

# Better **x86** Assembly Generation with Go

---

Michael McLoughlin

Golang Bristol, March 28, 2019

Uber Advanced Technologies Group

# Introduction

---

# Assembly Language

Go provides the ability to write functions in *assembly language*.

Assembly language is a general term for *low-level* languages that allow programming at the architecture *instruction level*.



Should I write Go functions in Assembly?

No



## Go Proverbs which Might Have Been

*€go Assembly is not Go.*

My Inner Rob Pike

*With the unsafe package assembly there are no guarantees.*

Made-up Go Proverb

## Go Proverbs which Might Have Been

*€go Assembly is not Go.*

My Inner Rob Pike

*With the unsafe package assembly there are no guarantees.*

Made-up Go Proverb



*We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%.*

Donald Knuth, 1974

*We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*

Donald Knuth, 1974

## The Critical 3%?

To take advantage of:

- Missed optimizations by the compiler
- Special hardware instructions

Common use cases:

- Math compute kernels
- System Calls
- Low-level Runtime Details
- Cryptography



**CAUTION  
avalanche danger**

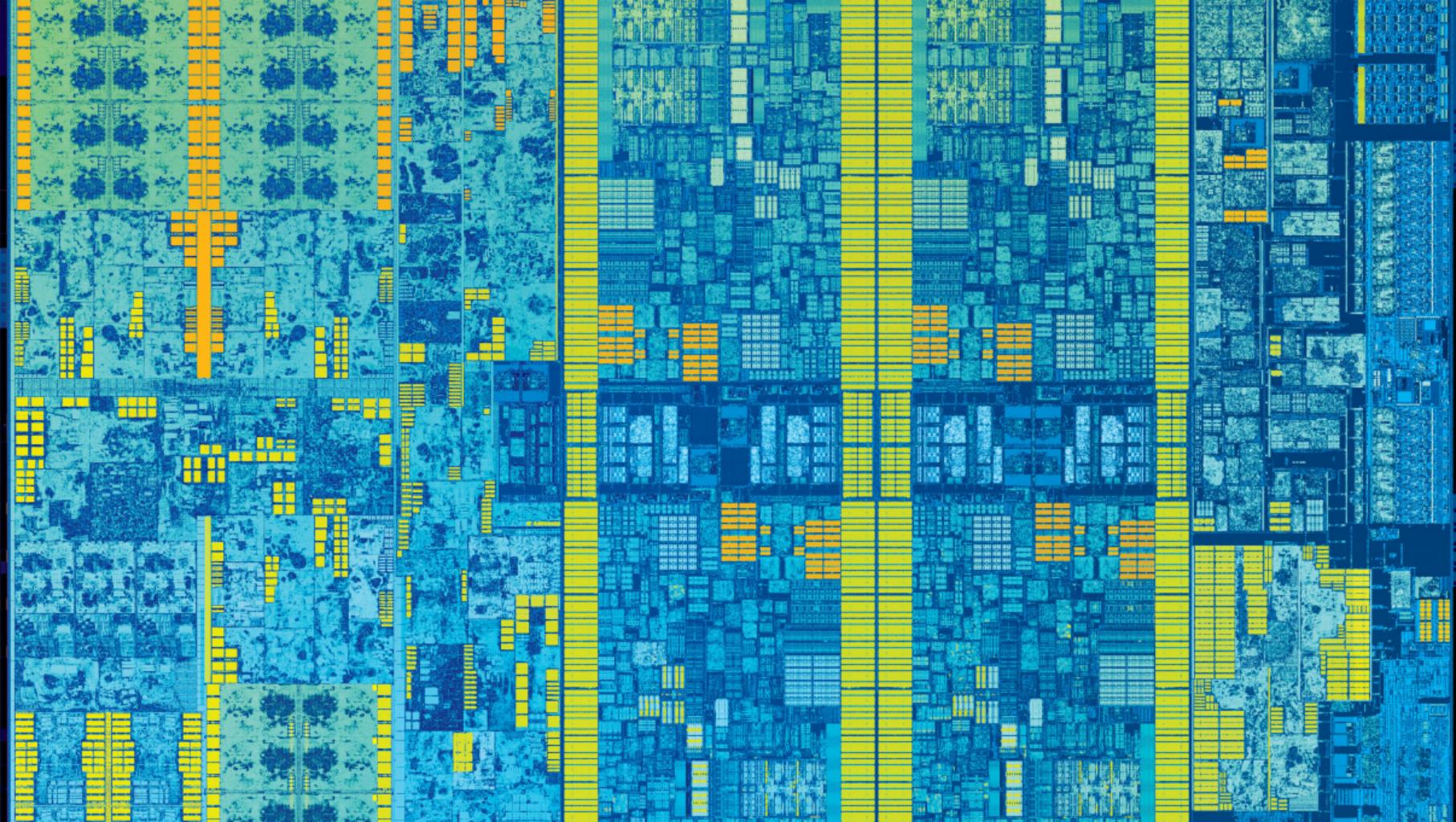


# Outline

- Intro to x86 assembly in Go
- Problem Statement
- Code Generation
- The `avo` Library

# Learn **x86** in Two Minutes

---



## General Purpose Registers: 64-bits

RAX
RCX
RDX
RBX
RSP
RBP
RSI
RDI
R8
R9
R10
R11
R12
R13
R14
R15

## General Purpose Registers: 32-bits

RAX	EAX
RCX	ECX
RDX	EDX
RBX	EBX
RSP	ESP
RBP	EBP
RSI	ESI
RDI	EDI
R8	R8L
R9	R9L
R10	R10L
R11	R11L
R12	R12L
R13	R13L
R14	R14L
R15	R15L

## General Purpose Registers: 16-bits

RAX	EAX	AX
RCX	ECX	CX
RDX	EDX	DX
RBX	EBX	BX
RSP	ESP	SP
RBP	EBP	BP
RSI	ESI	SI
RDI	EDI	DI
R8	R8L	R8W
R9	R9L	R9W
R10	R10L	R10W
R11	R11L	R11W
R12	R12L	R12W
R13	R13L	R13W
R14	R14L	R14W
R15	R15L	R15W

## General Purpose Registers: 8-bits

RAX	EAX	AX	AL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RBX	EBX	BX	BL
RSP	ESP	SP	SPB
RBP	EBP	BP	BPB
RSI	ESI	SI	SIB
RDI	EDI	DI	DIB
R8	R8L	R8W	R8B
R9	R9L	R9W	R9B
R10	R10L	R10W	R10B
R11	R11L	R11W	R11B
R12	R12L	R12W	R12B
R13	R13L	R13W	R13B
R14	R14L	R14W	R14B
R15	R15L	R15W	R15B

Load Immediate Value

MOVQ \$2, AX

Load 2 into AX

## Load From Memory

MOVQ 8(CX), R8

Load memory offset CX + 8 into R8

## Instruction

ADDQ AX, R8

Add AX to R8 and place result in R8

## Special Instructions

Vector Registers:

- **XMM** 128-bit wide
- **YMM** 256-bit wide
- **ZMM** 512-bit wide

Special instructions:

- Vector instructions: SSE, AVX, AVX2, AVX-512
- One-offs: BMI, AES-NI, ...

# Go Assembly Primer

---

# Hello, World!

```
package add

// Add x and y.
func Add(x, y uint64) uint64 {
    return x + y
}
```

## Go Disassembler

The Go disassembler may be used to inspect generated machine code.

```
go build -o add.a  
go tool objdump add.a
```

```
TEXT %22%22.Add(SB) gofile..../Users/michaelmcloughlin/Dev...
add.go:5    0x2e7      488b442410    MOVQ 0x10(SP), AX
add.go:5    0x2ec      488b4c2408    MOVQ 0x8(SP), CX
add.go:5    0x2f1      4801c8        ADDQ CX, AX
add.go:5    0x2f4      4889442418    MOVQ AX, 0x18(SP)
add.go:5    0x2f9      c3           RET
```

```
TEXT %22%22.Add(SB)  gofile..../Users/michaelmcloughlin/Dev...
add.go:5      0x2e7      488b442410      MOVQ 0x10(SP), AX
add.go:5      0x2ec      488b4c2408      MOVQ 0x8(SP), CX
add.go:5      0x2f1      4801c8          ADDQ CX, AX
add.go:5      0x2f4      4889442418      MOVQ AX, 0x18(SP)
add.go:5      0x2f9      c3              RET
```

## Function Stubs

```
package add

// Add x and y.
func Add(x, y uint64) uint64
```

Missing function body will be implemented in assembly.

Implementation provided in `add_amd64.s`.

```
#include "textflag.h"

// func Add(x, y uint64) uint64
TEXT ·Add(SB), NOSPLIT, $0-24
    MOVQ x+0(FP), AX
    MOVQ y+8(FP), CX
    ADDQ CX, AX
    MOVQ AX, ret+16(FP)
    RET
```

Implementation provided in `add_amd64.s`.

```
#include "textflag.h"

// func Add(x, y uint64) uint64
TEXT ·Add(SB), NOSPLIT, $0-24          ← Declaration
    MOVQ x+0(FP), AX
    MOVQ y+8(FP), CX
    ADDQ CX, AX
    MOVQ AX, ret+16(FP)
    RET
```

Implementation provided in `add_amd64.s`.

```
#include "textflag.h"

// func Add(x, y uint64) uint64
TEXT ·Add(SB), NOSPLIT, $0-24
    MOVQ x+0(FP), AX          < Read x from stack frame
    MOVQ y+8(FP), CX          < Read y
    ADDQ CX, AX
    MOVQ AX, ret+16(FP)
    RET
```



Implementation provided in `add_amd64.s`.

```
#include "textflag.h"

// func Add(x, y uint64) uint64
TEXT ·Add(SB), NOSPLIT, $0-24
    MOVQ x+0(FP), AX
    MOVQ y+8(FP), CX
    ADDQ CX, AX
    MOVQ AX, ret+16(FP)           < Write return value
    RET
```

# Problem Statement

---

24,962

**Table 1:** Assembly Lines by Top-Level Packages

Lines	Package
8140	<code>crypto</code>
8069	<code>runtime</code>
5686	<code>internal</code>
1173	<code>math</code>
1005	<code>syscall</code>
574	<code>cmd</code>
279	<code>hash</code>
36	<code>reflect</code>



Table 2: Top 10 Assembly Files by Lines

---

Lines	File
2695	internal/x/crypto/.../chacha20poly1305_amd64.s
2348	crypto/elliptic/p256_asm_amd64.s
1632	runtime/asm_amd64.s
1500	crypto/sha1/sha1block_amd64.s
1468	crypto/sha512/sha512block_amd64.s
1377	internal/x/crypto/curve25519/ladderstep_amd64.s
1286	crypto/aes/gcm_amd64.s
1031	crypto/sha256/sha256block_amd64.s
743	runtime/sys_darwin_amd64.s
727	runtime/sys_linux_amd64.s

---

Table 2: Top 10 Assembly Files by Lines

---

Lines	File
2695	internal/x/ <b>crypto</b> /.../chacha20poly1305_amd64.s
2348	<b>crypto/elliptic/p256_asm_amd64.s</b>
1632	runtime/asm_amd64.s
1500	<b>crypto/sha1/sha1block_amd64.s</b>
1468	<b>crypto/sha512/sha512block_amd64.s</b>
1377	internal/x/ <b>crypto/curve25519/ladderstep_amd64.s</b>
1286	<b>crypto/aes/gcm_amd64.s</b>
1031	<b>crypto/sha256/sha256block_amd64.s</b>
743	runtime/sys_darwin_amd64.s
727	runtime/sys_linux_amd64.s

---



```

// Special optimization for buffers smaller than 321 bytes
openAVX2320:
    // For up to 320 bytes of ciphertext and 64 bytes for the poly key, we process six blocks
    VMOVDQA AA0, AA1; VMOVDQA BB0, BB1; VMOVDQA CC0, CC1; VPADD .avx2IncMask<>(SB), DD0, DD1
    VMOVDQA AA0, AA2; VMOVDQA BB0, BB2; VMOVDQA CC0, CC2; VPADD .avx2IncMask<>(SB), DD1, DD2
    VMOVDQA BB0, TT1; VMOVDQA CC0, TT2; VMOVDQA DD0, TT3
    MOVQ    $10, itr2

openAVX2320InnerCipherLoop:
    chachaQR_AVX2(AA0, BB0, CC0, DD0, TT0); chachaQR_AVX2(AA1, BB1, CC1, DD1, TT0); chachaQR_AVX2(AA2, BB2, CC2, DD2, TT2)
    VPALIGNR $4, BB0, BB0, BB0; VPALIGNR $4, BB1, BB1, BB1; VPALIGNR $4, BB2, BB2, BB2
    VPALIGNR $8, CC0, CC0, CC0; VPALIGNR $8, CC1, CC1, CC1; VPALIGNR $8, CC2, CC2, CC2
    VPALIGNR $12, DD0, DD0, DD0; VPALIGNR $12, DD1, DD1, DD1; VPALIGNR $12, DD2, DD2, DD2
    chachaQR_AVX2(AA0, BB0, CC0, DD0, TT0); chachaQR_AVX2(AA1, BB1, CC1, DD1, TT0); chachaQR_AVX2(AA2, BB2, CC2, DD2, TT2)
    VPALIGNR $12, BB0, BB0, BB0; VPALIGNR $12, BB1, BB1, BB1; VPALIGNR $12, BB2, BB2, BB2
    VPALIGNR $8, CC0, CC0, CC0; VPALIGNR $8, CC1, CC1, CC1; VPALIGNR $8, CC2, CC2, CC2
    VPALIGNR $4, DD0, DD0, DD0; VPALIGNR $4, DD1, DD1, DD1; VPALIGNR $4, DD2, DD2, DD2
    DECQ    itr2
    JNE     openAVX2320InnerCipherLoop

    VMOVDQA .chacha20Constants<>(SB), TT0
    VPADD TT0, AA0, AA0; VPADD TT0, AA1, AA1; VPADD TT0, AA2, AA2
    VPADD TT1, BB0, BB0; VPADD TT1, BB1, BB1; VPADD TT1, BB2, BB2
    VPADD TT2, CC0, CC0; VPADD TT2, CC1, CC1; VPADD TT2, CC2, CC2

```

internal/x/.../chacha20poly1305\_amd64.s lines 1072-1095 (go1.12)

```

openAVX2Tail512LoopA:
    VPADD    BB0, AA0, AA0; VPADD    BB1, AA1, AA1; VPADD    BB2, AA2, AA2; VPADD    BB3, AA3, AA3
    VPXOR    AA0, DD0, DD0; VPXOR    AA1, DD1, DD1; VPXOR    AA2, DD2, DD2; VPXOR    AA3, DD3, DD3
    VPSHUFB ·rol16<>(SB), DD0, DD0; VPSHUFB ·rol16<>(SB), DD1, DD1; VPSHUFB ·rol16<>(SB), DD2, DD2; VPSHUFB ·rol16<>(SB),
    VPADD    DD0, CC0, CC0; VPADD    DD1, CC1, CC1; VPADD    DD2, CC2, CC2; VPADD    DD3, CC3, CC3
    VPXOR    CC0, BB0, BB0; VPXOR    CC1, BB1, BB1; VPXOR    CC2, BB2, BB2; VPXOR    CC3, BB3, BB3
    VMOVEDQA CC3, tmpStoreAVX2
    VPSLLD   $12, BB0, CC3; VPSRLD   $20, BB0, BB0; VPXOR    CC3, BB0, BB0
    VPSLLD   $12, BB1, CC3; VPSRLD   $20, BB1, BB1; VPXOR    CC3, BB1, BB1
    VPSLLD   $12, BB2, CC3; VPSRLD   $20, BB2, BB2; VPXOR    CC3, BB2, BB2
    VPSLLD   $12, BB3, CC3; VPSRLD   $20, BB3, BB3; VPXOR    CC3, BB3, BB3
    VMOVEDQA tmpStoreAVX2, CC3
polyAdd(0*8(itr2))
polyMulAVX2
    VPADD    BB0, AA0, AA0; VPADD    BB1, AA1, AA1; VPADD    BB2, AA2, AA2; VPADD    BB3, AA3, AA3
    VPXOR    AA0, DD0, DD0; VPXOR    AA1, DD1, DD1; VPXOR    AA2, DD2, DD2; VPXOR    AA3, DD3, DD3
    VPSHUFB ·rol8<>(SB), DD0, DD0; VPSHUFB ·rol8<>(SB), DD1, DD1; VPSHUFB ·rol8<>(SB), DD2, DD2; VPSHUFB ·rol8<>(SB),
    VPADD    DD0, CC0, CC0; VPADD    DD1, CC1, CC1; VPADD    DD2, CC2, CC2; VPADD    DD3, CC3, CC3
    VPXOR    CC0, BB0, BB0; VPXOR    CC1, BB1, BB1; VPXOR    CC2, BB2, BB2; VPXOR    CC3, BB3, BB3
    VMOVEDQA CC3, tmpStoreAVX2
    VPSLLD   $7, BB0, CC3; VPSRLD   $25, BB0, BB0; VPXOR    CC3, BB0, BB0
    VPSLLD   $7, BB1, CC3; VPSRLD   $25, BB1, BB1; VPXOR    CC3, BB1, BB1
    VPSLLD   $7, BB2, CC3; VPSRLD   $25, BB2, BB2; VPXOR    CC3, BB2, BB2
    VPSLLD   $7, BB3, CC3; VPSRLD   $25, BB3, BB3; VPXOR    CC3, BB3, BB3

```

internal/x/.../chacha20poly1305\_amd64.s lines 1374-1397 (go1.12)

```
sealAVX2Tail512LoopB:
    VPADD    BB0, AA0, AA0; VPADD    BB1, AA1, AA1; VPADD    BB2, AA2, AA2; VPADD    BB3, AA3, AA3
    VPXOR    AA0, DD0, DD0; VPXOR    AA1, DD1, DD1; VPXOR    AA2, DD2, DD2; VPXOR    AA3, DD3, DD3
    VPSHUFB ·rol16<>(SB), DD0, DD0; VPSHUFB ·rol16<>(SB), DD1, DD1; VPSHUFB ·rol16<>(SB), DD2, DD2; VPSHUFB ·rol16<>(SB),
    VPADD    DD0, CC0, CC0; VPADD    DD1, CC1, CC1; VPADD    DD2, CC2, CC2; VPADD    DD3, CC3, CC3
    VPXOR    CC0, BB0, BB0; VPXOR    CC1, BB1, BB1; VPXOR    CC2, BB2, BB2; VPXOR    CC3, BB3, BB3
    VMOVEDQA CC3, tmpStoreAVX2
    VPSLLD   $12, BB0, CC3; VPSRLD   $20, BB0, BB0; VPXOR    CC3, BB0, BB0
    VPSLLD   $12, BB1, CC3; VPSRLD   $20, BB1, BB1; VPXOR    CC3, BB1, BB1
    VPSLLD   $12, BB2, CC3; VPSRLD   $20, BB2, BB2; VPXOR    CC3, BB2, BB2
    VPSLLD   $12, BB3, CC3; VPSRLD   $20, BB3, BB3; VPXOR    CC3, BB3, BB3
    VMOVEDQA tmpStoreAVX2, CC3
polyAdd(0*8(out))
polyMulAVX2
    VPADD    BB0, AA0, AA0; VPADD    BB1, AA1, AA1; VPADD    BB2, AA2, AA2; VPADD    BB3, AA3, AA3
    VPXOR    AA0, DD0, DD0; VPXOR    AA1, DD1, DD1; VPXOR    AA2, DD2, DD2; VPXOR    AA3, DD3, DD3
    VPSHUFB ·rol8<>(SB), DD0, DD0; VPSHUFB ·rol8<>(SB), DD1, DD1; VPSHUFB ·rol8<>(SB), DD2, DD2; VPSHUFB ·rol8<>(SB),
    VPADD    DD0, CC0, CC0; VPADD    DD1, CC1, CC1; VPADD    DD2, CC2, CC2; VPADD    DD3, CC3, CC3
    VPXOR    CC0, BB0, BB0; VPXOR    CC1, BB1, BB1; VPXOR    CC2, BB2, BB2; VPXOR    CC3, BB3, BB3
    VMOVEDQA CC3, tmpStoreAVX2
    VPSLLD   $7, BB0, CC3; VPSRLD   $25, BB0, BB0; VPXOR    CC3, BB0, BB0
    VPSLLD   $7, BB1, CC3; VPSRLD   $25, BB1, BB1; VPXOR    CC3, BB1, BB1
    VPSLLD   $7, BB2, CC3; VPSRLD   $25, BB2, BB2; VPXOR    CC3, BB2, BB2
    VPSLLD   $7, BB3, CC3; VPSRLD   $25, BB3, BB3; VPXOR    CC3, BB3, BB3
```

internal/x/.../chacha20poly1305\_amd64.s lines 2593-2616 (go1.12)

Is this fine?



# crypto/elliptic: carry bug in x86-64 P-256 #20040

 **Closed**

agl opened this issue on Apr 19, 2017 · 12 comments

## 10 src/crypto/elliptic/p256\_asm\_amd64.s

[View file](#)

1314	ADCQ p256const0<>(SB), acc5	1314	ADCQ p256const0<>(SB), acc5
1315	ADCQ \$0, acc6	1315	ADCQ \$0, acc6
1316	ADCQ p256const1<>(SB), acc7	1316	ADCQ p256const1<>(SB), acc7
1317	- ADCQ \$0, mul0	1317	+ ANDQ \$1, mul0
1318		1318	
1319	- CMOVQE acc0, acc4	1319	+ CMOVQE acc0, acc4
1320	- CMOVQE acc1, acc5	1320	+ CMOVQE acc1, acc5
1321	- CMOVQE acc2, acc6	1321	+ CMOVQE acc2, acc6
1322	- CMOVQE acc3, acc7	1322	+ CMOVQE acc3, acc7
1323		1323	
1324	RET	1324	RET
1325	/* ----- */	1325	/* ----- */

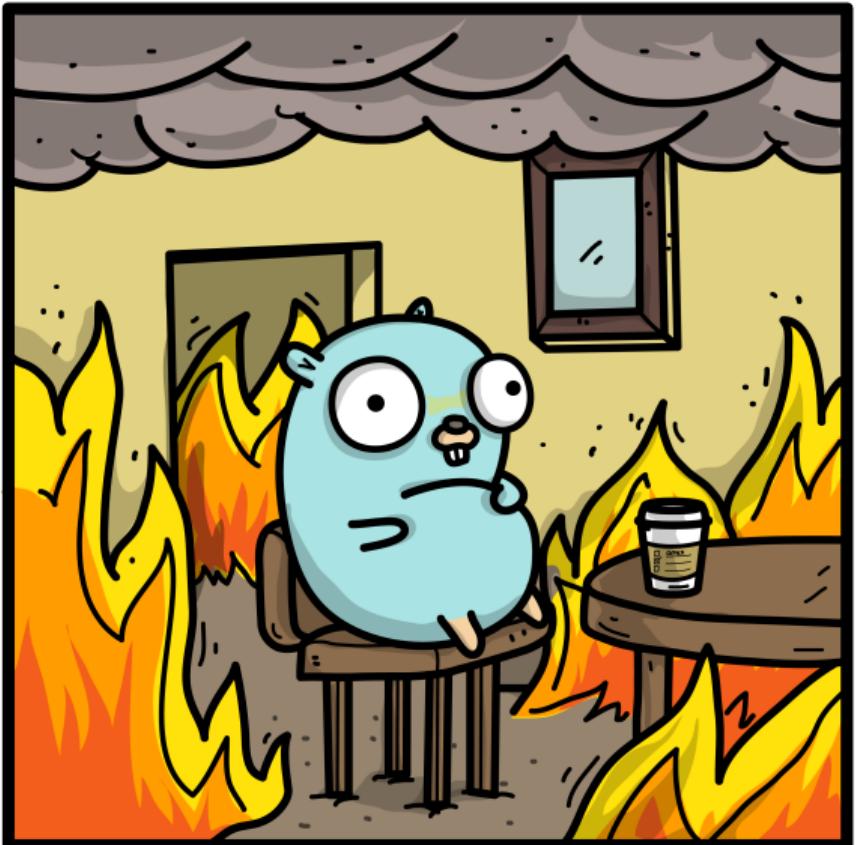
agl commented on May 23, 2017

Author

Contributor

...

(This issue is CVE-2017-8932.)



# Go Assembly Policy

---

*We prefer portable Go, not assembly. Code in assembly means ( $N$  packages \*  $M$  architectures) to maintain, rather than just  $N$  packages.*

Go Assembly Policy, Rule I

*Minimize use of assembly. We'd rather have a small amount of assembly for a 50% speedup rather than twice as much assembly for a 55% speedup. Explain the decision to place the assembly/Go boundary where it is in the commit message, and support it with benchmarks.*

Go Assembly Policy, Rule II

*Explain the root causes in code comments or commit messages.  
What changes in the compiler and standard library would allow  
you to replace this assembly with Go? (New intrinsics, SSA pattern  
matching, other optimizations.)*

Go Assembly Policy, Rule III

*Test it well. The bar for new assembly code is high; it needs commensurate test coverage. Existing high-level tests for Go implementations are often inadequate for hundreds of lines of assembly. Test subroutines individually. Fuzz the assembly implementation against the Go implementation.*

Go Assembly Policy, Rule V

*If possible, port existing reviewed implementations. A tool should make it easy to review diffs from decompiler output. Consider the license implications.*

Go Assembly Policy, Future Directions

*Make your assembly easy to review; ideally, auto-generate it using a simpler Go program. Comment it well.*

Go Assembly Policy, Rule IV

# Code Generation

---

There's a reason people use compilers.

## Intrinsics

```
__m256d latq = _mm256_loadu_pd(lat);
latq = _mm256_mul_pd(latq, _mm256_set1_pd(1 / 180.0));
latq = _mm256_add_pd(latq, _mm256_set1_pd(1.5));
__m256i lati = _mm256_srli_epi64(_mm256_castpd_si256(latq),  
  
__m256d lngq = _mm256_loadu_pd(lng);
lngq = _mm256_mul_pd(lngq, _mm256_set1_pd(1 / 360.0));
lngq = _mm256_add_pd(lngq, _mm256_set1_pd(1.5));
__m256i lngi = _mm256_srli_epi64(_mm256_castpd_si256(lngq),
```

## High-level Assembler

Assembly language plus **high-level language features**.

Macro assemblers: Microsoft Macro Assembler (MASM), Netwide Assembler (NASM), ...

## High-level Assembler

Assembly language plus high-level language features.

Macro assemblers: Microsoft Macro Assembler (MASM), Netwide Assembler (NASM), ...

# OpenSSL<sup>TM</sup>

Cryptography and SSL/TLS Toolkit



PeachPy



Python-based High-Level Assembler

What about Go?

# The **avo** Library

---



<https://github.com/mmcloughlin/avo>

Use Go control structures for assembly generation;  
**avo** programs are Go programs

**Register allocation:** write functions with virtual registers and avo assigns physical registers for you

Automatically load arguments and store return  
values: ensure memory offsets are correct for  
complex structures

Generation of stub files to interface with your Go package

```
import . "github.com/mmcloughlin/avo/build"

func main() {
    TEXT("Add", NOSPLIT, "func(x, y uint64) uint64")
    Doc("Add adds x and y.")
    x := Load(Param("x"), GP64())
    y := Load(Param("y"), GP64())
    ADDQ(x, y)
    Store(y, ReturnIndex(0))
    RET()
    Generate()
}
```





```
import . "github.com/mmcloughlin/avo/build"

func main() {
    TEXT("Add", NOSPLIT, "func(x, y uint64) uint64")
    Doc("Add adds x and y.")
    x := Load(Param("x"), GP64())           < Param references
    y := Load(Param("y"), GP64())           < Allocates register
    ADDQ(x, y)
    Store(y, ReturnIndex(0))
    RET()
    Generate()
}
```

```
import . "github.com/mmccloughlin/avo/build"

func main() {
    TEXT("Add", NOSPLIT, "func(x, y uint64) uint64")
    Doc("Add adds x and y.")
    x := Load(Param("x"), GP64())
    y := Load(Param("y"), GP64())
    ADDQ(x, y)                      < ADD can take virtual registers
    Store(y, ReturnIndex(0))
    RET()
    Generate()
}
```

```
import . "github.com/mmccloughlin/avo/build"

func main() {
    TEXT("Add", NOSPLIT, "func(x, y uint64) uint64")
    Doc("Add adds x and y.")
    x := Load(Param("x"), GP64())
    y := Load(Param("y"), GP64())
    ADDQ(x, y)
    Store(y, ReturnIndex(0))           < Store return value
    RET()
    Generate()
}
```

```
import . "github.com/mmcloughlin/avo/build"

func main() {
    TEXT("Add", NOSPLIT, "func(x, y uint64) uint64")
    Doc("Add adds x and y.")
    x := Load(Param("x"), GP64())
    y := Load(Param("y"), GP64())
    ADDQ(x, y)
    Store(y, ReturnIndex(0))
    RET()
    Generate()           < Generate compiles and outputs assembly
}
```

## Build

```
go run asm.go -out add.s -stubs stubs.go
```

## Generated Assembly

```
// Code generated by command: go run asm.go -out add.s -stubs stu

#include "textflag.h"

// func Add(x uint64, y uint64) uint64
TEXT ·Add(SB), NOSPLIT, $0-24
    MOVQ x+0(FP), AX
    MOVQ y+8(FP), CX
    ADDQ AX, CX
    MOVQ CX, ret+16(FP)
    RET
```

## Generated Assembly

```
// Code generated by command: go run asm.go -out add.s -stubs stu

#include "textflag.h"

// func Add(x uint64, y uint64) uint64
TEXT ·Add(SB), NOSPLIT, $0-24          ‹ Computed stack sizes
    MOVQ x+0(FP), AX
    MOVQ y+8(FP), CX
    ADDQ AX, CX
    MOVQ CX, ret+16(FP)
    RET
```

## Generated Assembly

```
// Code generated by command: go run asm.go -out add.s -stubs stu

#include "textflag.h"

// func Add(x uint64, y uint64) uint64
TEXT •Add(SB), NOSPLIT, $0-24
    MOVQ x+0(FP), AX                                < Computed offsets
    MOVQ y+8(FP), CX
    ADDQ AX, CX
    MOVQ CX, ret+16(FP)
    RET
```

## Generated Assembly

```
// Code generated by command: go run asm.go -out add.s -stubs stu

#include "textflag.h"

// func Add(x uint64, y uint64) uint64
TEXT •Add(SB), NOSPLIT, $0-24
    MOVQ x+0(FP), AX          < Registers allocated
    MOVQ y+8(FP), CX
    ADDQ AX, CX
    MOVQ CX, ret+16(FP)
    RET
```

## Auto-generated Stubs

```
// Code generated by command: go run asm.go -out add.s -stubs stubs

package addavo

// Add adds x and y.
func Add(x uint64, y uint64) uint64
```

## Go Control Structures

```
TEXT("Mul5", NOSPLIT, "func(x uint64) uint64")
Doc("Mul5 adds x to itself five times.")
x := Load(Param("x"), GP64())
p := GP64()
MOVQ(x, p)
for i := 0; i < 4; i++ {
    ADDQ(x, p)
}
Store(p, ReturnIndex(0))
RET()
```

## Go Control Structures

```
TEXT("Mul5", NOSPLIT, "func(x uint64) uint64")
Doc("Mul5 adds x to itself five times.")
x := Load(Param("x"), GP64())
p := GP64()
MOVQ(x, p)
for i := 0; i < 4; i++ {
    ADDQ(x, p)           < Generate four ADDQ instructions
}
Store(p, ReturnIndex(0))
RET()
```

## Generated Assembly

```
// func Mul5(x uint64) uint64
TEXT ·Mul5(SB), NOSPLIT, $0-16
    MOVQ x+0(FP), AX
    MOVQ AX, CX
    ADDQ AX, CX
    ADDQ AX, CX
    ADDQ AX, CX
    ADDQ AX, CX
    MOVQ CX, ret+8(FP)
    RET
```

## Generated Assembly

```
// func Mul5(x uint64) uint64
TEXT ·Mul5(SB), NOSPLIT, $0-16
    MOVQ x+0(FP), AX
    MOVQ AX, CX
    ADDQ AX, CX          < Look, there's four of them!
    ADDQ AX, CX
    ADDQ AX, CX
    ADDQ AX, CX
    MOVQ CX, ret+8(FP)
    RET
```

## Complex Parameter Loading

```
type Struct struct {
    A     byte
    B     uint32
    Sub  [7]complex64
    C     uint16
}
```

# Complex Parameter Loading

```
Package("github.com/mmcloughlin/params")
TEXT("Sub5Imag", NOSPLIT, "func(s Struct) float32")
Doc("Returns the imaginary part of s.Sub[5]")
x := Load(Param("s")).Field("Sub").Index(5).Imag(), XMM()
Store(x, ReturnIndex(0))
RET()
```

# Complex Parameter Loading

```
Package("github.com/mmcloughlin/params")           < Types
TEXT("Sub5Imag", NOSPLIT, "func(s Struct) float32")
Doc("Returns the imaginary part of s.Sub[5]")
x := Load(Param("s")).Field("Sub").Index(5).Imag(), XMM()
Store(x, ReturnIndex(0))
RET()
```

# Complex Parameter Loading

```
Package("github.com/mmcloughlin/params")
TEXT("Sub5Imag", NOSPLIT, "func(s Struct) float32")
Doc("Returns the imaginary part of s.Sub[5]")
x := Load(Param("s")).Field("Sub").Index(5).Imag(), XMM()
Store(x, ReturnIndex(0))
RET()
```

## Generated Assembly

```
// func Sub5IImg(s Struct) float32
TEXT ·Sub5IImg(SB), NOSPLIT, $0-76
    MOVSS s_Sub_5_imag+52(FP), X0
    MOVSS X0, ret+72(FP)
    RET
```

## Generated Assembly

```
// func Sub5IImg(s Struct) float32
TEXT ·Sub5IImg(SB), NOSPLIT, $0-76
    MOVSS s_Sub_5_imag+52(FP), X0 // Of course it was 52 bytes
    MOVSS X0, ret+72(FP)
    RET
```

## SHA-1

Cryptographic hash function.

- 80 rounds
- Constants and bitwise functions vary
- Message update rule
- State update rule

`avo` can be used to create a completely *unrolled* implementation.

## SHA-1 Subroutines

```
func majority(b, c, d Register) Register {
    t, r := GP32(), GP32()
    MOVL(b, t)
    ORL(c, t)
    ANDL(d, t)
    MOVL(b, r)
    ANDL(c, r)
    ORL(t, r)
    return r
}
```

## SHA-1 Subroutines

```
func xor(b, c, d Register) Register {
    r := GP32()
    MOVL(b, r)
    XORL(c, r)
    XORL(d, r)
    return r
}
```

## SHA-1 Loops

```
Comment("Load initial hash.")
hash := [5]Register{GP32(), GP32(), GP32(), GP32(), GP32()}
for i, r := range hash {
    MOVL(h.Offset(4*i), r)
}

Comment("Initialize registers.")
a, b, c, d, e := GP32(), GP32(), GP32(), GP32(), GP32()
for i, r := range []Register{a, b, c, d, e} {
    MOVL(hash[i], r)
}
```



```
for r := 0; r < 80; r++ {                                < Loop over rounds
    Commentf("Round %d.", r)
    ...
    q := quarter[r/20]
    t := GP32()
    MOVL(a, t)
    ROLL(U8(5), t)
    ADDL(q.F(b, c, d), t)
    ADDL(e, t)
    ADDL(U32(q.K), t)
    ADDL(u, t)
    ROLL(Imm(30), b)
    a, b, c, d, e = t, a, b, c, d
}
```

```
for r := 0; r < 80; r++ {  
    Commentf("Round %d.", r)  
    ...  
    q := quarter[r/20]  
    t := GP32()                                < State update  
    MOVL(a, t)  
    ROLL(U8(5), t)  
    ADDL(q.F(b, c, d), t)  
    ADDL(e, t)  
    ADDL(U32(q.K), t)  
    ADDL(u, t)  
    ROLL(Imm(30), b)  
    a, b, c, d, e = t, a, b, c, d  
}
```

## SHA-1 Conditionals

```
u := GP32()
if r < 16 {
    MOVL(m.Offset(4*r), u)
    BSWAPL(u)
} else {
    MOVL(w(r-3), u)
    XORL(w(r-8), u)
    XORL(w(r-14), u)
    XORL(w(r-16), u)
    ROLL(U8(1), u)
}
```

## SHA-1 Conditionals

```
u := GP32()
if r < 16 {                                ‹ Early rounds
    MOVL(m.Offset(4*r), u)                  ‹ Read from memory
    BSWAPL(u)
} else {
    MOVL(w(r-3), u)
    XORL(w(r-8), u)
    XORL(w(r-14), u)
    XORL(w(r-16), u)
    ROLL(U8(1), u)
}
```

## SHA-1 Conditionals

```
u := GP32()
if r < 16 {
    MOVL(m.Offset(4*r), u)
    BSWAPL(u)
} else {
    MOVL(w(r-3), u)           < Formula in later rounds
    XORL(w(r-8), u)
    XORL(w(r-14), u)
    XORL(w(r-16), u)
    ROLL(U8(1), u)
}
```

116 **avo** lines to 1507 assembly lines

# Real **avo** Examples

With the help of Damian Gryski.

- SHA-1
- FNV-1a
- Vector Dot Product
- Marvin32
- Sip13
- SPECK
- Bloom Index
- Chaskey MAC
- Farmhash32/64



Thank You

<https://github.com/mmcloughlin/avo>  
@mbmcloughlin