avo — "Generate x86 Assembly with Go"

Wojciech Muła, 0x80.pl April 2022

Project avo highlights:

- A Go program generates assembly files
- The generator takes care of register allocation
- ... but we can use explicit hardware registers (for instance `CX` for shift amounts)
- Takes care of proper accesing to structures
- ... with hand-written assembly we need `go vet -asmdecl`
- Can generate Go stub files

Major modules — 1/3

- `avo/build` — all instructions, stack allocation, program comments, virtual registers:
- ... `GP64()` — 64-bit GPR
- ... `GP64().As32()` — the lower 32 bits of GPR
- ... `GP64().As16()` — the lower 16 bits of GPR
- ... `GP64().As8()` — the lower 8 bits of GPR
- ... `GP64().As8H()` — the higher 8 bits of GPR (like `AH`, not always available)
- ... `XMM()` — SSE register
- ... `YMM()` — AVX register
- ... `ZMM()` — ZMM register
- ... `x := AllocLocal(8)`

Major modules — 2/3

- `avo/operand` — labels, immediate types, memory address
- `... Label("name")` — declare label
- `... JMP(LabelRef("name"))` — jump to label
- `... ADDQ(U8(42), reg)` — U8, U16, U32
- `... Mem{Base: reg, Index: reg, Scale: imm, Disp: imm}` — description of x86 address

Major modules — 3/3

- `avo/buildtags` — construct build tags, like `go:build !appengine && !noasm && gc && !noasm`
- `avo/reg` — types for registers, names of physical registers, etc.

Example 1 — explicit registers

```
1  package main
2
3  import (
4        . "github.com/mmcloughlin/avo/build"
5        . "github.com/mmcloughlin/avo/operand"
6        "github.com/mmcloughlin/avo/buildtags"
7        "github.com/mmcloughlin/avo/reg"
8  )
9
10 func main() {
11       generateSub()
12       Generate()
13 }
14
15 func generateSub() {
16       TEXT("Sub", NOSPLIT, "func(x, y uint64) uint64")
17       Doc("Sub subtracts x and y.")
18       x := reg.R11
19       y := reg.R12
20
21       Load(Param("x"), x)
22       Load(Param("y"), y)
23       SUBQ(x, y)
24       Store(y, ReturnIndex(0))
25       RET()
26 }
```

Example 1 — output

```
1 #include "textflag.h"
2
3 // func Sub(x uint64, y uint64) uint64
4 TEXT Sub(SB), NOSPLIT, $0-24
5         MOVQ x+0(FP), R11
6         MOVQ y+8(FP), R12
7         SUBQ R11, R12
8         MOVQ R12, ret+16(FP)
9         RET
```

Example 2 — implicit registers

```
1  package main
2
3  import (
4      . "github.com/mmcloughlin/avo/build"
5      . "github.com/mmcloughlin/avo/operand"
6      "github.com/mmcloughlin/avo/buildtags"
7      "github.com/mmcloughlin/avo/reg"
8  )
9
10 func main() {
11     generateAdd()
12     Generate()
13 }
14
15 func generateAdd() {
16     TEXT("Add", NOSPLIT, "func(x, y uint64) uint64")
17     Doc("Add adds x and y.")
18     x := Load(Param("x"), GP64())
19     y := Load(Param("y"), GP64())
20     ADDQ(x, y)
21     Store(y, ReturnIndex(0))
22     RET()
23 }
```

Example 2 — output

```
1 #include "textflag.h"
2
3 // func Add(x uint64, y uint64) uint64
4 TEXT Add(SB), NOSPLIT, $0-24
5         MOVQ x+0(FP), AX
6         MOVQ y+8(FP), CX
7         ADDQ AX, CX
8         MOVQ CX, ret+16(FP)
9         RET
```

Example 3 — accessing structure

```
1  package main
2
3  import (
4      . "github.com/mmcloughlin/avo/build"
5      . "github.com/mmcloughlin/avo/operand"
6      "github.com/mmcloughlin/avo/buildtags"
7      "github.com/mmcloughlin/avo/reg"
8  )
9
10 type Structure struct {
11     bytes []byte
12     value uint64
13 }
14
15 func main() {
16     generateStruct()
17     Generate()
18 }
19
20 func generateStruct() {
21     Package("main")
22     TEXT("CapPlusLen", NOSPLIT, "func(s* Structure)")
23
24     s := Dereference(Param("s"))
25
26     Comment("s.value = len(s.bytes) + cap(s.bytes)")
27     length := GP64()
28     Load(s.Field("bytes").Len(), length)
29     capacity := GP64()
30     Load(s.Field("bytes").Cap(), capacity)
31
32     tmp := GP64()
33     LEAQ(Mem{Base: length, Index: capacity, Scale: 1}, tmp)
34
35     Store(tmp, s.Field("value"))
36 }
```

Example 3 — output

```
1  #include "textflag.h"
2
3  // func CapPlusLen(s *Structure)
4  TEXT CapPlusLen(SB), NOSPLIT, $0-8
5          MOVQ s+0(FP), AX
6
7          // s.value = len(s.bytes) + cap(s.bytes)
8          MOVQ 8(AX), CX
9          MOVQ 16(AX), DX
10         LEAQ (CX)(DX*1), CX
11         MOVQ CX, 24(AX)
```