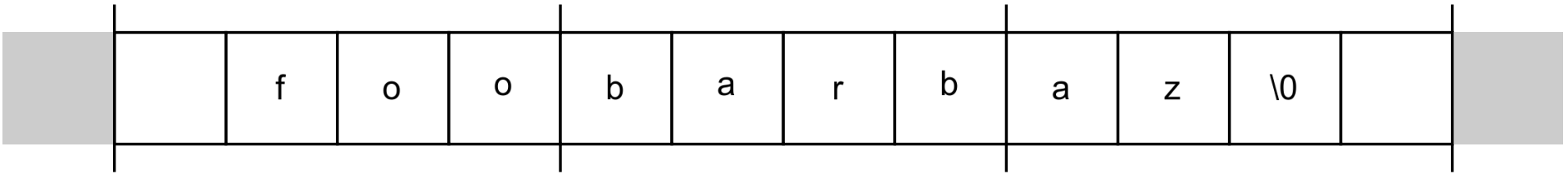
thus:

- if we're careful, it might just work!

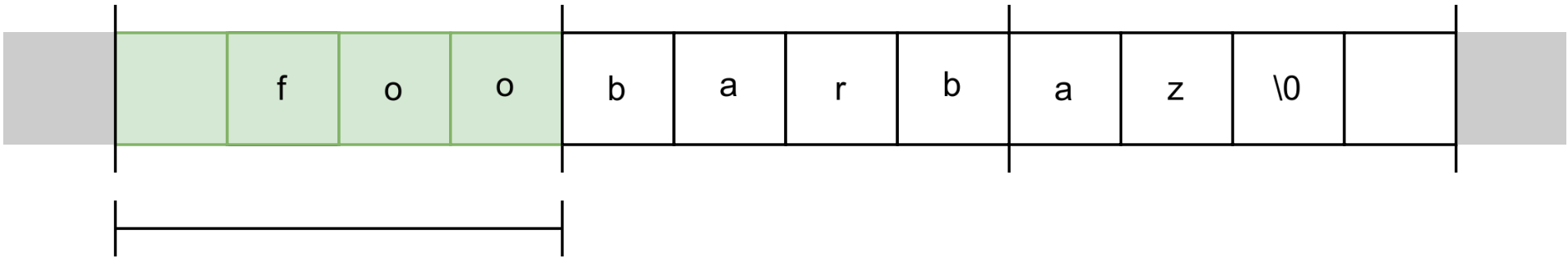
What does that look like?



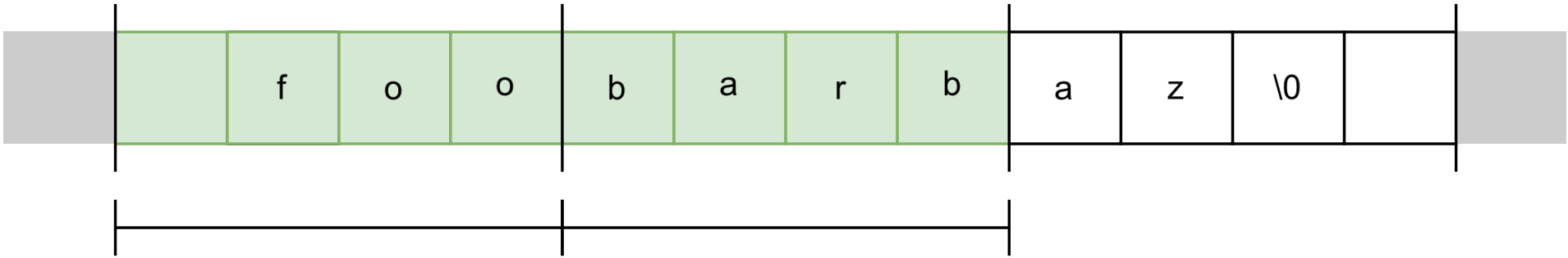
What does that look like?



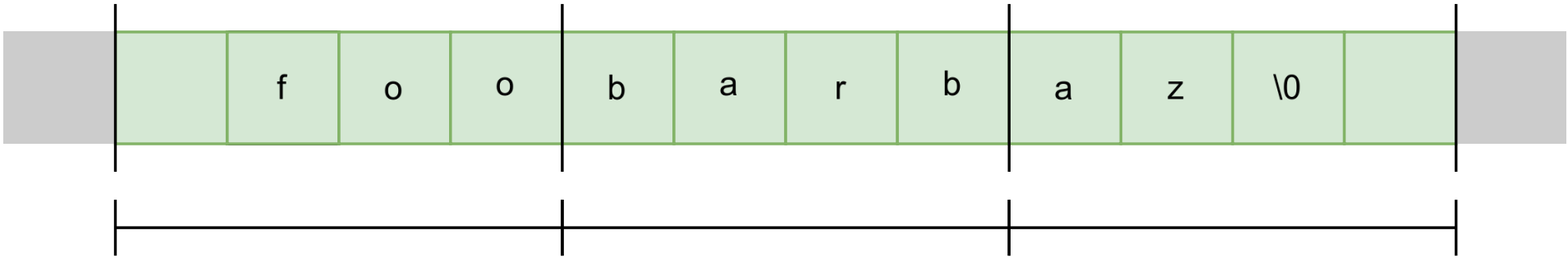
What does that look like?



What does that look like?



What does that look like?



Writing Strings

Can't use the same approach:

- overreads are fine, overwrites are no good

Writing Strings

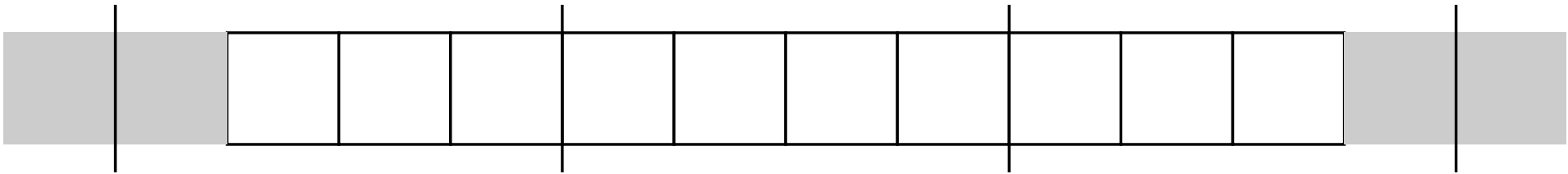
Can't use the same approach:

- overreads are fine, overwrites are no good

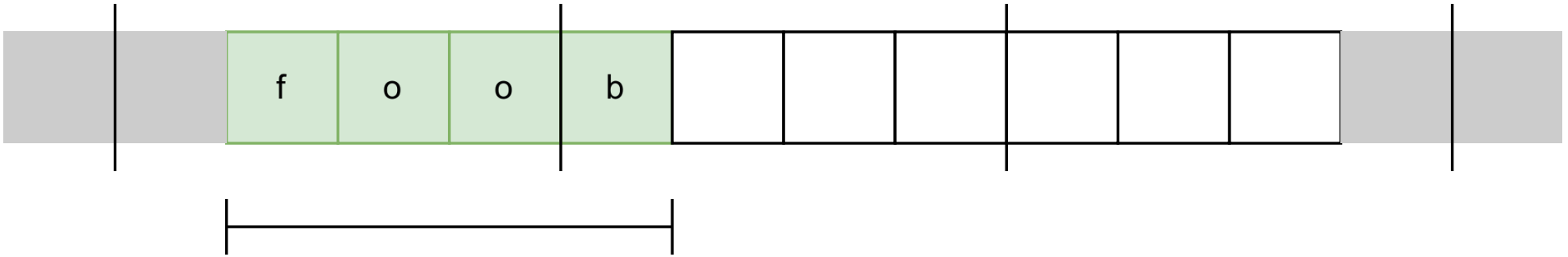
Instead

- write (possibly unaligned) chunks
- last write may overlap previous writes

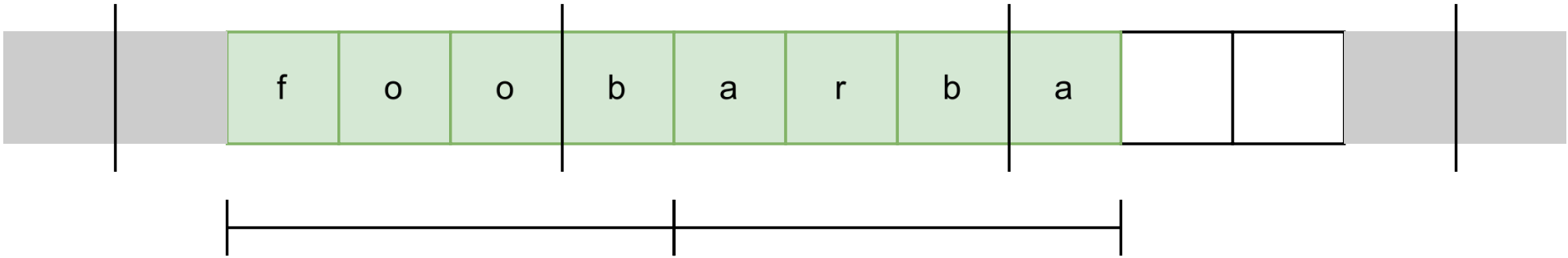
Writing Strings



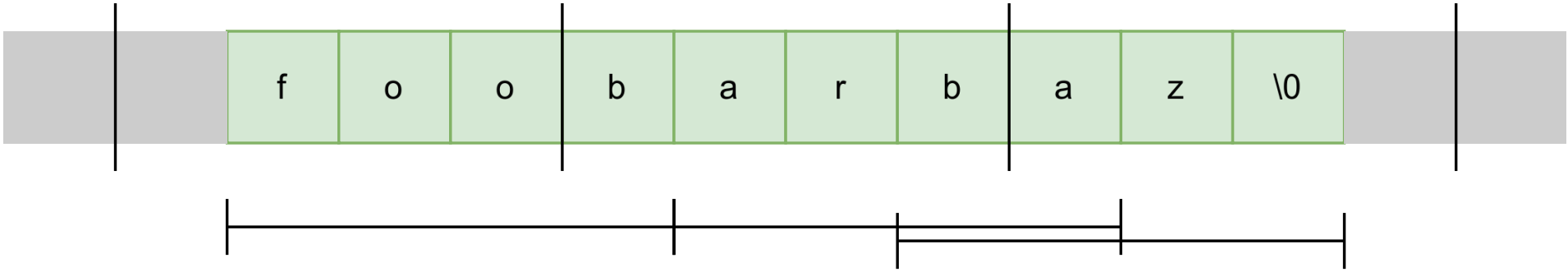
Writing Strings



Writing Strings



Writing Strings



Comparing Strings

Can't use the same approach

- strings may have different misalignment
- can't fix this after loading with SSE2

Comparing Strings

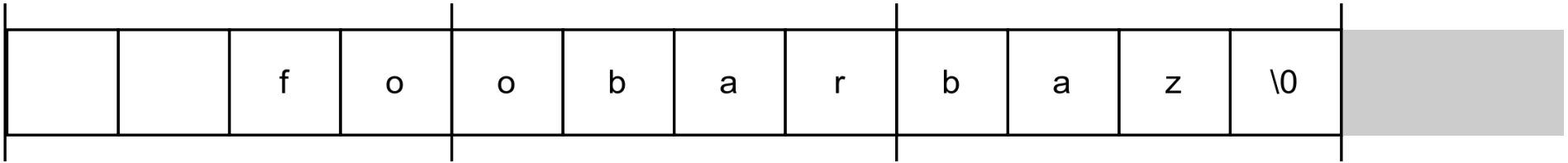
Can't use the same approach

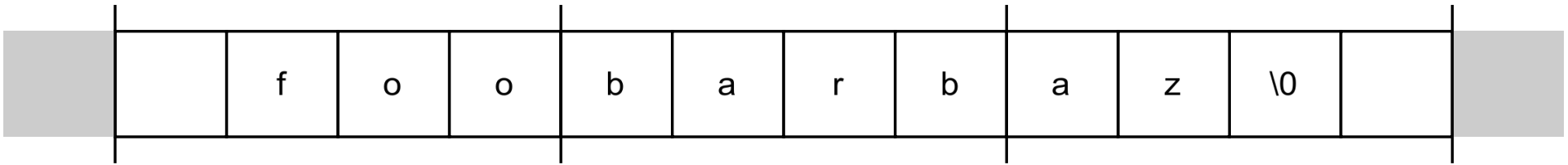
- strings may have different misalignment
- can't fix this after loading with SSE2

Instead

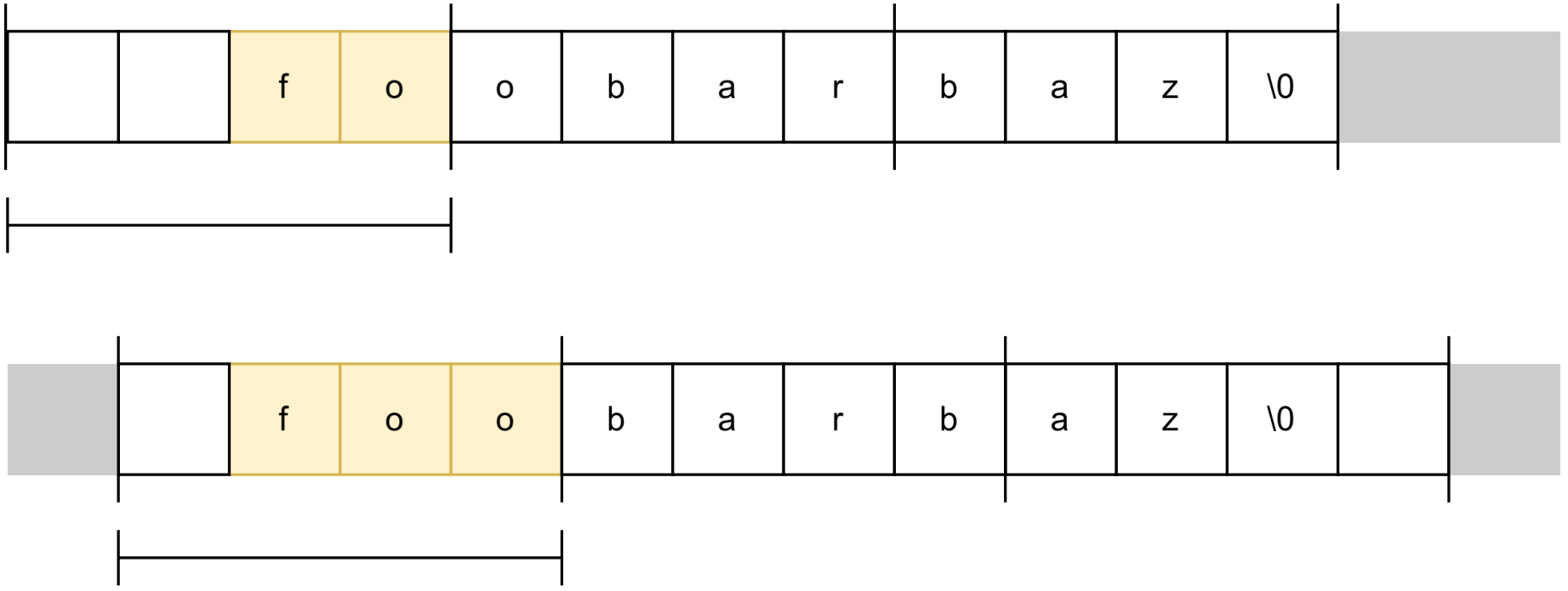
- do aligned reads to check for nul bytes
- then unaligned reads to compare characters

Comparing Strings

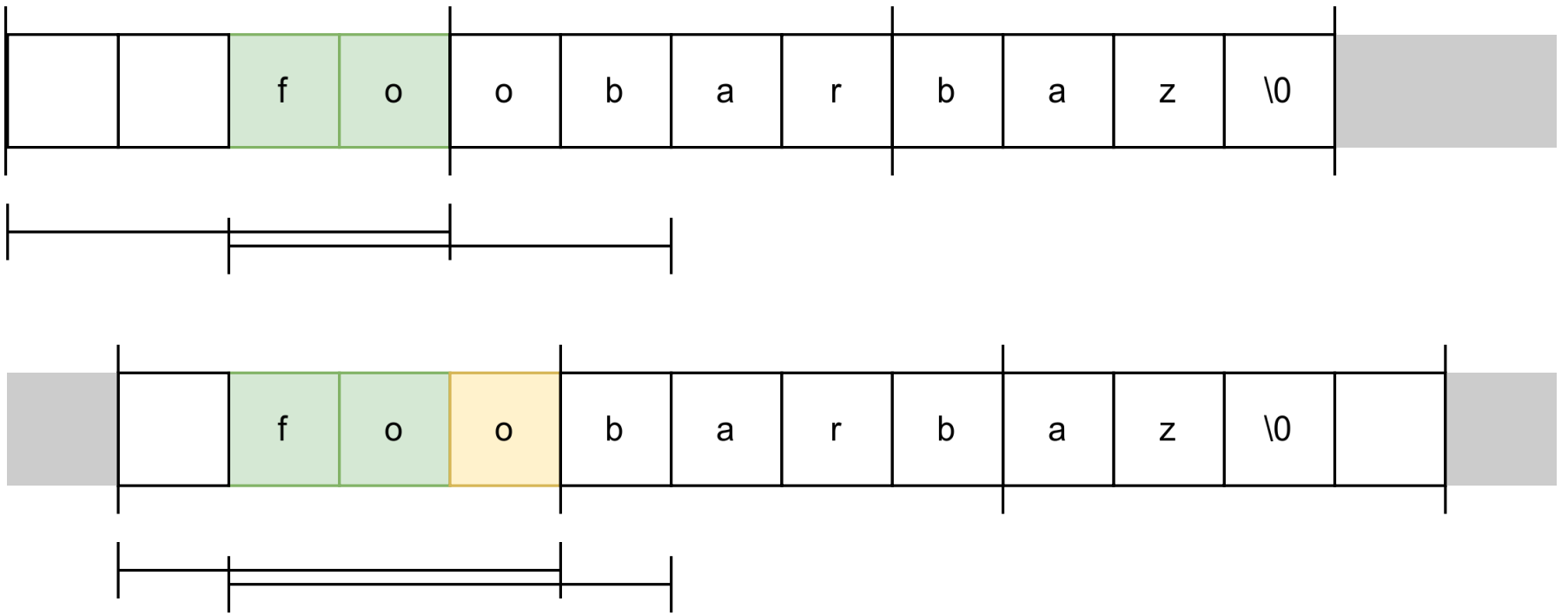




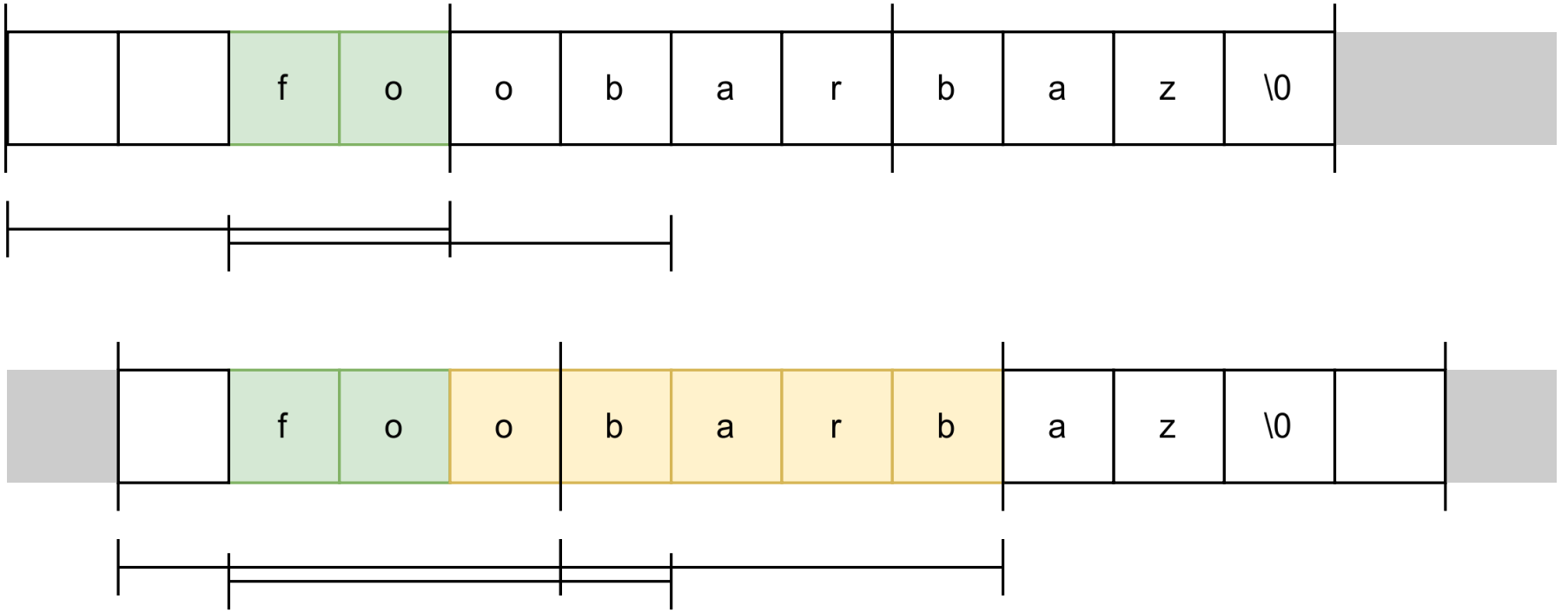
Comparing Strings



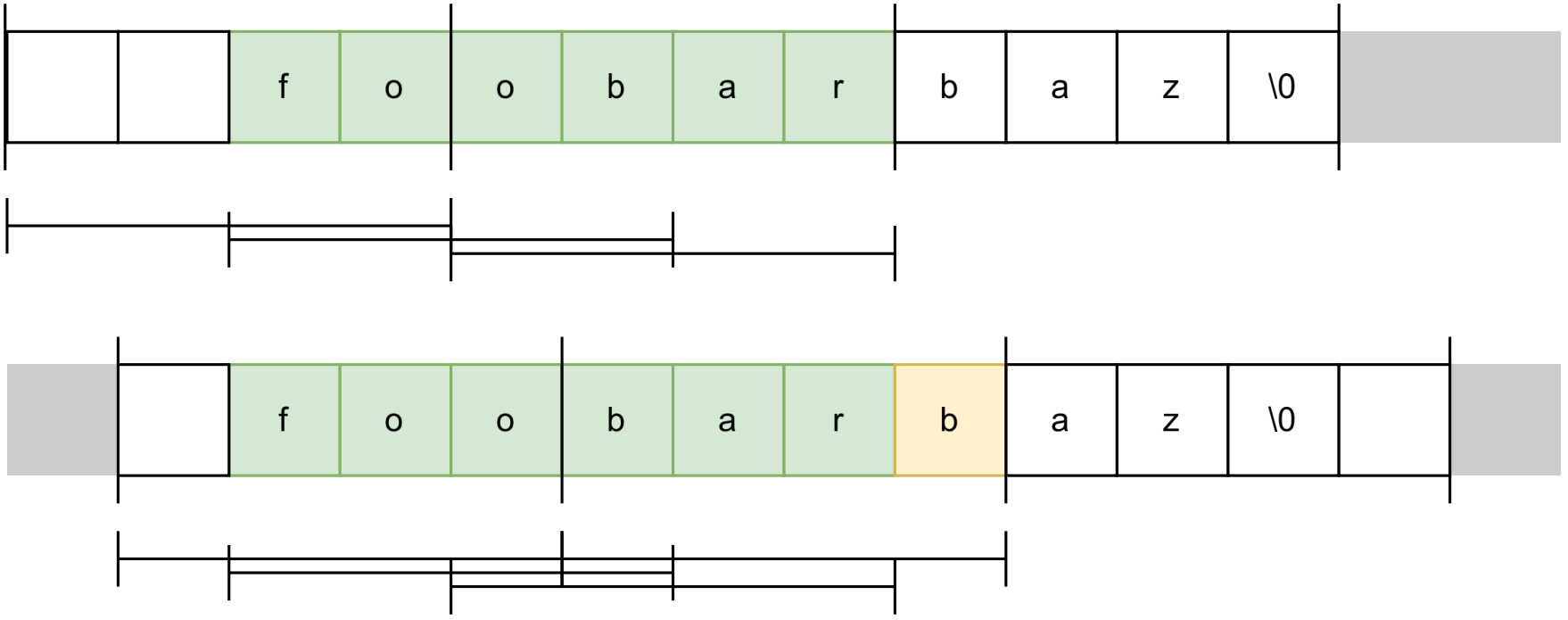
Comparing Strings



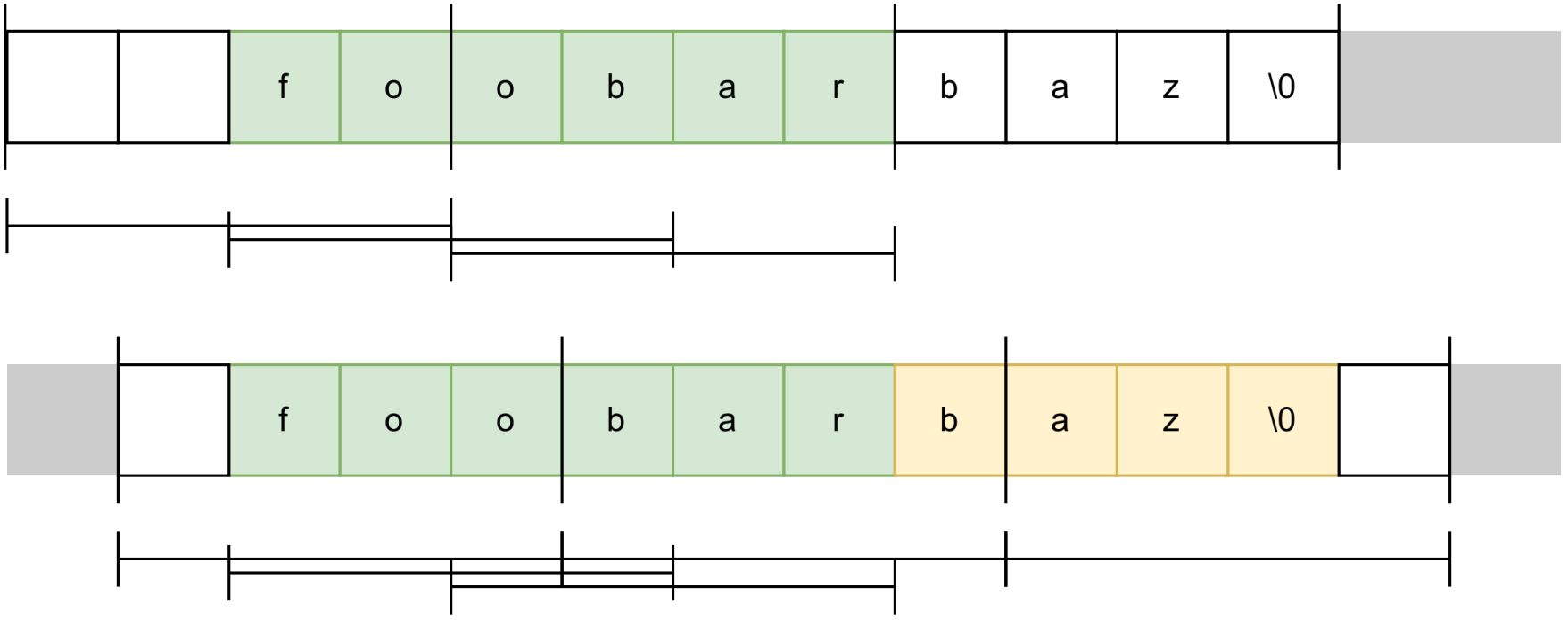
Comparing Strings



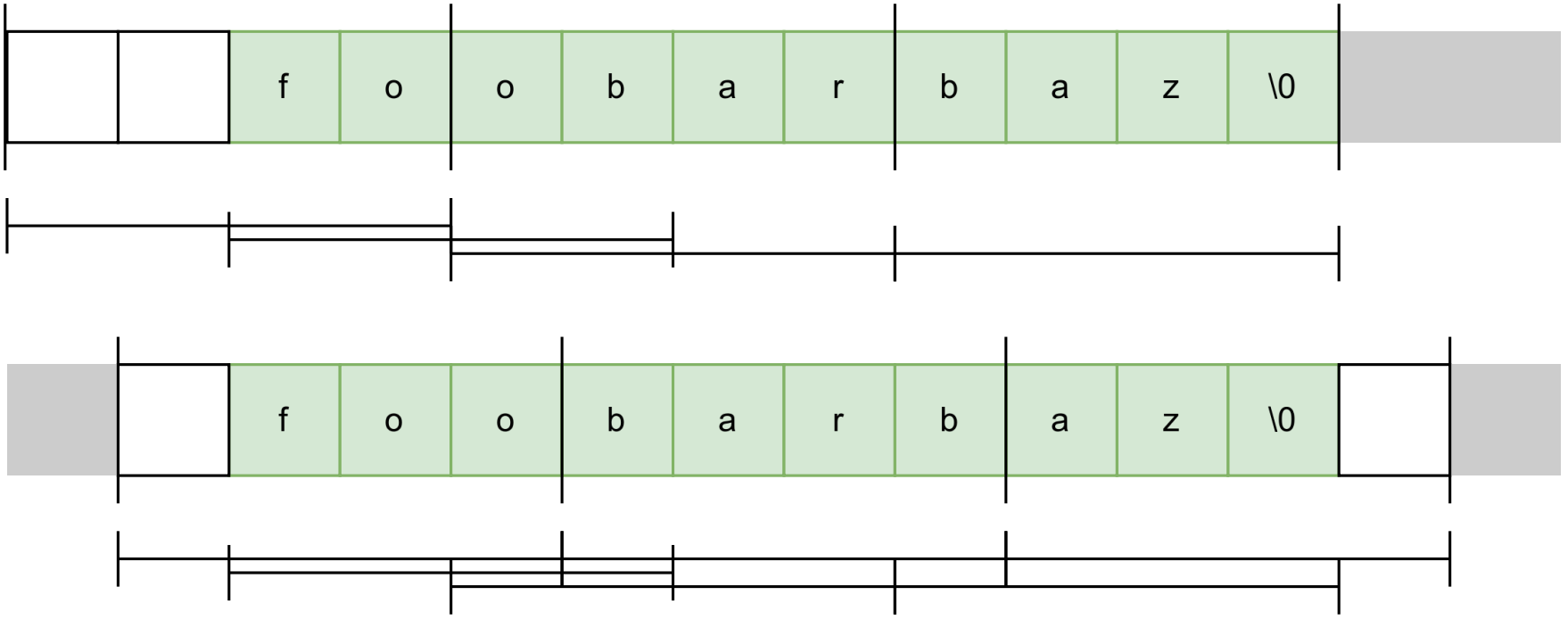
Comparing Strings



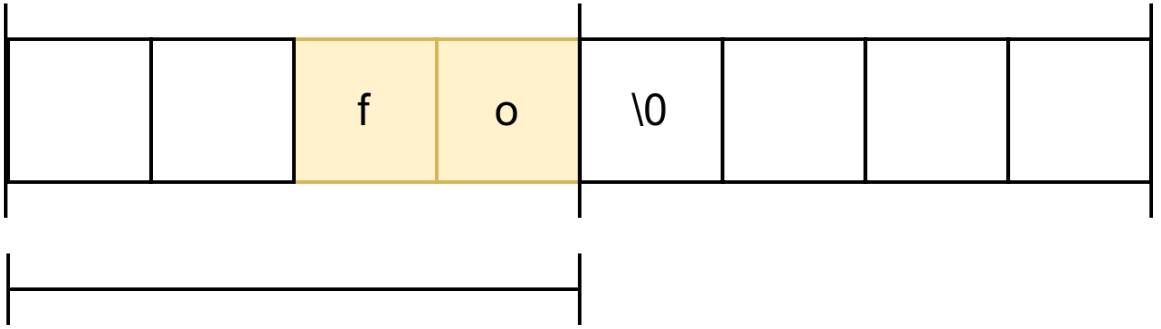
Comparing Strings

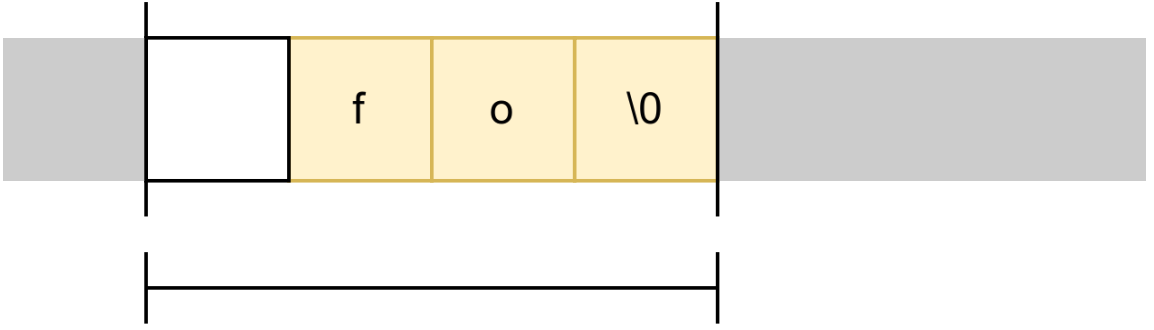


Comparing Strings

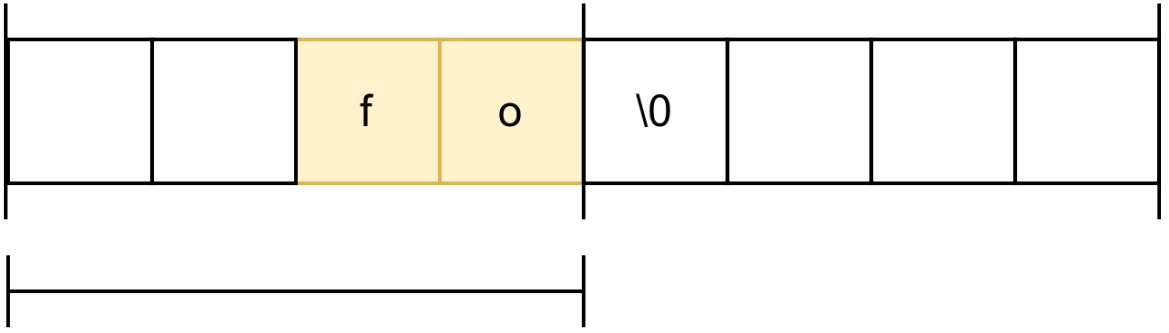


Comparing Strings

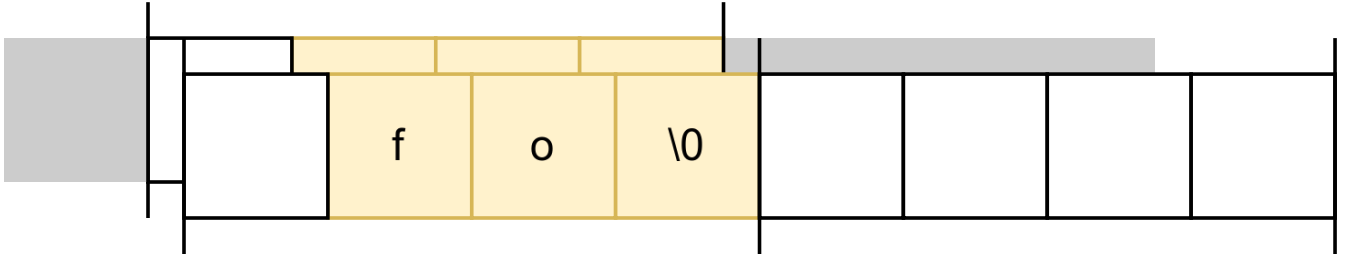




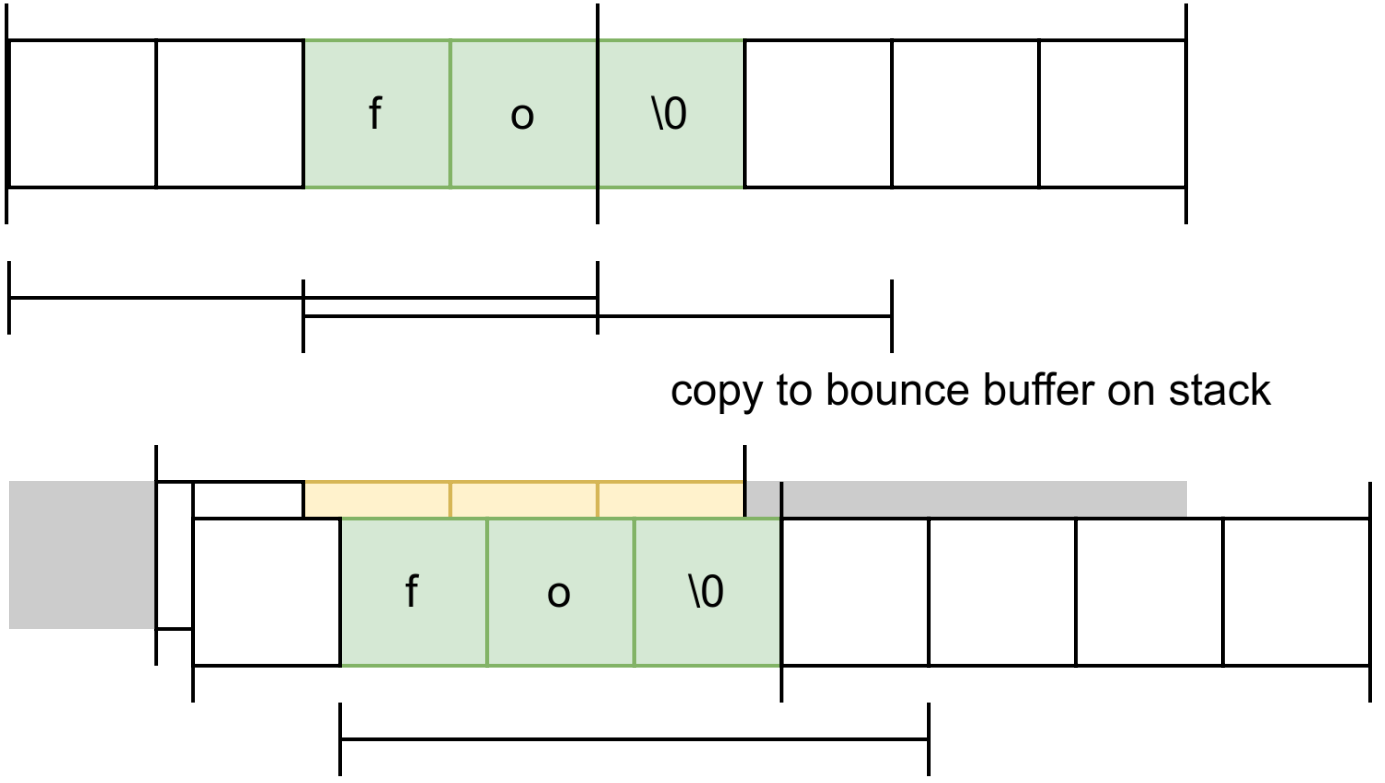
Comparing Strings



copy to bounce buffer on stack



Comparing Strings



Set Matching

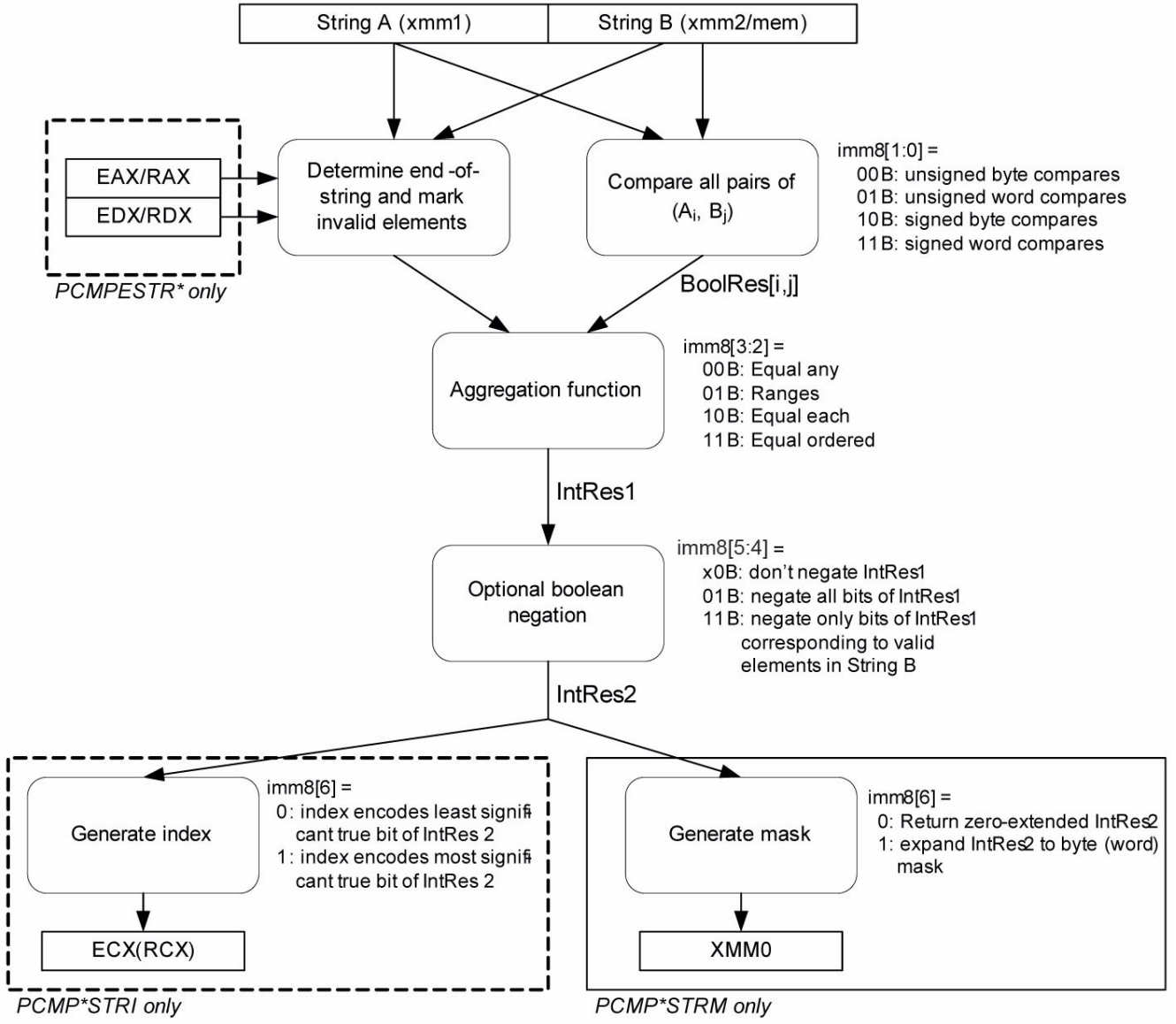
strcspn("foo bar", " \t\n");

- matches each char in string against set
- portable approach: Muła / Langdale algorithm
  http://0x80.pl/articles/simd-byte-lookup.html
- can we do better?

Set Matching

The Intel way: **pcmpistrm**

- **p**acked **c**o**mp**are **i**mplicity-terminated **str**ing, return **m**ask
- set matching and lots of other features
- conveniently also checks for nul terminators
- probably also brews coffee if you ask nicely


String A (xmm1)
String B (xmm2/mem)
EAX/RAX
EDX/RDX
PCMPESTR* only
Determine end -of-string and mark invalid elements
Compare all pairs of (A_i, B_j)
imm8[1:0] =
00B: unsigned byte compares
01B: unsigned word compares
10B: signed byte compares
11B: signed word compares
BoolRes[i,j]
Aggregation function
imm8[3:2] =
00B: Equal any
01B: Ranges
10B: Equal each
11B: Equal ordered
IntRes1
Optional boolean negation
imm8[5:4] =
x0B: don't negate IntRes1
01B: negate all bits of IntRes1
11B: negate only bits of IntRes1 corresponding to valid elements in String B
IntRes2
Generate index
imm8[6] =
0: index encodes least significant true bit of IntRes 2
1: index encodes most significant true bit of IntRes 2
ECX(RCX)
PCMP*STRI only
Generate mask
imm8[6] =
0: Return zero-extended IntRes2
1: expand IntRes2 to byte (word) mask
XMM0
PCMP*STRM only

Substring Matching

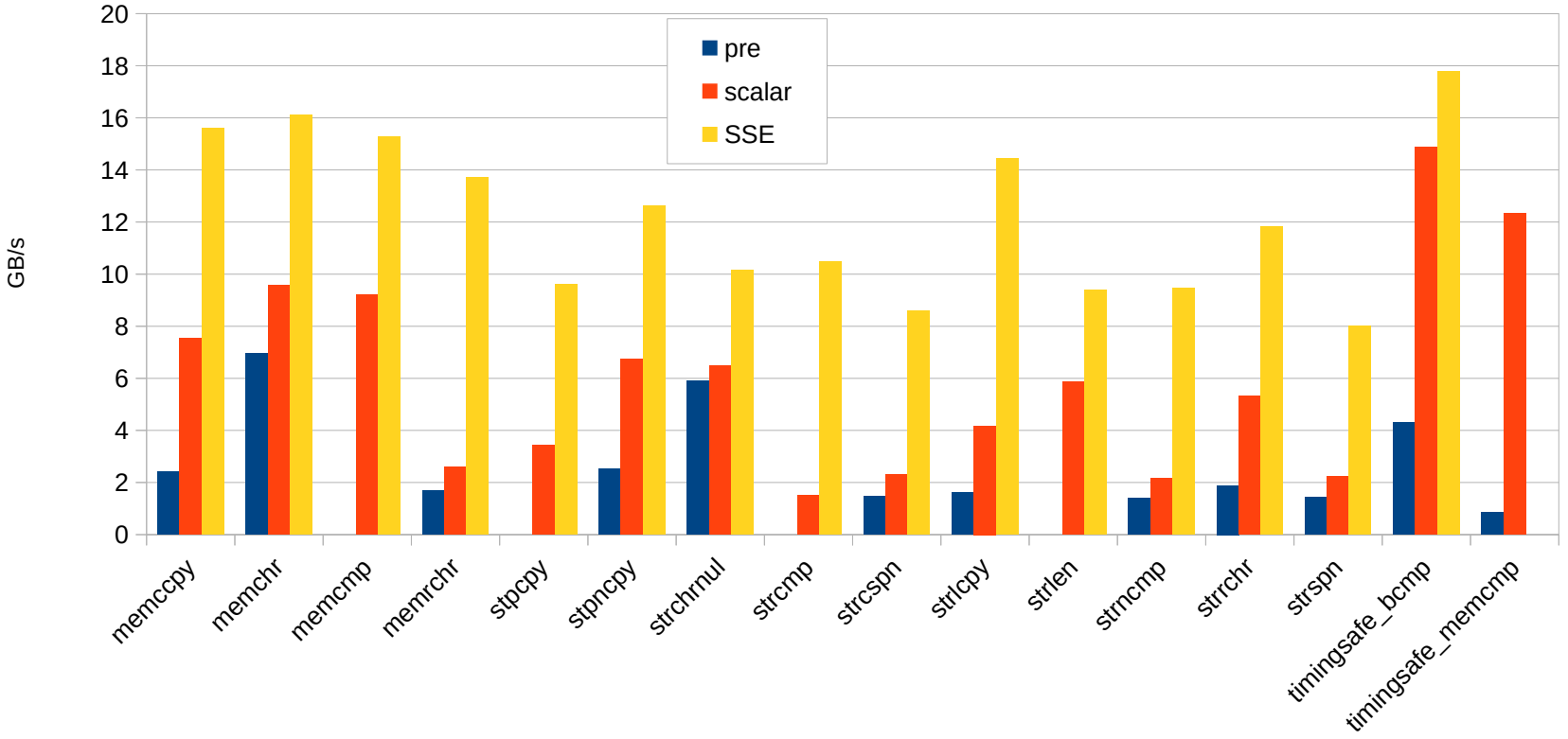
That means *strstr()*, *memmem()*

- really tricky
- most fancy algorithms are optimised for long strings, but our strings are usually short
- wip

Current Progress

- 2023 rework of the libc string functions for amd64
  - paid by The FreeBSD Foundation
  - almost all of <string.h>
  - for amd64 baseline (SSE2), some for x86-64-v2
  - landed for 14.1-RELEASE
- later ports as part of GSoC 2024
  - AArch64 by getz@ (acceptance testing in progress)
  - riscv64 by strajabot@ (work in progress)

Results (amd64)



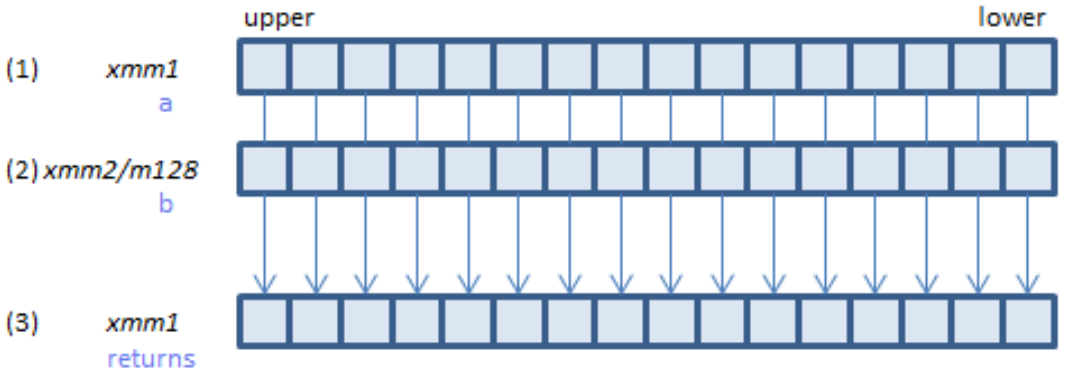
AMD64 <-> Aarch64

Background

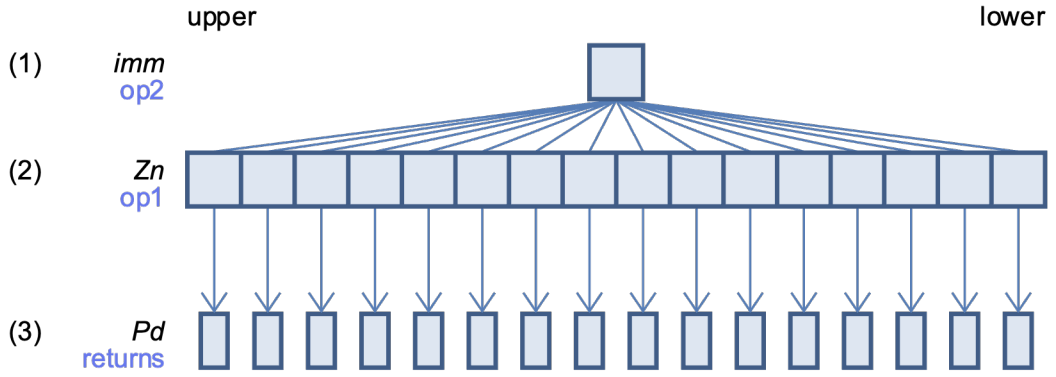
- Project as part of Google Summer of Code 2024
- Port amd64 SIMD libc optimizations to Aarch64
- Another contributor ported to RISC-V
- Several functions already had efficient implementations as part of the Arm Optimized Routines repository in src/contrib
- Several functions had less efficient implementations.
- Some functions missing
- Write all the string functions!

Most common instructions are available

- Bit scanning instructions (minor variations) Performed in a GPR after a match is found.
- Bytewise comparisons



PCMPEQB (**p**acked **com**pare for **eq**uality **b**ytes)



CMEQ (**com**pare bitwise **eq**ual)

Some require extra fiddling

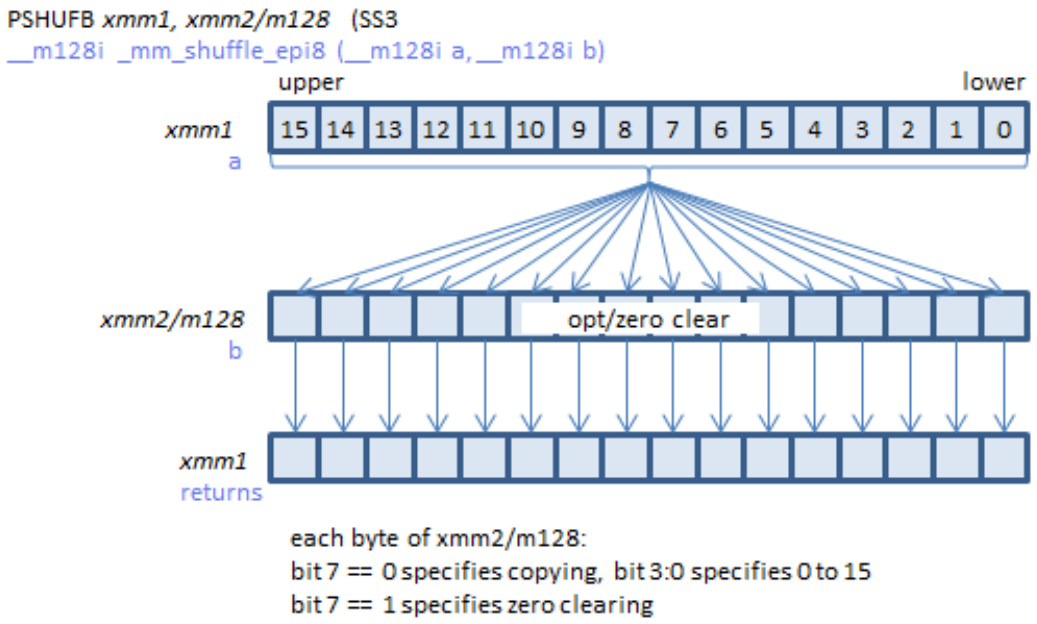
- For counted string functions we avoid branches by inducing a “fake” match in the match mask where the buffer ends.

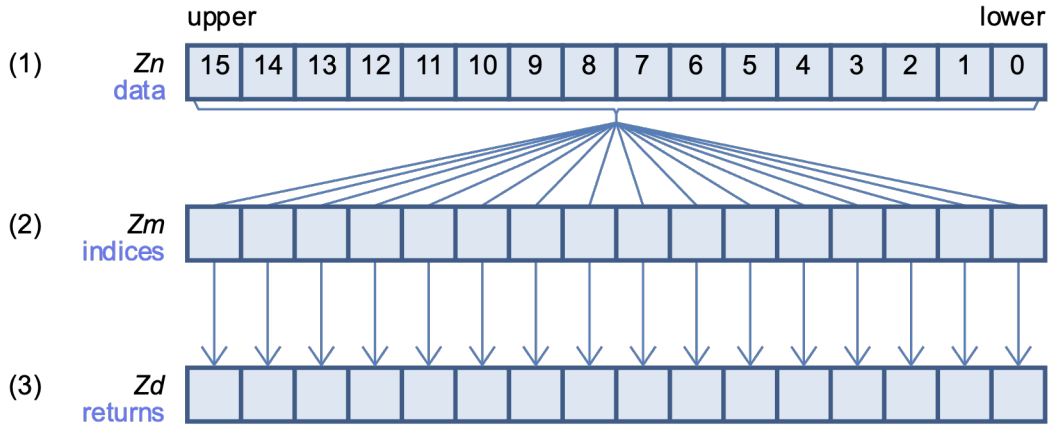
```
/* end of buffer will occur in next 32 bytes */
.Ltail:
          movdqu    (%rdi, %rbx, 1), %xmm0
          pxor      %xmm1, %xmm1
          pcmpeqb   (%rdi, %rsi, 1), %xmm1
          pcmpeqb   (%rdi), %xmm0
          pmovmskb  %xmm1, %r8d
          pmovmskb  %xmm0, %r9d
          bts       %edx, %r8d
          test      %r8w, %r8w
          jnz       .Lnul_found
          xor       $0xffff, %r9d
          jnz       .Lmismatch
```

```
.Ltail:
          ldr       q0, [x8, x11]
          ldr       q1, [x8, x10]
          ldr       q2, [x8]

          cmeq      v1.16b, v1.16b, #0
          cmeq      v0.16b, v0.16b, v2.16b

          shrn      v1.8b, v1.8h, #4
          shrn      v0.8b, v0.8h, #4
          fmov      x6, d1
          fmov      x5, d0

          mov       x13, #0xf
          lsl       x4, x2, #2
          lsl       x4, x13, x4
          orr       x3, x6, x4
          cmp       x2, #16
          csel      x6, x3, x6, lo

          cbnz      x6, .Lnulfound
          cbz       x5, .Lmismatch
```

How to be careful not to step into the void

- When buffer located at end of a page
- No variable shift for SIMD registers





```
movdqa          (%rdi), %xmm0
movdqa          (%rsi), %xmm2
mov             $-1, %r8d
mov             $-1, %r9d
mov             %eax, %ecx
shl             %cl, %r8d
mov             %edx, %ecx
shl             %cl, %r9d
movdqa          %xmm0, -40(%rsp)
movdqa          %xmm2, -24(%rsp)
pcmpeqb         %xmm1, %xmm0
pcmpeqb         %xmm1, %xmm2
pmovmskb        %xmm0, %r10d
pmovmskb        %xmm2, %r11d
test            %r8d, %r10d
lea             -40(%rsp), %r8
cmovz           %rdi, %r8
test            %r9d, %r11d
lea             -24(%rsp), %r9
cmovz           %rsi, %r9
movdqu          (%r8, %rax, 1), %xmm0
movdqu          (%r9, %rdx, 1), %xmm4
```

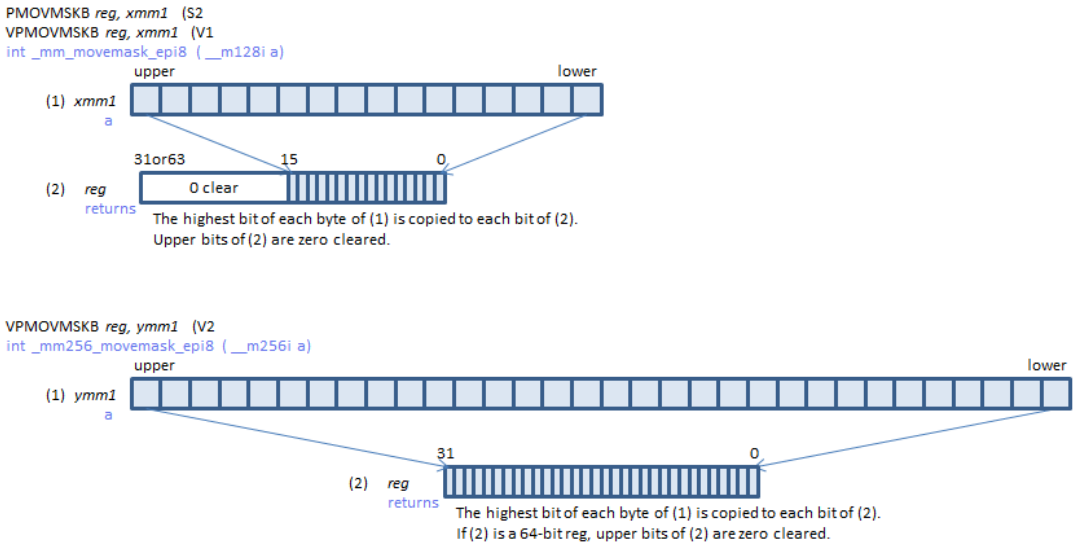
amd64

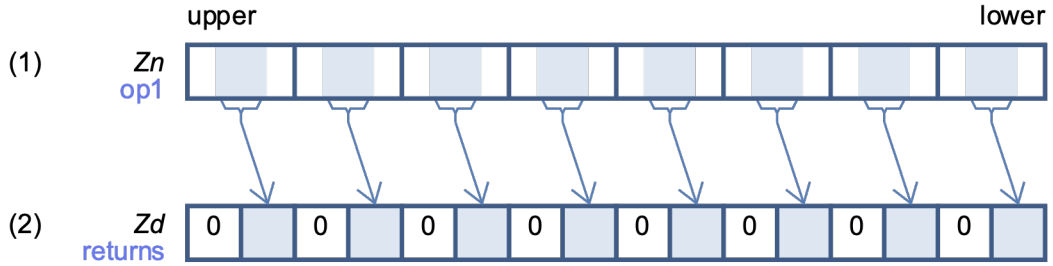
```
        bic     x8, x0, #0xf
        and     x9, x0, #0xf
        ldr     q0, [x8]
        ldr     q1, [x10]
        adrp    x14, shift_data
        add     x14, x14, :lo12:shift_data

        /* heads may cross page boundary, avoid
        unmapped loads */
        tst     x5, x3
        b.eq    0f

        ldr     q4, [x14, x9]
        tbl     v0.16b, {v0.16b}, v4.16b

        .section .rodata
        .p2align 4
shift_data:
        .byte 0, 1, 2, 3, 4, 5, 6, 7
        .byte 8, 9, 10, 11, 12, 13, 14, 15
        .fill 16, 1, -1
        .size shift_data, .-shift_data
```

Aarch64

Some require imagination

- Reducing the match from 128 -> 64 bits
- No pmovmskb in Aarch64 but shrn is a good enough substitute
- Several solutions available





Simple strlen(3)

```
1:      pxor            %xmm1, %xmm1
        pcmpeqb         (%rdi), %xmm1
        pmovmskb        %xmm1, %eax
        test            %eax, %eax
        add             $16, %rdi
        jz              1b

        /* match found in loop body */
        tzcnt  %eax, %eax
        sub    %rsi, %rdi
        lea    -16(%rdi, %rax, 1), %rax
        ret
```

```
.Lloop:
        ldr    q0, [x10, #16]!
        cmeq   v0.16b, v0.16b, #0
        shrn   v0.8b, v0.8h, #4
        fcmp   d0, #0.0
        b.eq   .Lloop
        fmov   x1, d0
.Ldone:
        sub    x0, x10, x0
        rbit   x1, x1
        clz    x3, x1
        lsr    x3, x3, #2
        add    x0, x0, x3
```

Notable Alternatives

- UMAXV for the hot path then SHRN on exit
  beneficial for long strings

- PCMEQ to turn matches into 0xff, then ORR with 0, 1, ..., 15, and finally UMINV to find the index of the first mismatch (or -1 if there is none)
  beneficial for very short strings

Some require a lot of imagination

- str(c)spn(3) greatly benefits from the SSE4.2 PCMPISTRI instruction
- Really tricky to port, heavy use of *slow* tbl instruction
- Current implementation with a lookup table (LUT) for >2 byte sets
- Empty set degrades to strlen(3), 1 char set degrades to strchrnul(3)

Future work

- Implement the Muła / Langdale algorithm for Aarch64
- SVE support D43306
- Add an ARCHLEVEL flag for Aarch64
- Port to AVX2/AVX-512, SVE
- Locale stuff (nasty)
- strstr(3)
- Possible other areas that could benefit from SIMD optimizations