Better **x86** Assembly Generation with Go

Michael McLoughlin

dotGo 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

Go Proverbs which Might Have Been

~~Cgo~~ 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

~~Cgo~~ Assembly is not Go.

My Inner Rob Pike

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

Made-up Go Proverb


Donald
Knuth '60
50th Reunion

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

Go Assembly Primer

Hello, World!

```
package add

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

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)
    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	`crypto`
8069	`runtime`
5686	`internal`
1173	`math`
1005	`syscall`
574	`cmd`
279	`hash`
36	`reflect`

Table 1: Assembly Lines by Top-Level Packages

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

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/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

```
openAVX2InternalLoop:
    // Lets just say this spaghetti loop interleaves 2 quarter rounds with 3 poly multiplications
    // Effectively per 512 bytes of stream we hash 480 bytes of ciphertext
    polyAdd(0*8(inp)(itr1*1))
    VPADDD   BB0, AA0, AA0; VPADDD BB1, AA1, AA1; VPADDD BB2, AA2, AA2; VPADDD BB3, AA3, AA3
    polyMulStage1_AVX2
    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<>(S
    polyMulStage2_AVX2
    VPADDD   DD0, CC0, CC0; VPADDD DD1, CC1, CC1; VPADDD DD2, CC2, CC2; VPADDD DD3, CC3, CC3
    VPXOR    CC0, BB0, BB0; VPXOR CC1, BB1, BB1; VPXOR CC2, BB2, BB2; VPXOR CC3, BB3, BB3
    polyMulStage3_AVX2
    VMOVDQA  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
    VMOVDQA  tmpStoreAVX2, CC3
    polyMulReduceStage
    VPADDD   BB0, AA0, AA0; VPADDD BB1, AA1, AA1; VPADDD BB2, AA2, AA2; VPADDD 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),
    polyAdd(2*8(inp)(itr1*1))
    VPADDD   DD0, CC0, CC0; VPADDD DD1, CC1, CC1; VPADDD DD2, CC2, CC2; VPADDD DD3, CC3, CC3
```

`internal/x/.../chacha20poly1305_amd64.s` lines 856-879 (`go1.12`)

```
// 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; VPADDD ·avx2IncMask<>(SB), DD0, DD1
    VMOVDQA AA0, AA2; VMOVDQA BB0, BB2; VMOVDQA CC0, CC2; VPADDD ·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, T
    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, T
    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
    VPADDD  TT0, AA0, AA0; VPADDD TT0, AA1, AA1; VPADDD TT0, AA2, AA2
    VPADDD  TT1, BB0, BB0; VPADDD TT1, BB1, BB1; VPADDD TT1, BB2, BB2
    VPADDD  TT2, CC0, CC0; VPADDD TT2, CC1, CC1; VPADDD TT2, CC2, CC2
```

`internal/x/.../chacha20poly1305_amd64.s` lines 1072-1095 (`go1.12`)

```
openAVX2Tail512LoopA:
    VPADDD   BB0, AA0, AA0; VPADDD BB1, AA1, AA1; VPADDD BB2, AA2, AA2; VPADDD 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<>(S
    VPADDD   DD0, CC0, CC0; VPADDD DD1, CC1, CC1; VPADDD DD2, CC2, CC2; VPADDD DD3, CC3, CC3
    VPXOR    CC0, BB0, BB0; VPXOR CC1, BB1, BB1; VPXOR CC2, BB2, BB2; VPXOR CC3, BB3, BB3
    VMOVDQA  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
    VMOVDQA  tmpStoreAVX2, CC3
    polyAdd(0*8(itr2))
    polyMulAVX2
    VPADDD   BB0, AA0, AA0; VPADDD BB1, AA1, AA1; VPADDD BB2, AA2, AA2; VPADDD 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),
    VPADDD   DD0, CC0, CC0; VPADDD DD1, CC1, CC1; VPADDD DD2, CC2, CC2; VPADDD DD3, CC3, CC3
    VPXOR    CC0, BB0, BB0; VPXOR CC1, BB1, BB1; VPXOR CC2, BB2, BB2; VPXOR CC3, BB3, BB3
    VMOVDQA  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:
    VPADDD   BB0, AA0, AA0; VPADDD BB1, AA1, AA1; VPADDD BB2, AA2, AA2; VPADDD 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<>(S
    VPADDD   DD0, CC0, CC0; VPADDD DD1, CC1, CC1; VPADDD DD2, CC2, CC2; VPADDD DD3, CC3, CC3
    VPXOR    CC0, BB0, BB0; VPXOR CC1, BB1, BB1; VPXOR CC2, BB2, BB2; VPXOR CC3, BB3, BB3
    VMOVDQA  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
    VMOVDQA  tmpStoreAVX2, CC3
    polyAdd(0*8(oup))
    polyMulAVX2
    VPADDD   BB0, AA0, AA0; VPADDD BB1, AA1, AA1; VPADDD BB2, AA2, AA2; VPADDD 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),
    VPADDD   DD0, CC0, CC0; VPADDD DD1, CC1, CC1; VPADDD DD2, CC2, CC2; VPADDD DD3, CC3, CC3
    VPXOR    CC0, BB0, BB0; VPXOR CC1, BB1, BB1; VPXOR CC2, BB2, BB2; VPXOR CC3, BB3, BB3
    VMOVDQA  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`)

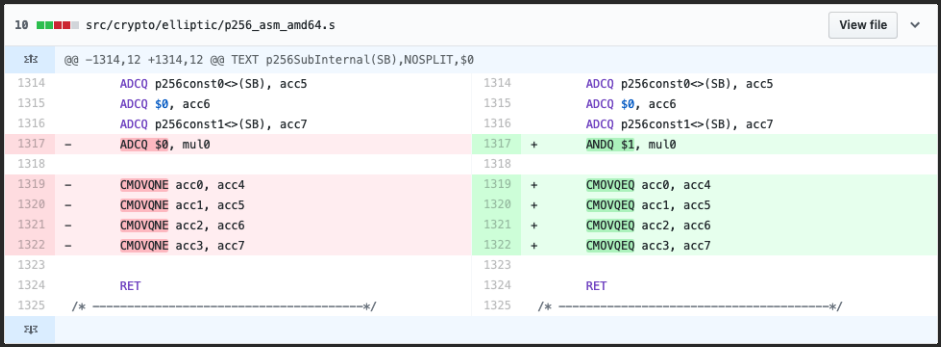
Is this fine?

```
TEXT p256SubInternal(SB),NOSPLIT,$0
    XORQ mul0, mul0
    SUBQ t0, acc4
    SBBQ t1, acc5
    SBBQ t2, acc6
    SBBQ t3, acc7
    SBBQ $0, mul0

    MOVQ acc4, acc0
    MOVQ acc5, acc1
    MOVQ acc6, acc2
    MOVQ acc7, acc3

    ADDQ $-1, acc4
    ADCQ p256const0<>(SB), acc5
    ADCQ $0, acc6
    ADCQ p256const1<>(SB), acc7
    ADCQ $0, mul0

    CMOVQNE acc0, acc4
    CMOVQNE acc1, acc5
    CMOVQNE acc2, acc6
    CMOVQNE acc3, acc7

    RET
```

crypto/elliptic/p256_asm_amd64.s lines 1300-1324 (94e44a9c8e)

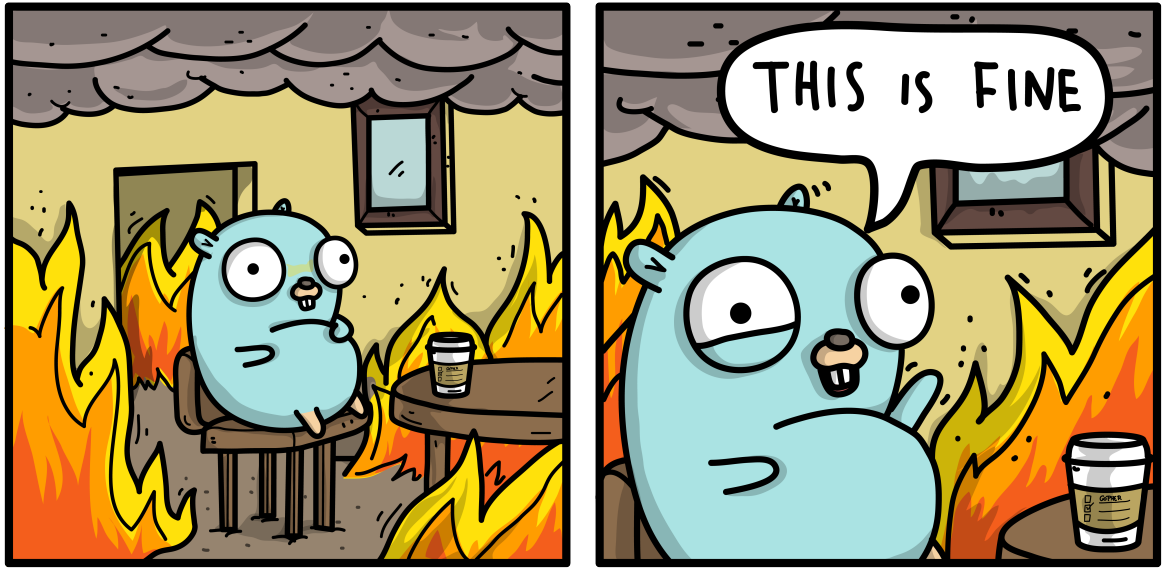
crypto/elliptic: carry bug in x86-64 P-256 #20040

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


10 src/crypto/elliptic/p256_asm_amd64.s
View file
@@ -1314,12 +1314,12 @@ TEXT p256SubInternal(SB),NOSPLIT,$0
1314 ADCQ p256const0<>(SB), acc5
1315 ADCQ $0, acc6
1316 ADCQ p256const1<>(SB), acc7
1317 - ADCQ $0, mul0
1318
1319 - CMOVQNE acc0, acc4
1320 - CMOVQNE acc1, acc5
1321 - CMOVQNE acc2, acc6
1322 - CMOVQNE acc3, acc7
1323
1324 RET
1325 /* ---------------------------------------*/
1314 ADCQ p256const0<>(SB), acc5
1315 ADCQ $0, acc6
1316 ADCQ p256const1<>(SB), acc7
1317 + ANDQ $1, mul0
1318
1319 + CMOVQEQ acc0, acc4
1320 + CMOVQEQ acc1, acc5
1321 + CMOVQEQ acc2, acc6
1322 + CMOVQEQ acc3, acc7
1323
1324 RET
1325 /* ---------------------------------------*/

agl commented on May 23, 2017 Author Contributor ...

(This issue is CVE-2017-8932.)


THIS IS FINE
GOPHER

Go Assembly Policy

1. Prefer Go, not assembly
2. Minimize use of assembly
3. Explain root causes
4. Test it well
5. Make assembly easy to review

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.
latq = _mm256_add_pd(latq, _mm256_set1_pd(1.5));
__m256i lati = _mm256_srli_epi64(_mm256_castpd_si2

__m256d lngq = _mm256_loadu_pd(lng);
lngq = _mm256_mul_pd(lngq, _mm256_set1_pd(1 / 360.
lngq = _mm256_add_pd(lngq, _mm256_set1_pd(1.5));
__m256i lngi = _mm256_srli_epi64(_mm256_castpd_si2
```

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™
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())
    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())
    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/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)          ‹ ADD can take virtual registers
    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())
    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 -stub

#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 -stub

#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 -stub

#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 -stub

#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 -stub

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
```

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(), GP3
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++ {
    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++ {          ‹ 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
- Farmhash64

Thank You

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