SIMD-enhanced libc string functions

how it's done

Robert Clausecker <fuz@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...)

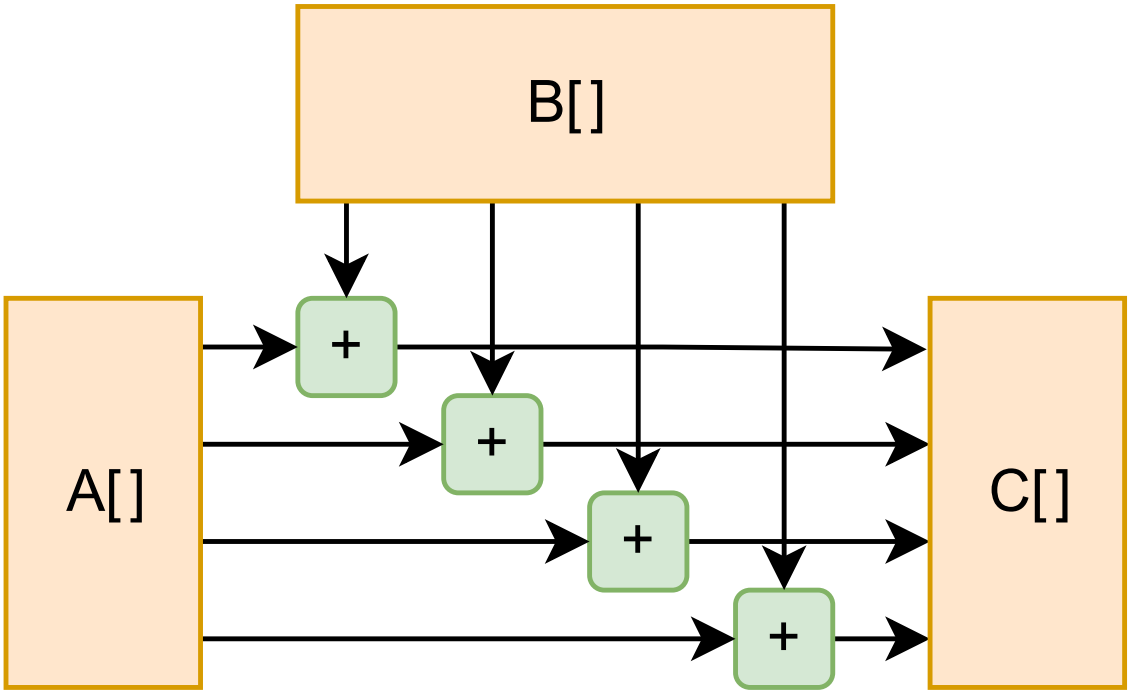
SIMD

Your new best friend

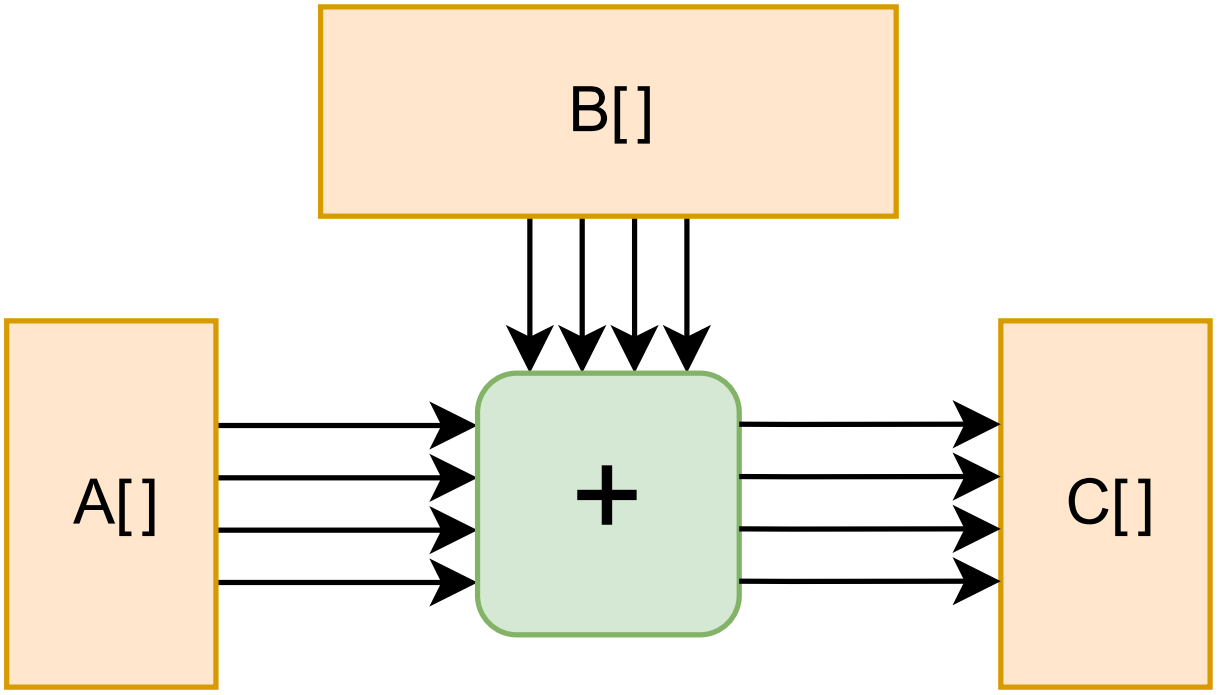
SIMD

- **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

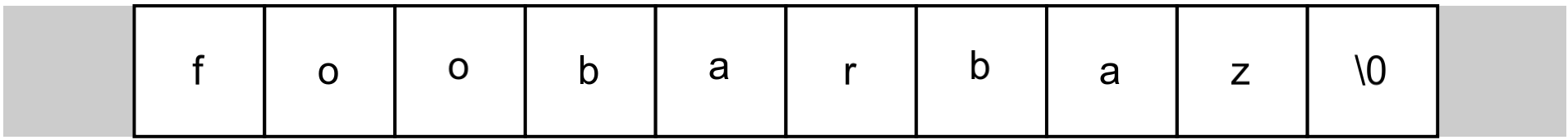
How can this help us with string processing?

Strings and SIMD

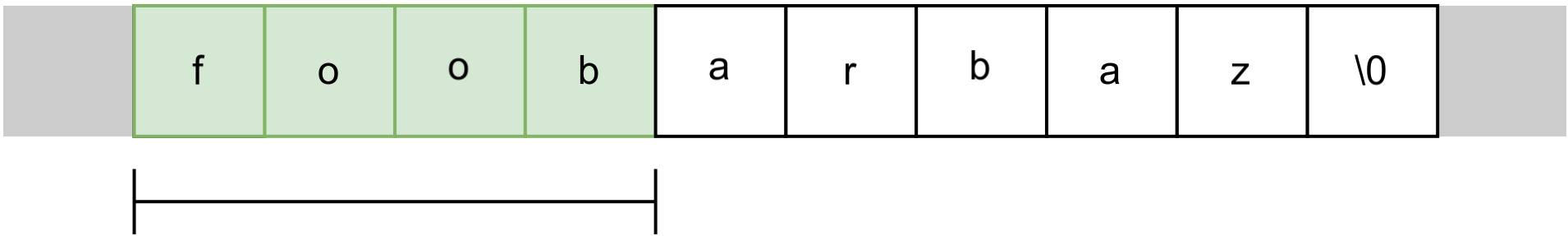
How can this help us with string processing?

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

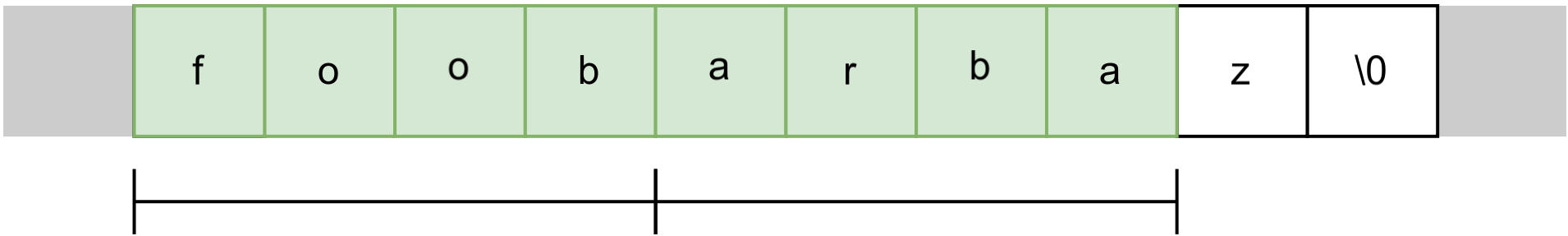
Difficulties



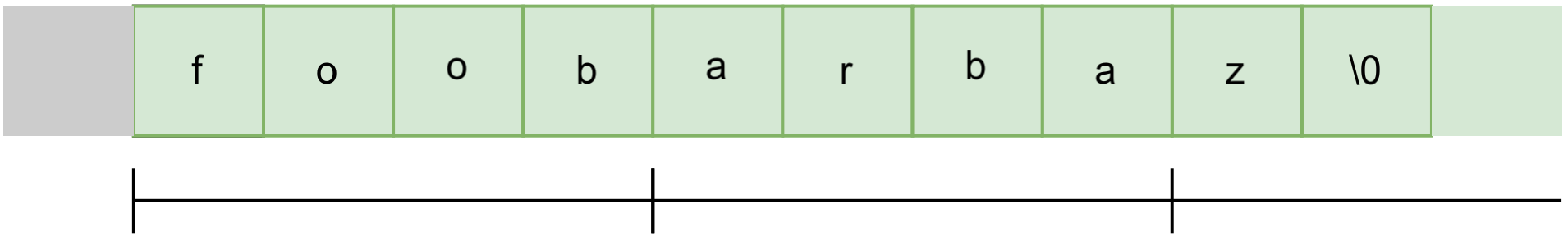
Difficulties



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)

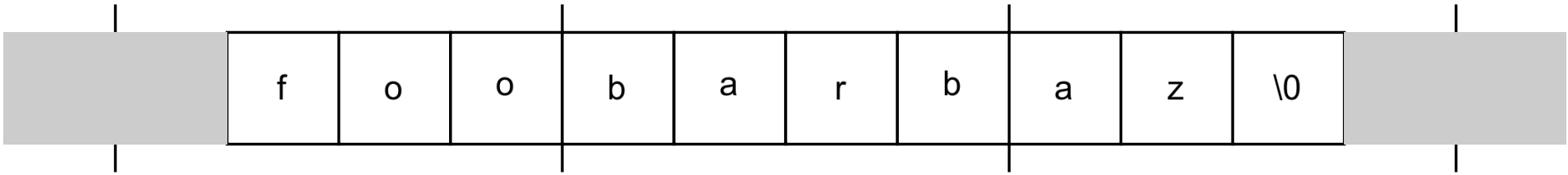
How far is too far?

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

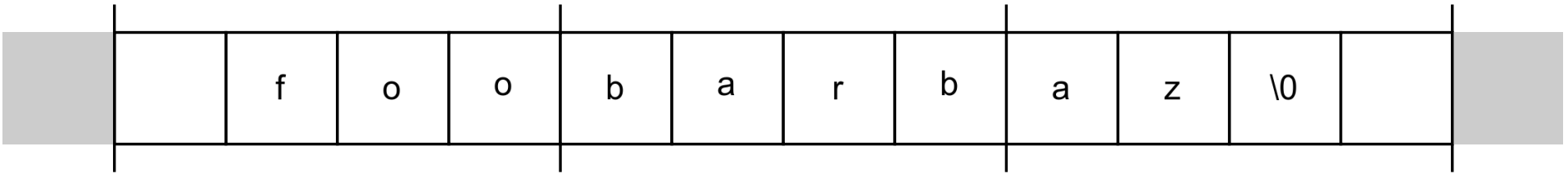
thus:

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

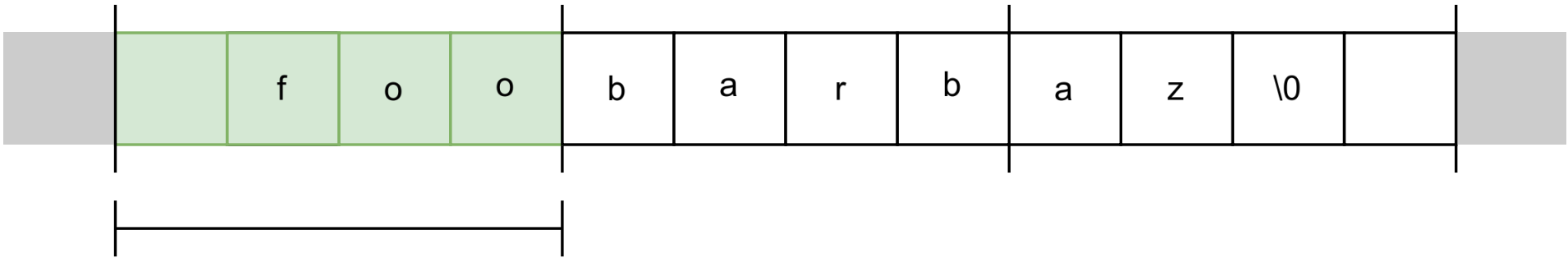
What does that look like?



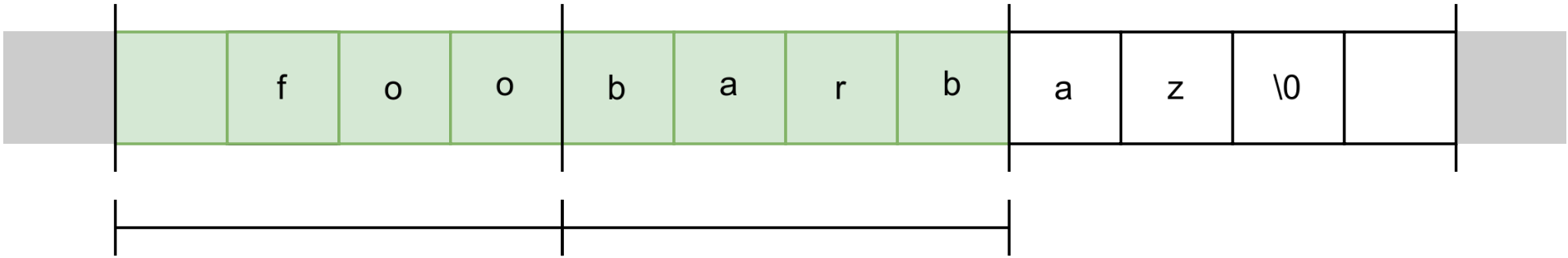
What does that look like?



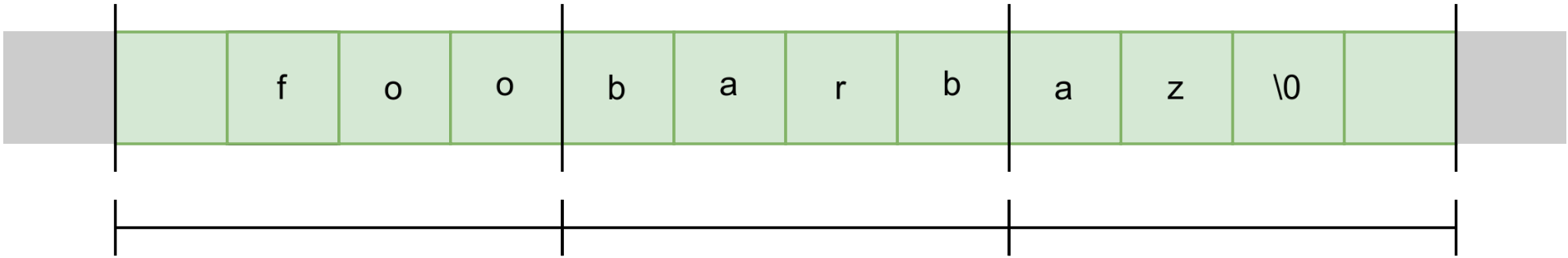
What does that look like?



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

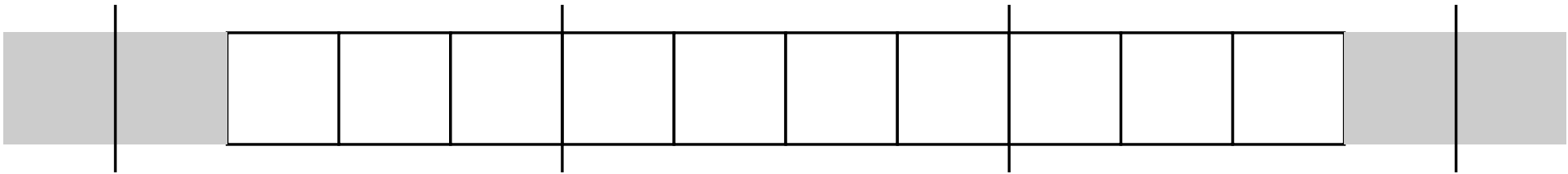
Can't use the same approach:

- overreads are fine, overwrites are no good

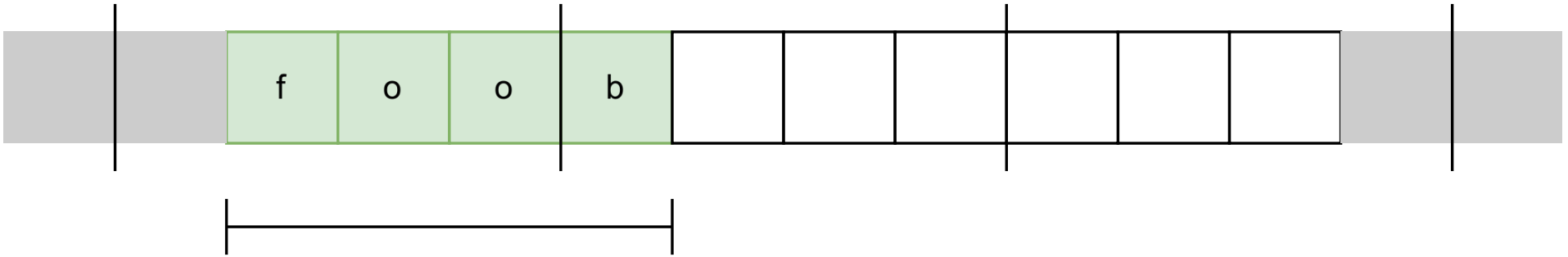
Instead

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

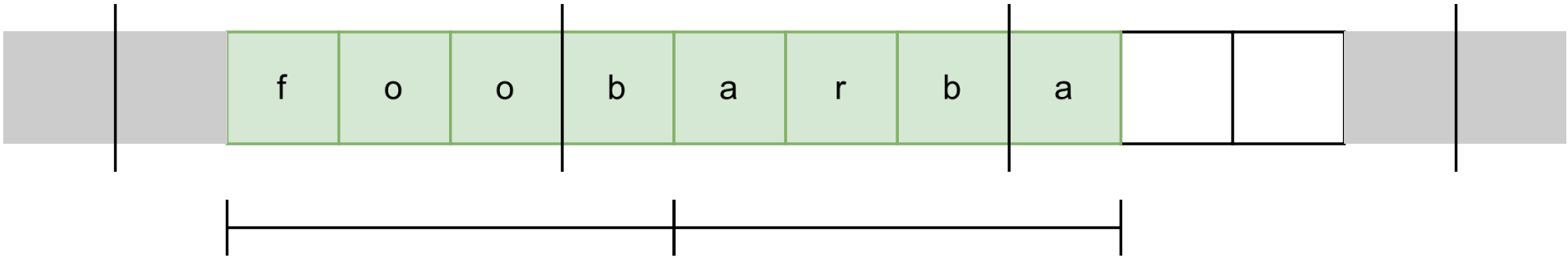
Writing Strings



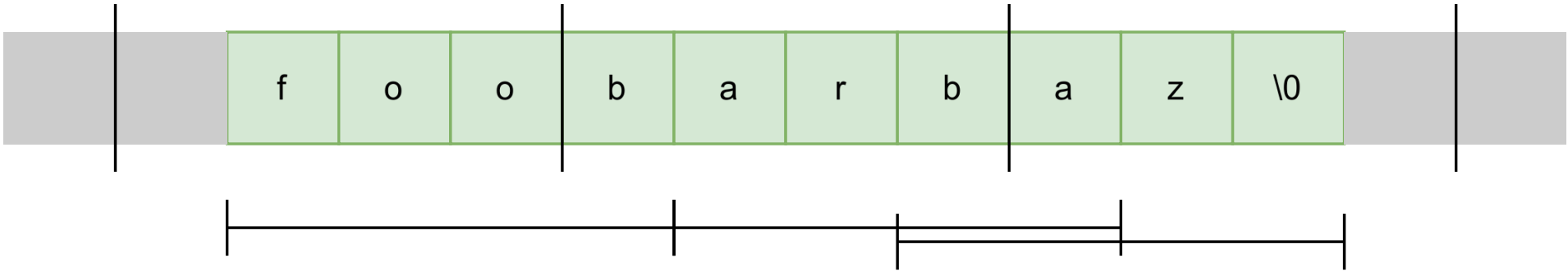
Writing Strings



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

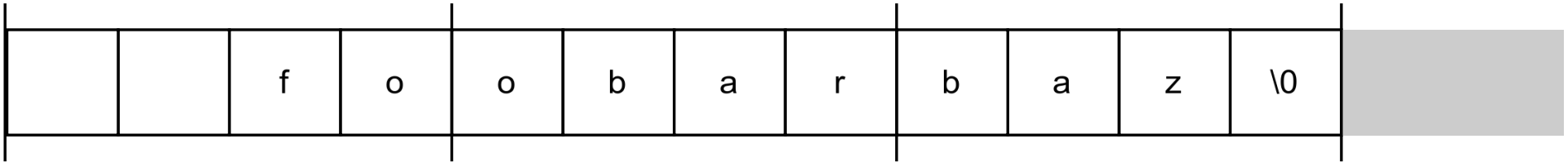
Can't use the same approach

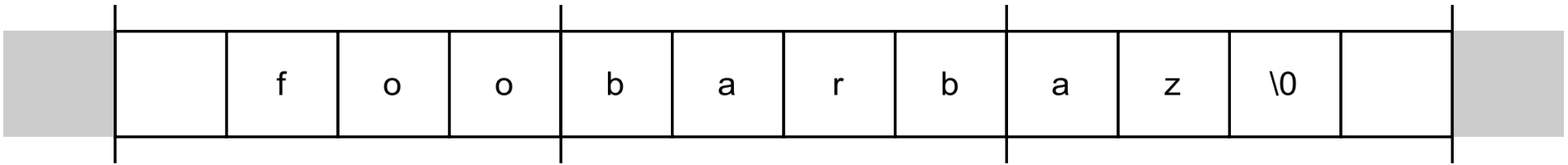
- strings may have different misalignment
- can't fix this after loading with SSE2

Instead

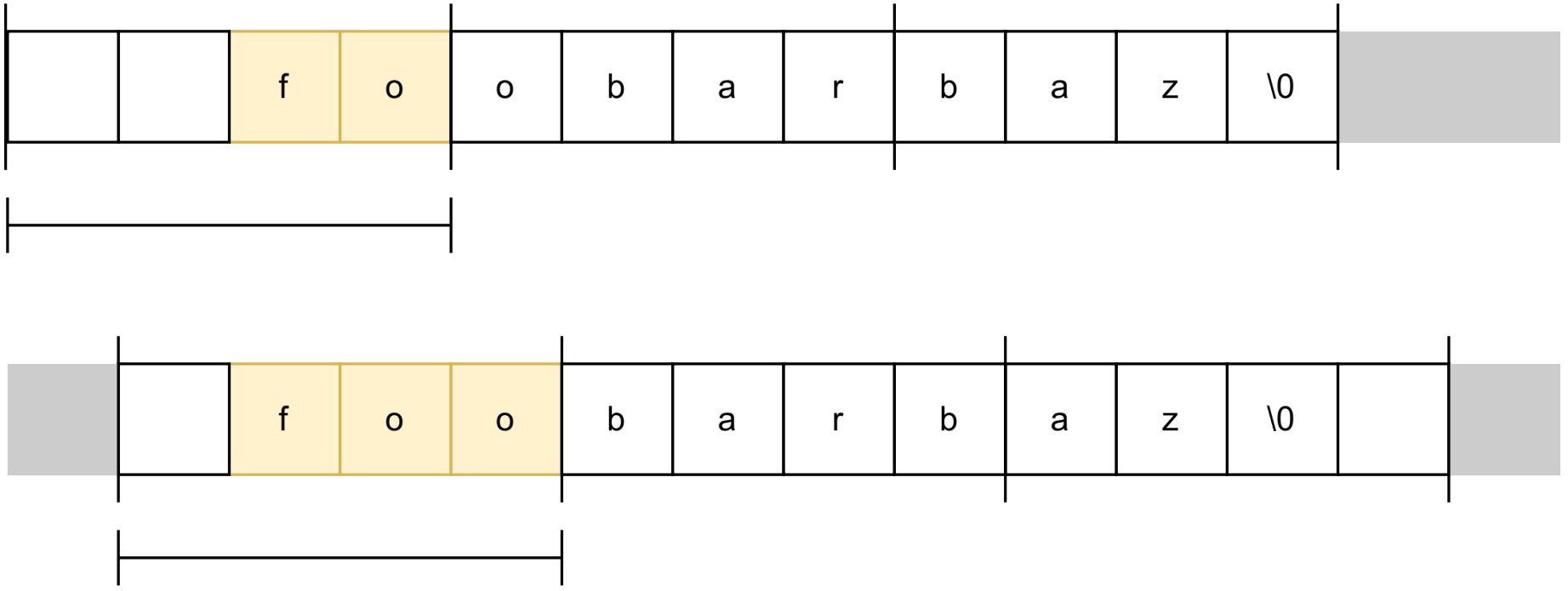
- do aligned reads to check for nul bytes
- then unaligned reads to compare characters

Comparing Strings

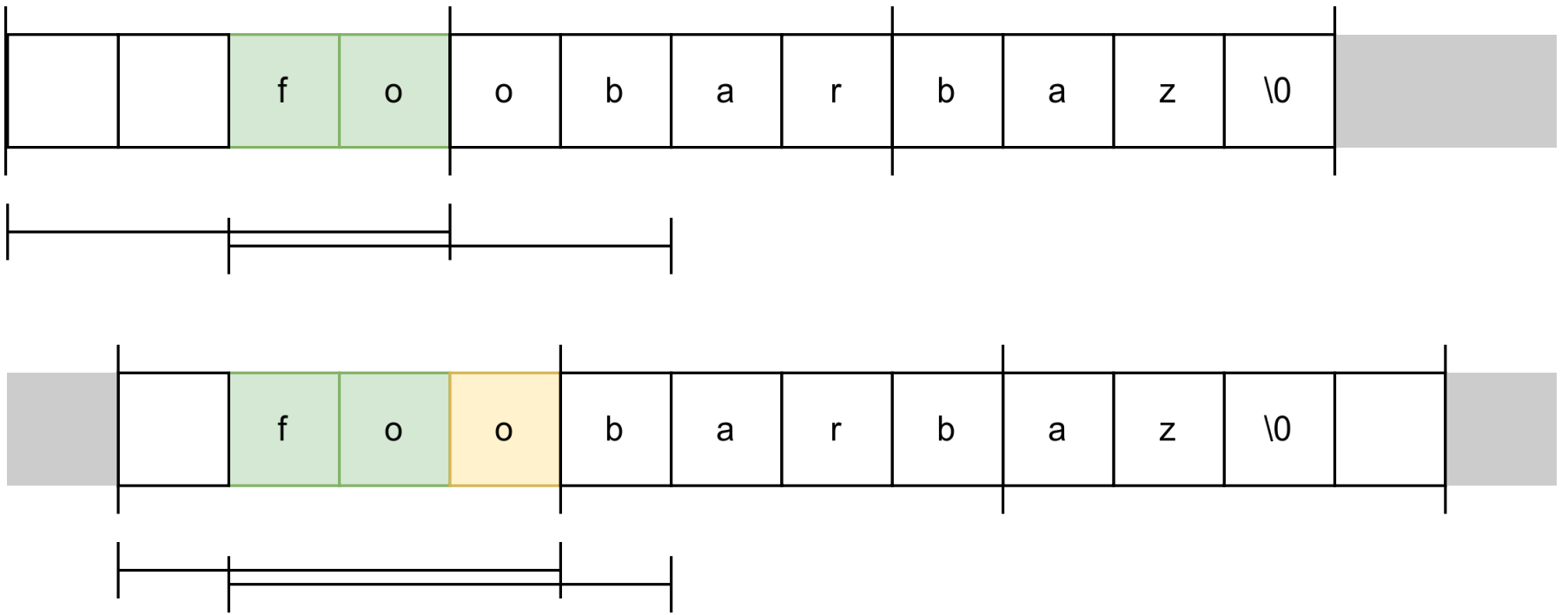




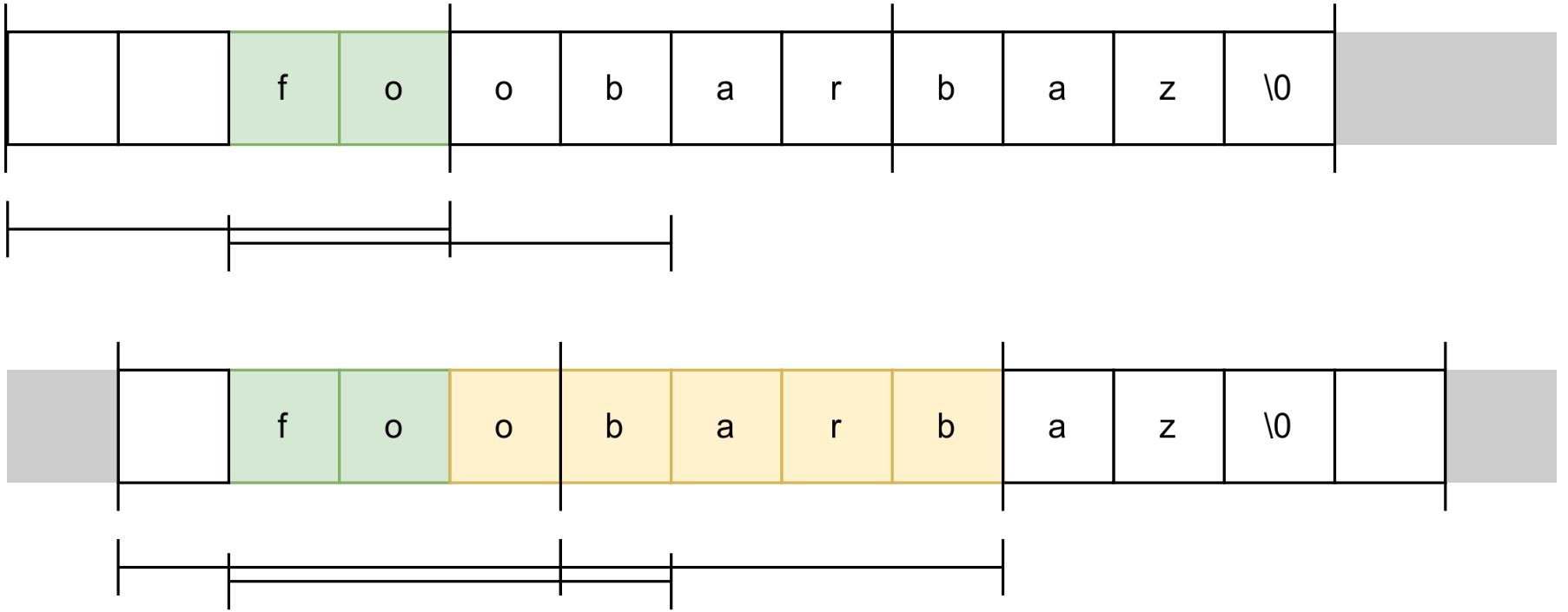
Comparing Strings



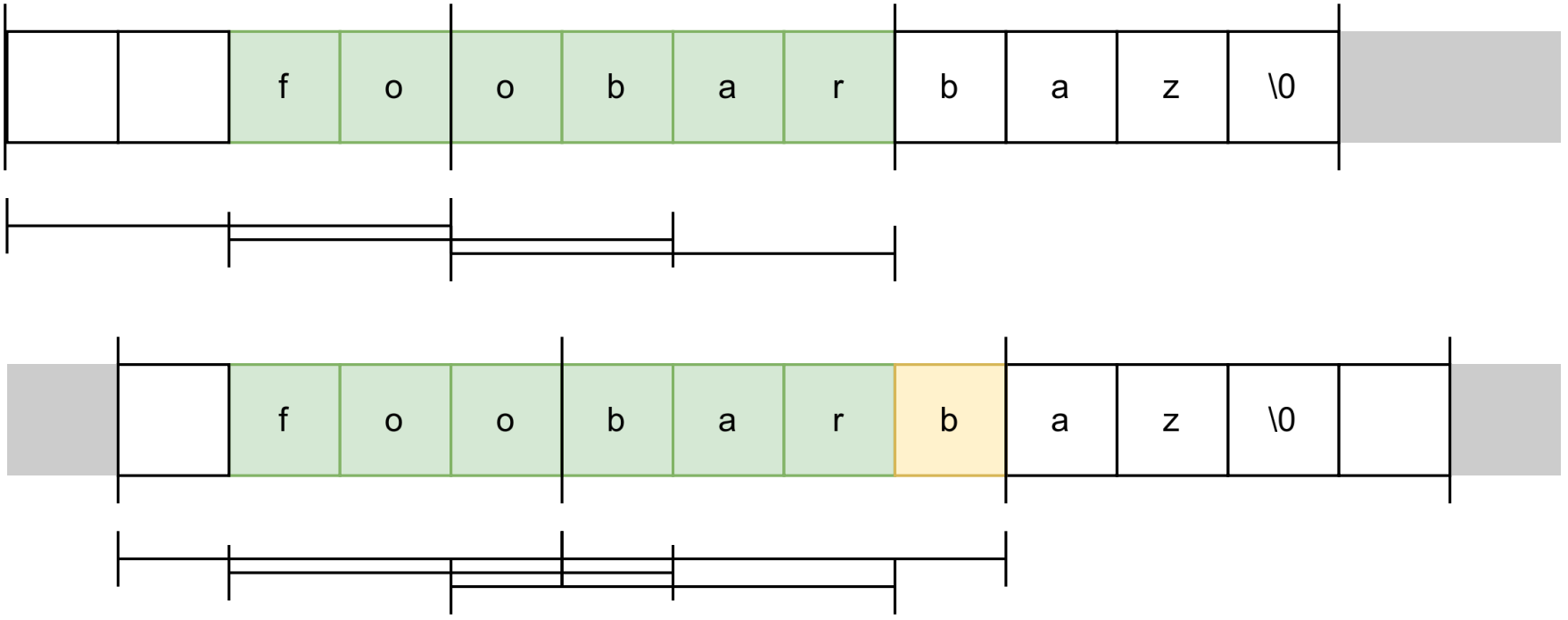
Comparing Strings



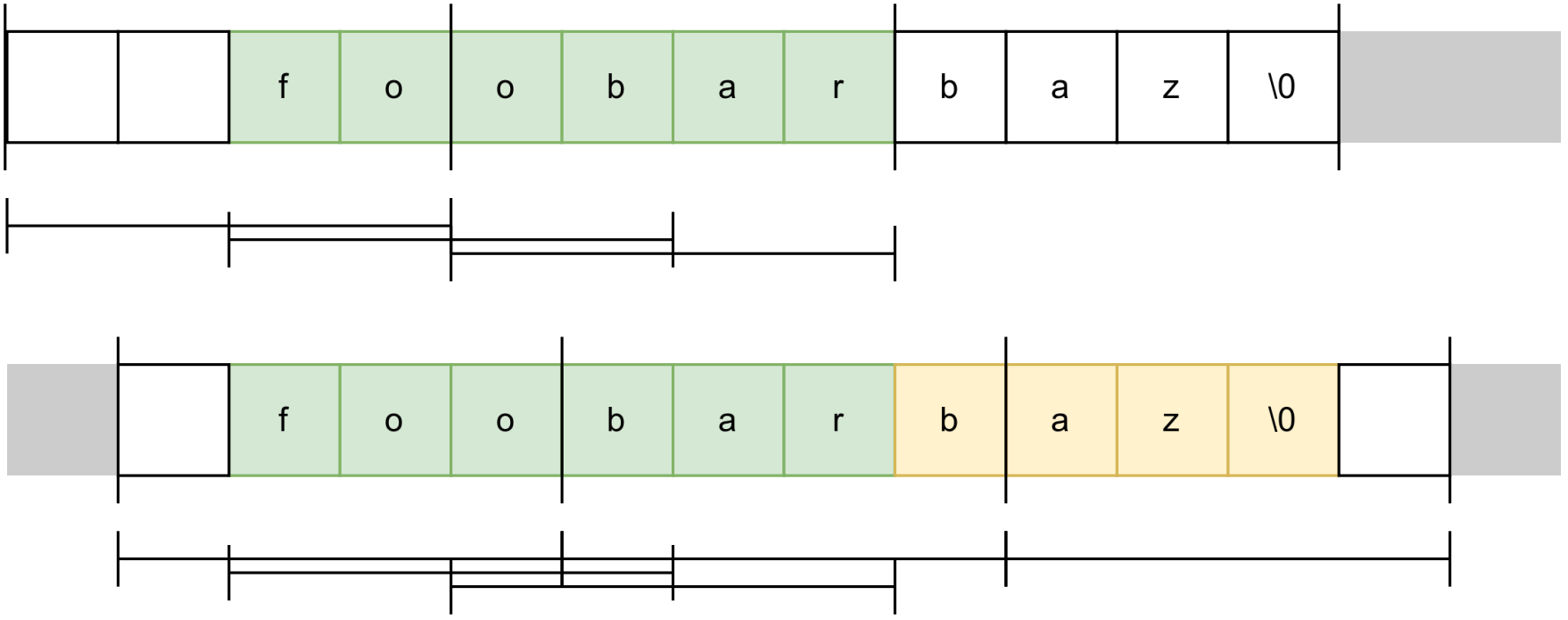
Comparing Strings



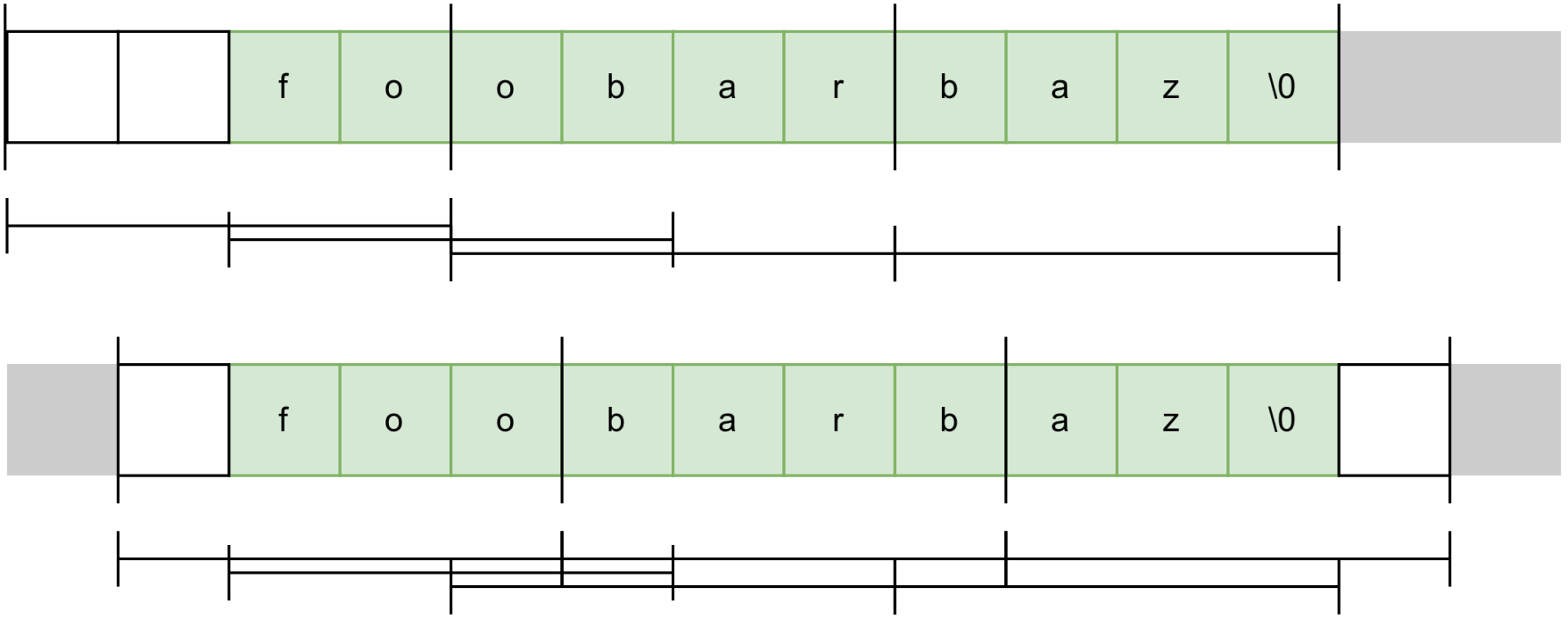
Comparing Strings



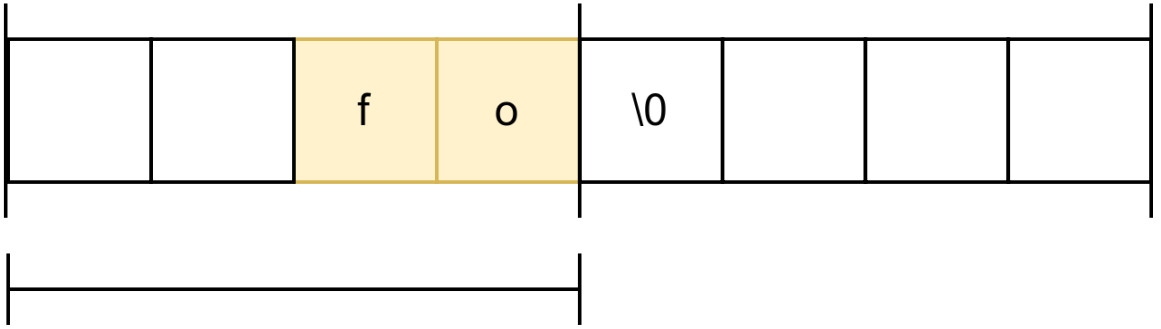
Comparing Strings

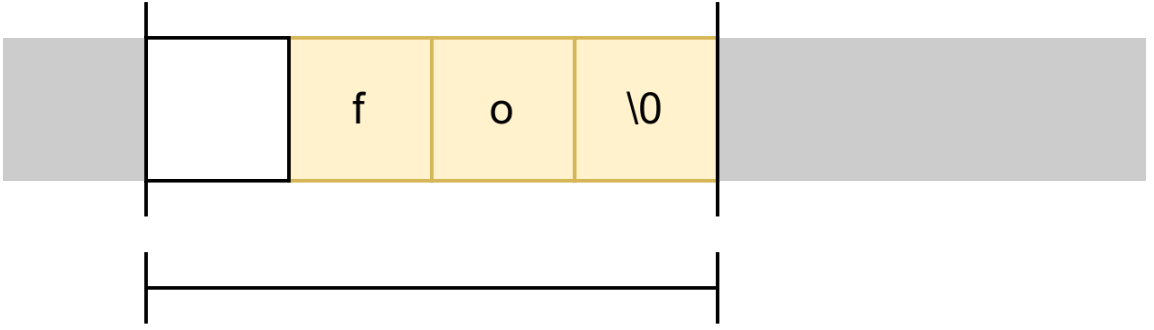


Comparing Strings

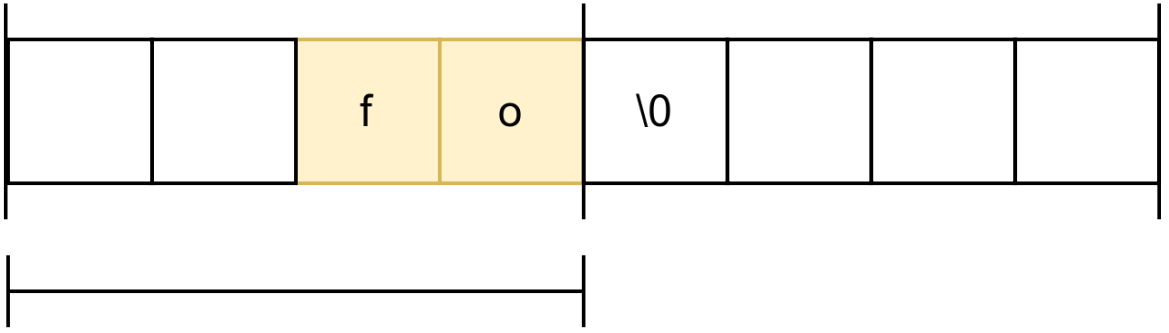


Comparing Strings

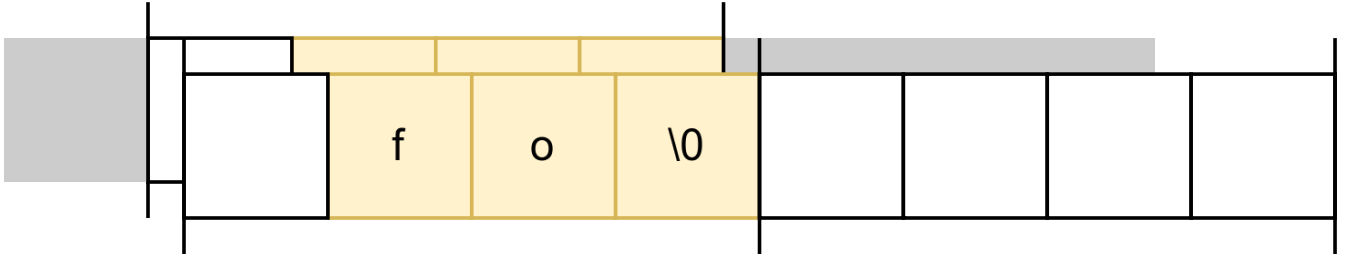




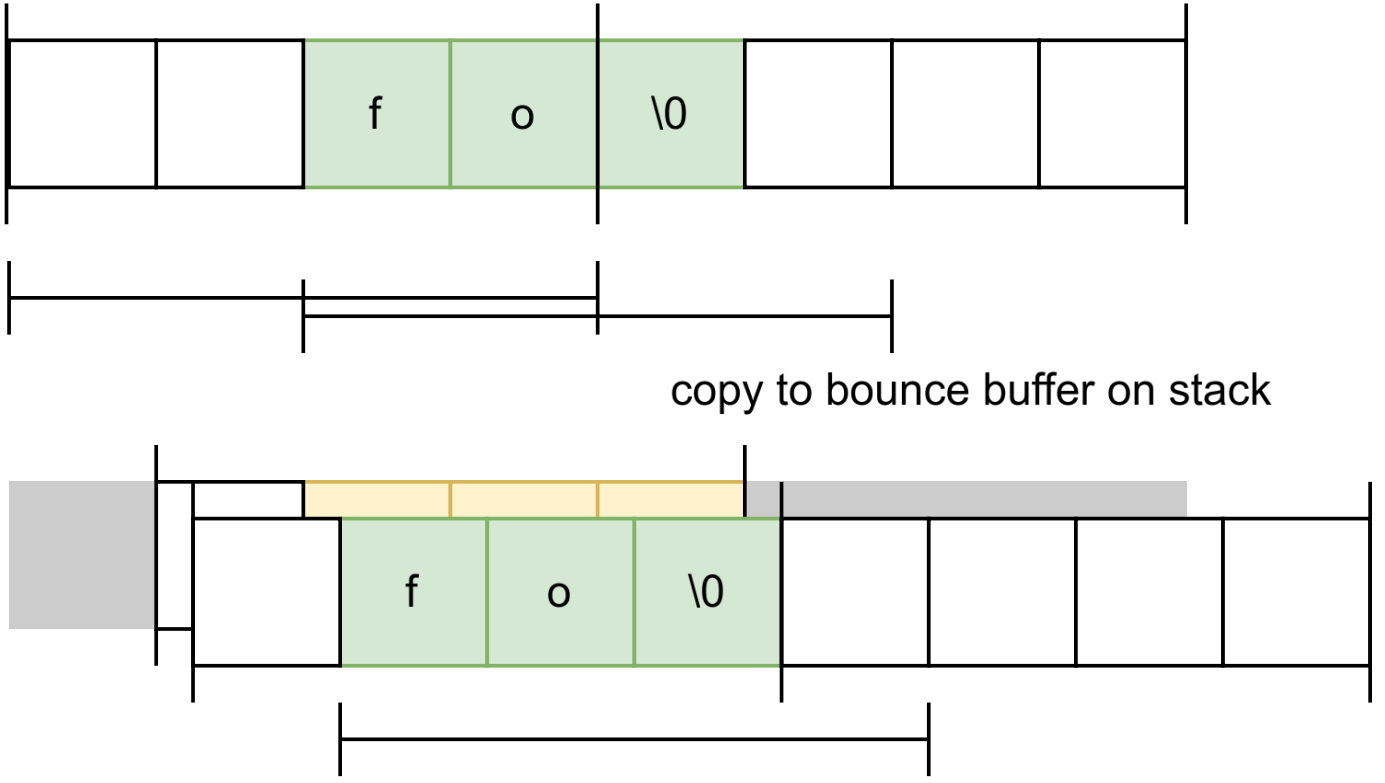
Comparing Strings



copy to bounce buffer on stack



Comparing Strings



Set Matching

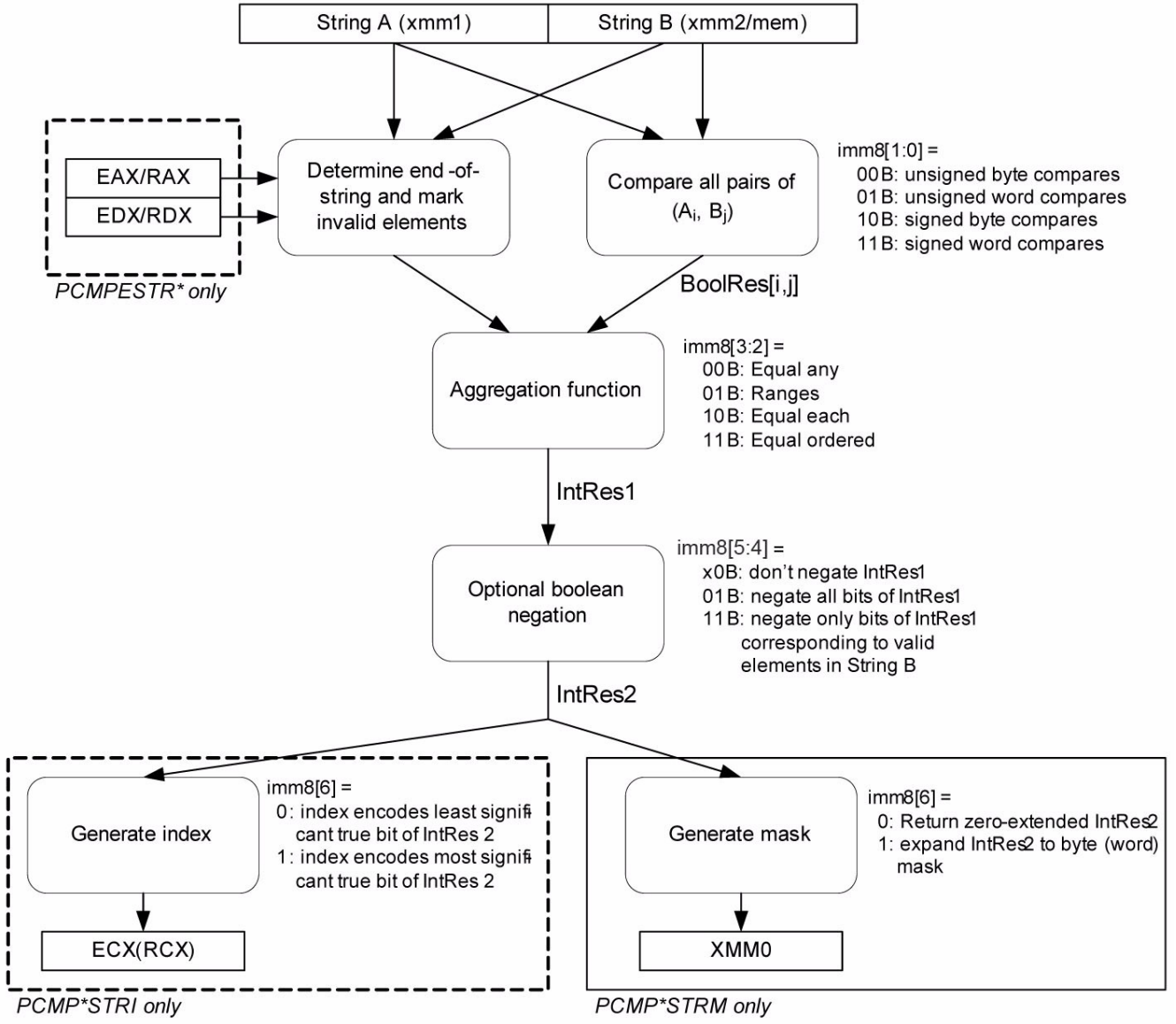
strcspn("foo bar", " \t\n");

- 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

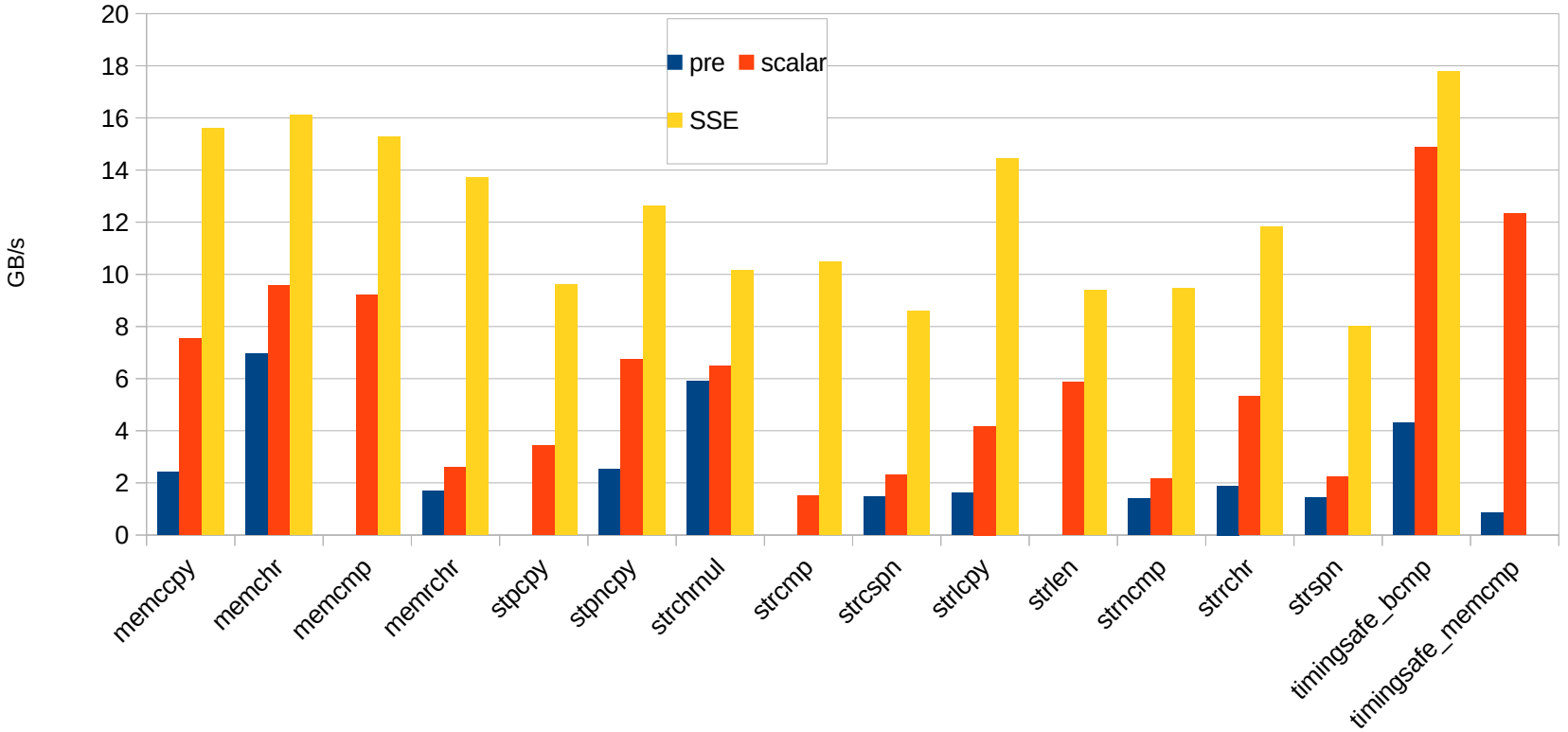

String A (xmm1)
String B (xmm2/mem)
EAX/RAX
EDX/RDX
PCMPESTR* only
Determine end -of-string and mark invalid elements
Compare all pairs of (A_i, B_j)
imm8[1:0] =
00B: unsigned byte compares
01B: unsigned word compares
10B: signed byte compares
11B: signed word compares
BoolRes[i,j]
Aggregation function
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

Results (amd64)



AArch64 port

- work by Getz Mikalsen
- complements the ARM Optimized Routines
- some functions redone due to poor perf
- others not present in ARM's codebase
- assembly functions: *strcmp*(), *strspn*(), *strcspn*(), *strlcpy*(), *strncmp*(), *memccpy*(), *strlen*(), *timingsafe_bcmp*(), *timingsafe_memcp*()
- new wrappers: *strpbrk*(), *strsep*(), *strcat*(), *strncat*(), *strlcat*(), *bcopy*(), *bzero*()

AArch64 difficulties

- no easy way to get syndrome bitmask from syndrome vector
  - amd64: *pmovmskb*
  - aarch64: *shrn* + *fmov* to get nibble mask
  - other tricks when only zero / nonzero is needed
- no *pcmpistrm*, Muła/Langdale algorithm instead
- can't count trailing zeroes, only leading zeroes

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@ (landed for upcoming 15-RELEASE)
  - riscv64 by strajabot@ (under acceptance testing)
- Future work: *strstr*(), CHERI / Morello support, PPC64?