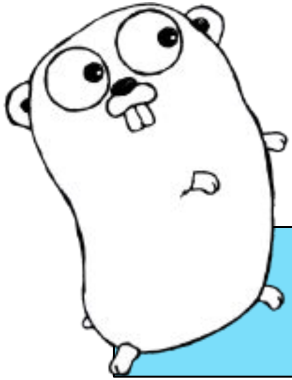
Go inline assembly

Iskander (Alex) Sharipov
@gowayfest, 2018

GoWayFest 2.0

About me

gocritic



golang

intel

open source

@quasilyte

Performance 🐌

Program never runs too fast

bytes.IndexByte performance (AMD64)

Pure Go	Assembly
1.68GB/s	48.80GB/s
2.43µs	0.08µs

Throughput: +2801.13%
Time: -96.55%

Kinds of parallelism

1. Multiple execution units (CPUs)
2. Task-level parallelism (goroutimes/threads)
3. Data-level parallelism (SIMD)

“gc” compiler does not make your program utilize DLP https://golang.org/issue/25489

The GOARCH=AMD64 arch is very limiting. Please checkout the issue linked above if you care about that

Assembly programming

What do we want

Access to fast machine-specific operations without writing tons of assembly

Inline assembly

- 90% Go code, 10% asm-like code
- Go values and expressions as an argument

Assembly

- 100% asm
- Manual registers management
- Have to deal with stack frame size, etc.

SIMD operations as a library. Write primitives in assembly once, then use them from Go.

```
// func trunc(x float64) float64
TEXT ·trunc(SB), NOSPLIT, $0-16
      MOVSD   x+0(FP), X0
      ROUNDSD $3, X0, X0
      MOVSD   X0, ret+8(FP)
      RET
```

Our math.Trunc equivalent in asm

```
// func trunc(x float64) float64
TEXT ·trunc(SB), NOSPLIT, $0-16
    - MOVSD   x+0(FP), X0
    - ROUNDSD $3, X0, X0
    + ROUNDSD $3, x+0(FP), X0
      MOVSD   X0, ret+8(FP)
      RET
```

Basically the same thing

Let’s benchmark against standard library math.Trunc function

math.Trunc performance (AMD64)

Our implementation	**Stdlib**
3.15ns	0.95ns

Time: +233.86%

Assembly functions are never inlined,
we're paying call overhead

So, what makes math/bits special?

Comparison of work performed

Our implementation	Stdlib
reigster->memory	SSE 4.1 check
ROUNDSD [memory]	ROUNDSD [register]
register->memory	
memory->register	

ABI changes?

Register-based ABI might help, but we’ll still have to use CALL

Write the whole algorithm in assembler, so data is already in registers and no function call is required?

Back to square one

Alternatives to endless assembly adventures

- Get over it. Go is pretty fast (recommended)
- Sufficiently Smart Compiler™
- More “intrinsified” packages like math/bits
- Something between Go and assembly (DIY)

GCC extended inlined assembly

- ⊖ May require special syntax
- ⊖ Almost always is too low-level
- ⊕ Unconstrained power

Intel C++ compiler style intrinsics

- ⊖ Hundreds of functions
- ⊖ Special types like m128, m256, etc.
- ⊕ Almost unconstrained power

Rust style intrinsics

⊖ Still many new symbols (functions)

⊖⊕ Somewhat higher-level

Proposed Go solution

- ⊖ Errors are reported by asm backend
- ⊖⊕ Tried to take the best of both worlds
- ⊕ Only 1 new function, unsafe.Asm
- ⊕ Simple to implement in the compiler

Why unsafe package?

What do we get

The rationale behind introducing a new feature

🎉 Tools support 🎉

Writing Go is so much more pleasant than writing ".s" prose. It also simpler to write a linter for unsafe.Asm.

🎉 Compiler cleanup 🎉

Can remove special handling of intrinsified functions and re-implement them as ordinary Go package

🔥 Fast 🔥

85-100% of performance in comparison with full assembly code

🔥 Inlineable 🔥

Makes efficient user-defined intrinsics possible

🔥 Hatch into compiler 🔥

Makes external optimizer possible (one that does auto vectorization, for example)

```
var x int64
var result int64
unsafe.Asm("MOVQ", x, 10)
unsafe.Asm("MOVQ", "AX", 20)
unsafe.Asm("ADDQ", result, x, "AX")
return result

// Can use:
// constants, Go values, registers directly
```

Simple example

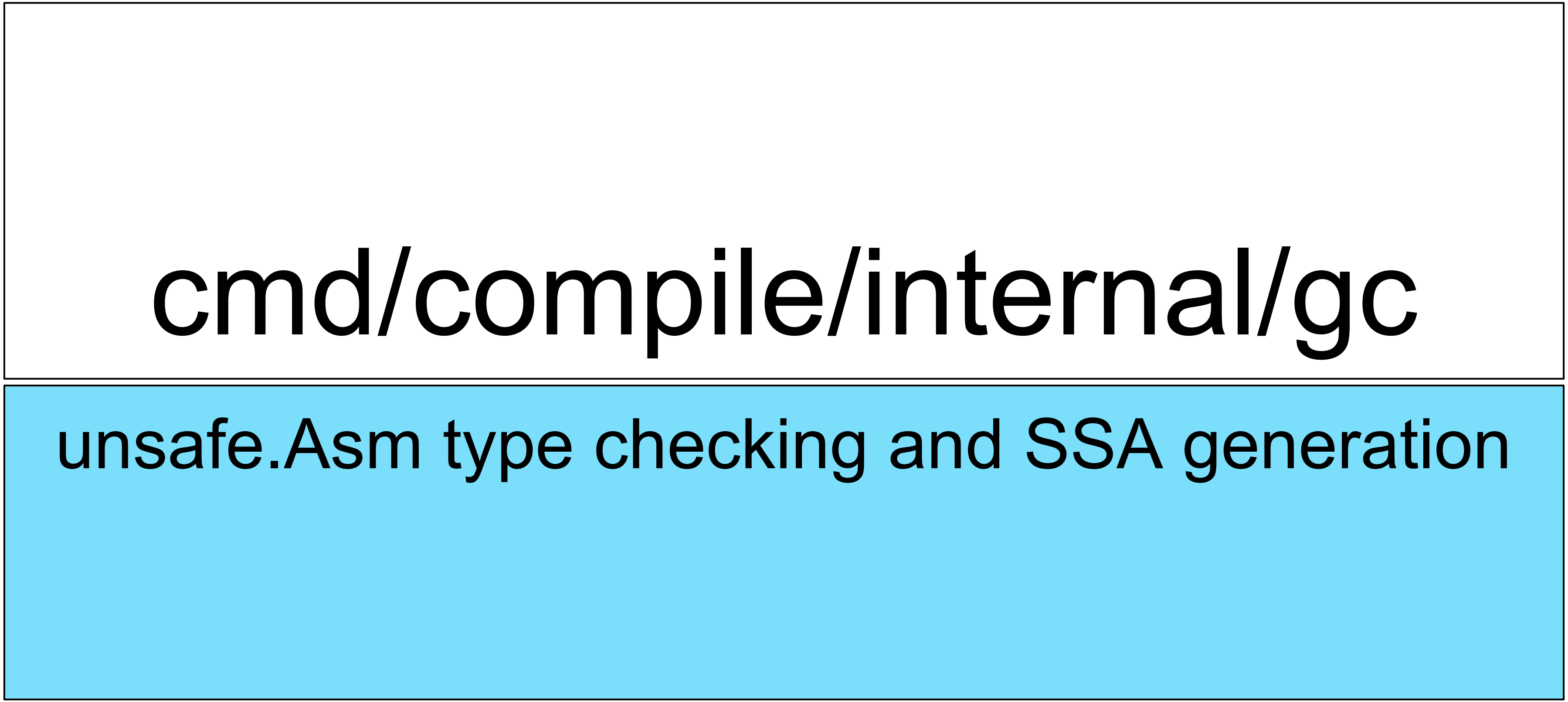
```
func Trunc(x float64) float64 {
  unsafe.Asm("ROUNDSD", x, 3, x)
  return x
}

// 100% same machine code for the function:
//    MOVSD   x+0(FP), X0
//    ROUNDSD $3, X0, X0
//    MOVSD   X0, ret+8(FP)
```

math.Trunc implementation (for x86)

Implementation

What it takes to integrate unsafe.Asm into Go



cmd/compile/internal/gc

unsafe.Asm type checking and SSA generation

cmd/compile/internal/ssa

Changes to regalloc pass plus new asm-related operations like OpAsmArg

cmd/compile/internal/amd64

Machine code generation for OpAsm

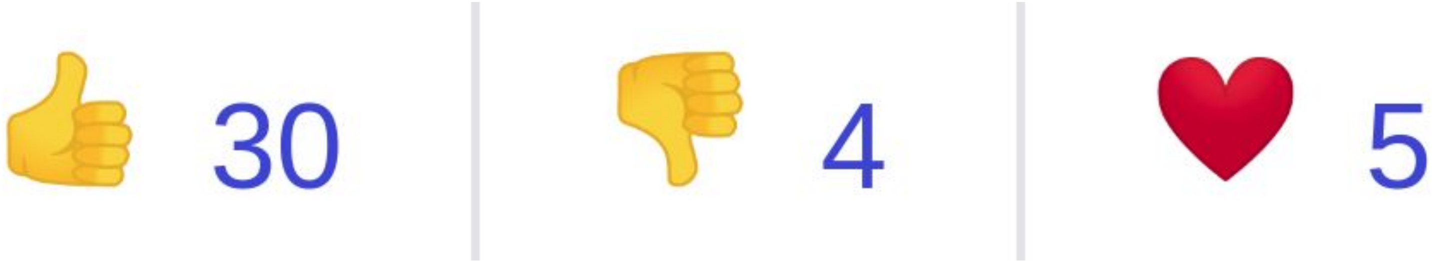
cmd/asm/internal

Parser is used to parse unsafe.Asm operand strings

Proposal

Technical document that describes proposed feature in details

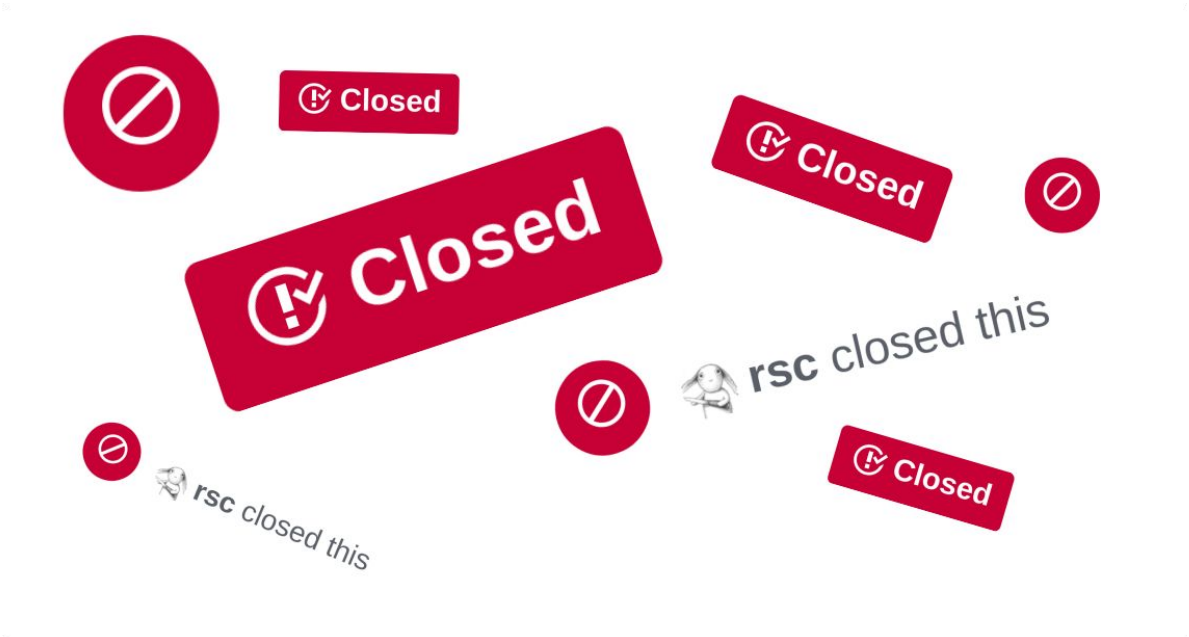
https://golang.org/issue/26891



👍 30 | 👎 4 | ❤ 5

Proposal

https://golang.org/issue/26891



Proposal

Problems and concerns

- Instructions with peculiar effects (IDIV, etc.)
- Non-gc compilers support
- Arguments evaluation between unsafe.Asm
- Unsafe feature that can be misused
- Mixes two worlds

What’s next?

Where do we move from here?

Another shot

Other approach, other solution to the performance problem. One that can accepted by the Go team

How can we ***find*** a better solution?

Performance

https://github.com/intel-go/golang

?

Experimental Go distribution

The end?