

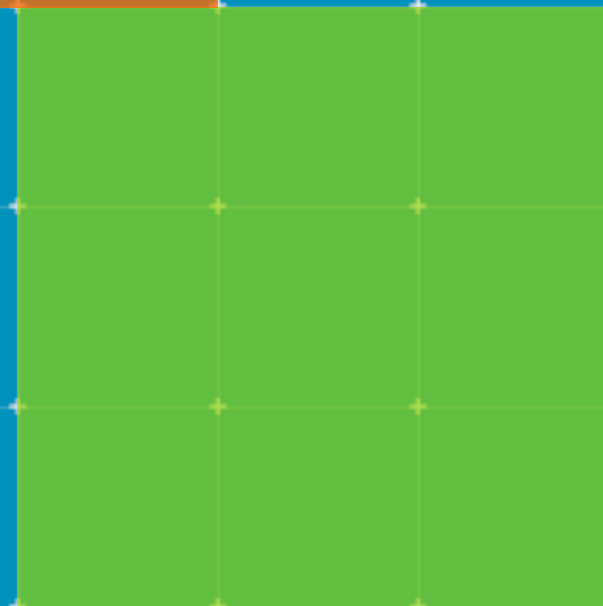


arm

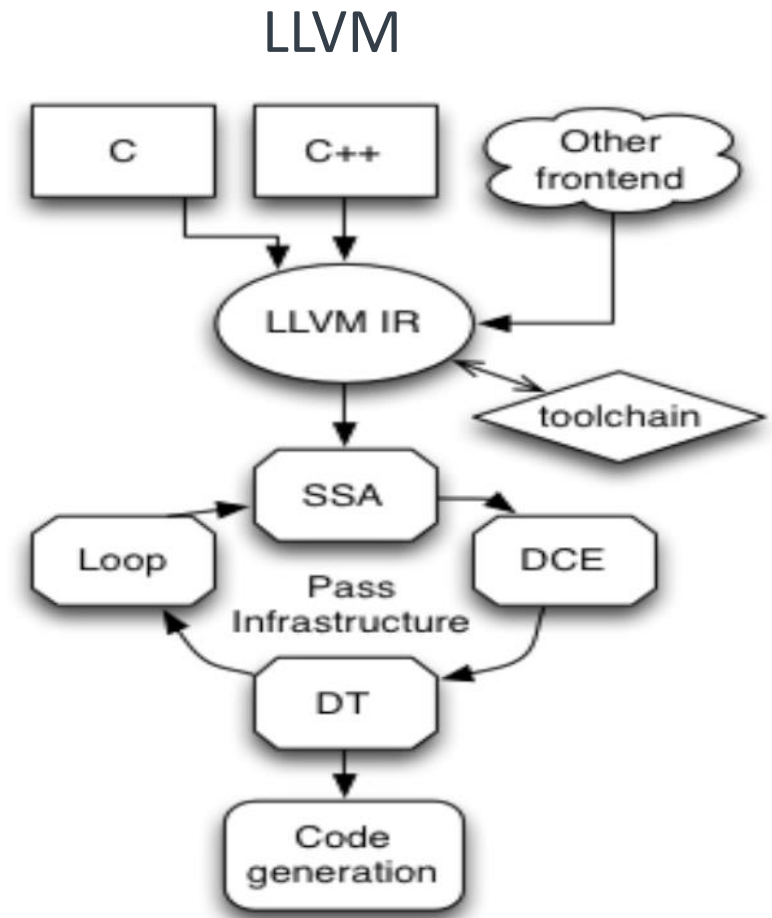
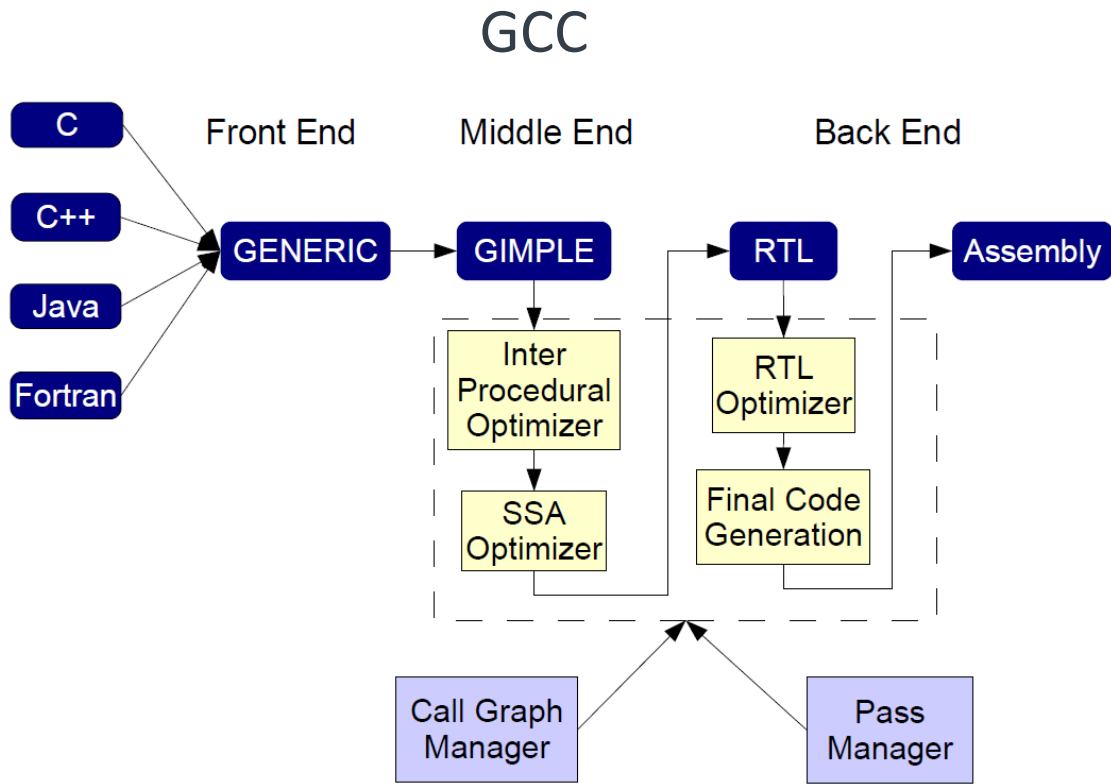
Golang Compiler Internals for arm64

Wei Xiao (wei.xiao@arm.com)
<https://github.com/williamweixiao>
December 20, 2017

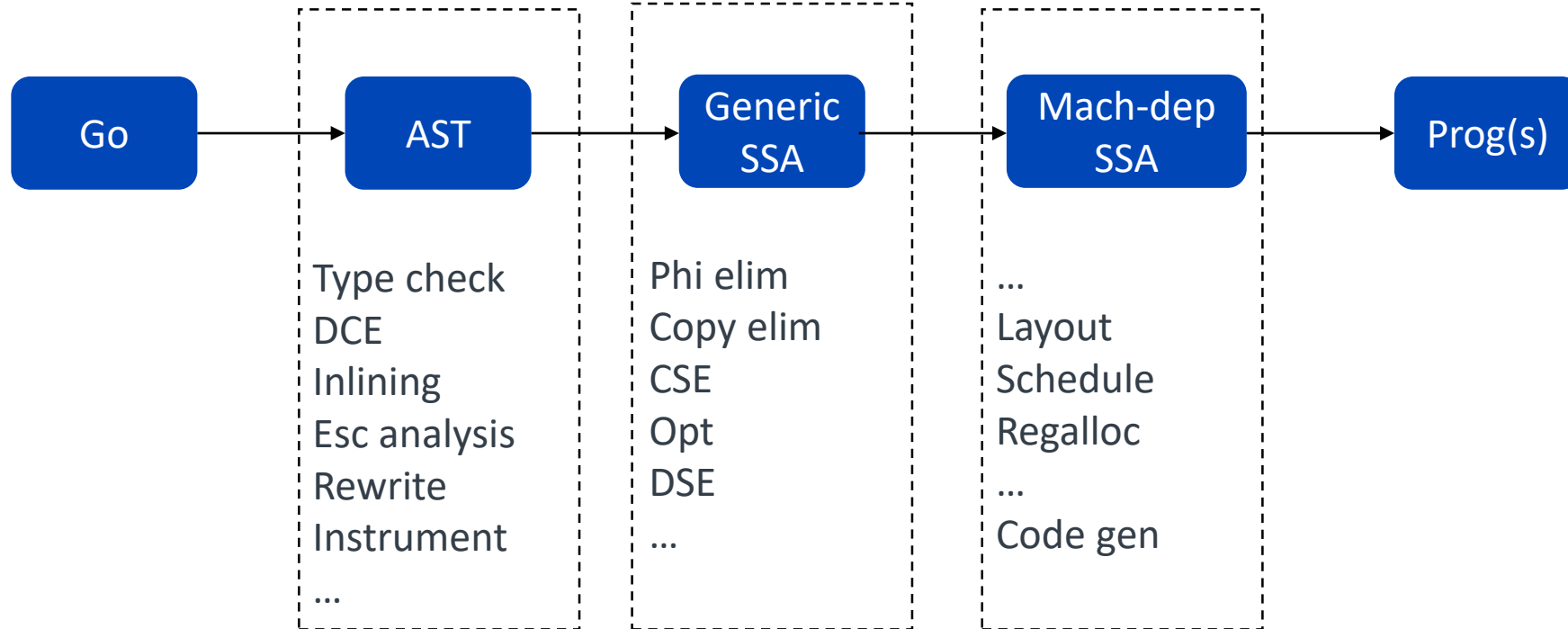
Overview



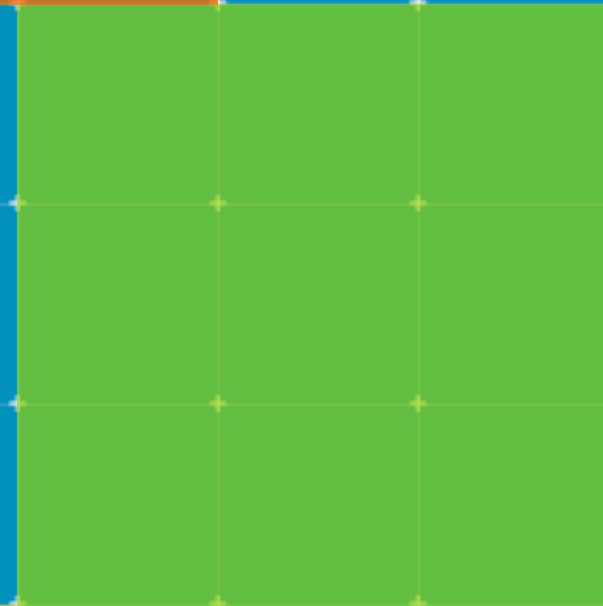
Classic Compiler Overview



GoLang Overview



Front End



Lexer

The whole lexer is just 765 LOC

Hand-written lexer made up of a big switch case

Doc: <https://golang.org/pkg/go/scanner/>

go/scanner/scanner.go

- func (s *Scanner) Init(file *token.File, src []byte, err ErrorHandler, mode Mode)
- func (s *Scanner) Scan() (pos token.Pos, tok token.Token, lit string)

```
602 func (s *Scanner) Scan() (pos token.Pos, tok token.Token, lit string) {
603 scanAgain:
604     s.skipwhitespace()
605
606     // current token start
607     pos = s.file.Pos(s.offset)
608
609     // determine token value
610     insertSemi := false
611     switch ch := s.ch; {
612     case isLetter(ch):
613         lit = s.scanIdentifier()
614         if len(lit) > 1 {
615             // keywords are longer than one letter - avoid lookup otherwise
616             tok = token.Lookup(lit)
617             switch tok {
618             case token.IDENT, token.BREAK, token.CONTINUE, token.FALLTHROUGH, token.RETURN:
619                 insertSemi = true
620             }
621         } else {
622             insertSemi = true
623             tok = token.IDENT
624         }
625     case '0' <= ch && ch <= '9':
626         insertSemi = true
627         tok, lit = s.scanNumber(false)
628     default:
629         s.next() // always make progress
630         switch ch {
631         case -1:
632             if s.insertSemi {
633                 s.insertSemi = false // EOF consumed
634                 return pos, token.SEMICOLON, "\n"
```

Parser

Hand-written LL(1) parser with 2521 LOC

Output is an abstract syntax tree (AST) representing the Go source (<https://golang.org/pkg/go/ast>)

Doc: <https://golang.org/pkg/go/parser>

`go/parser/interface.go`

- `func ParseFile(fset *token.FileSet, filename string, src interface{}, mode Mode) (f *ast.File, err error)`
- `func ParseExpr(x string) (ast.Expr, error)`

Go internal fast syntax tree (DAG) parser (`cmd/compile/internal/syntax`)

Syntax Tree Node

```
21 type Node struct {
22     // Tree structure.
23     // Generic recursive walks should follow these fields.
24     Left *Node
25     Right *Node
26     Ninit Nodes
27     Nbody Nodes
28     List Nodes
29     Rlist Nodes
30
31     // most nodes
32     Type *types.Type
33     Orig *Node // original form, for printing, and tracking copies of ONAMES
34
35     // func
36     Func *Func
37
38     // ONAME, OTYPE, OPACK, OLABEL, some OLITERAL
39     Name *Name
40
41     Sym *types.Sym // various
42     E interface{} // Opt or Val, see methods below
43
44     // Various. Usually an offset into a struct. For example:
45     // - ONAME nodes that refer to local variables use it to identify their stack frame position.
46     // - ODOT, ODOTPTR, and OINDREGSP use it to indicate offset relative to their base address.
47     // - OSTRUCTKEY uses it to store the named field's offset.
48     // - Named OLITERALS use it to store their ambient iota value.
49     // Possibly still more uses. If you find any, document them.
50     Xoffset int64
51
52     Pos src.XPos
53
54     flags bitset32
55
56     Esc uint16 // EscXXX
57
58     Op Op
59     Etype types.EType // op for OASOP, etype for OTYPE, exclam for export, 6g saved reg, ChanDir for OTCHAN, for OINDEXMAP 1=LHS,0=RHS
60 }
```

```
// names
ONAME // var, const or func name
ONONAME // unnamed arg or return value: f(int, string) (int, error) { etc }
OTYPE // type name
OPACK // import
OLITERAL // literal

// expressions
OADD // Left + Right
OSUB // Left - Right
OOR // Left | Right
OXOR // Left ^ Right
OADDSTR // +{List} (string addition, list elements are strings)
OADDR // &Left
OANDAND // Left && Right
OAPPEND // append(List); after walk, Left may contain elem type descriptor
OARRAYBYESTR // Type(Left) (Type is string, Left is a []byte)
OARRAYBYESTRTMP // Type(Left) (Type is string, Left is a []byte, ephemeral)
OARRAYRUNESTR // Type(Left) (Type is string, Left is a []rune)
OSTRARRAYBYTE // Type(Left) (Type is []byte, Left is a string)
OSTRARRAYBYETMP // Type(Left) (Type is []byte, Left is a string, ephemeral)
OSTRARRAYRUNE // Type(Left) (Type is []rune, Left is a string)
OAS // Left = Right or (if Colas=true) Left := Right
OAS2 // List = Rlist (x, y, z = a, b, c)
OAS2FUNC // List = Rlist (x, y = f())
OAS2RECV // List = Rlist (x, ok = <-c)
OAS2MAPR // List = Rlist (x, ok = m["foo"])
OAS2DOTTYPE // List = Rlist (x, ok = I.(int))
OASOP // Left Etype= Right (x += y)
OCALL // Left(List) (function call, method call or type conversion)
OCALLFUNC // Left(List) (function call f(args))
OCALLMETH // Left(List) (direct method call x.Method(args))
OCALLINTER // Left(List) (interface method call x.Method(args))
OCALLPART // Left.Right (method expression x.Method, not called)
OCAP // cap(Left)
OCLOSE // close(Left)
OCLOSURE // func Type { Body } (func literal)
OCMPIFACE // Left Etype Right (interface comparison, x == y or x != y)
OCMPSTR // Left Etype Right (string comparison, x == y, x < y, etc)
OAMPLIT // Right{List} (composite literal, not yet lowered to specific form)
OMAPLIT // Type{List} (composite literal, Type is map)
OSTRUCTLIT // Type{List} (composite literal, Type is struct)
OARRAYLIT // Type{List} (composite literal, Type is array)
```


Syntax Tree Example

```
1 package test
2
3 func fact(n int) int {
4     if n == 0 {
5         return 1
6     }
7     return n * fact(n-1)
8 }
```

Dump via compile flag: “-W”

\$ go tool compile -w example.go

```
1 . IF l(4) tc(1)
2 . . EQ l(4) tc(1) bool
3 . . . NAME-test.n a(true) g(2) l(3) x(0) class(PPARAM) f(1) tc(1) used int
4 . . . LITERAL-0 l(4) tc(1) int
5 . IF-body
6 . . RETURN l(5) tc(1)
7 . . RETURN-list
8 . . . LITERAL-1 l(5) tc(1) int
9
10 . AS l(7) tc(1)
11 . . NAME-test..autotmp_2 a(true) l(7) x(0) class(PAUTO) esc(N) tc(1) assigned used int
12 . . CALLFUNC l(7) tc(1) int
13 . . . NAME-test.fact a(true) l(3) x(0) class(PFUNC) tc(1) used FUNC-func(int) int
14 . . CALLFUNC-list
15 . . . SUB l(7) tc(1) int
16 . . . . NAME-test.n a(true) g(2) l(3) x(0) class(PPARAM) f(1) tc(1) used int
17 . . . . LITERAL-1 l(7) tc(1) int
18
19 . RETURN l(7) tc(1)
20 . RETURN-list
21 . . MUL l(7) tc(1) int
22 . . . NAME-test.n a(true) g(2) l(3) x(0) class(PPARAM) f(1) tc(1) used int
23 . . . NAME-test..autotmp_2 a(true) l(7) x(0) class(PAUTO) esc(N) tc(1) assigned used int
24
25 . VARKILL tc(1)
26 . . NAME-test..autotmp_2 a(true) l(7) x(0) class(PAUTO) esc(N) tc(1) assigned used int
```

Front End

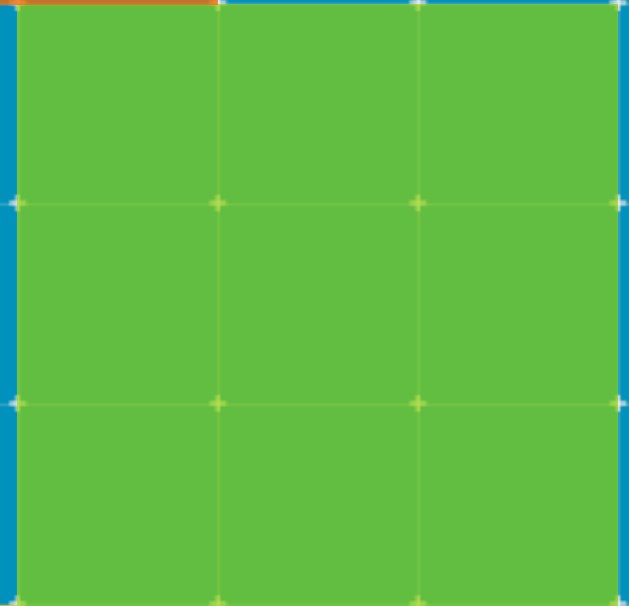
Works done by Golang (1.9) Front End:

- Syntax check
- Type check:
 - calculates expression types (e.g mismatched types)
 - evaluates compile time constants (e.g array bound must be non-negative)
 - rewrites n.Op to be more specific in some cases (e.g OADD → OADDSTR)
 - whether function terminates
- Dead code elimination
- Inlining: 80-nodes leaf functions (default)
- Escape analysis
- Declared and not used check
- Rewrite: e.g copy(a, b) → memmove, expand append(l1, l2...)
- Instrument: race & msan

There is no middle end for old versions (< 1.7) and lots of optimizations are done with AST.

Optimization example: <https://go-review.googlesource.com/c/go/+22292>

Middle End



SSA Form

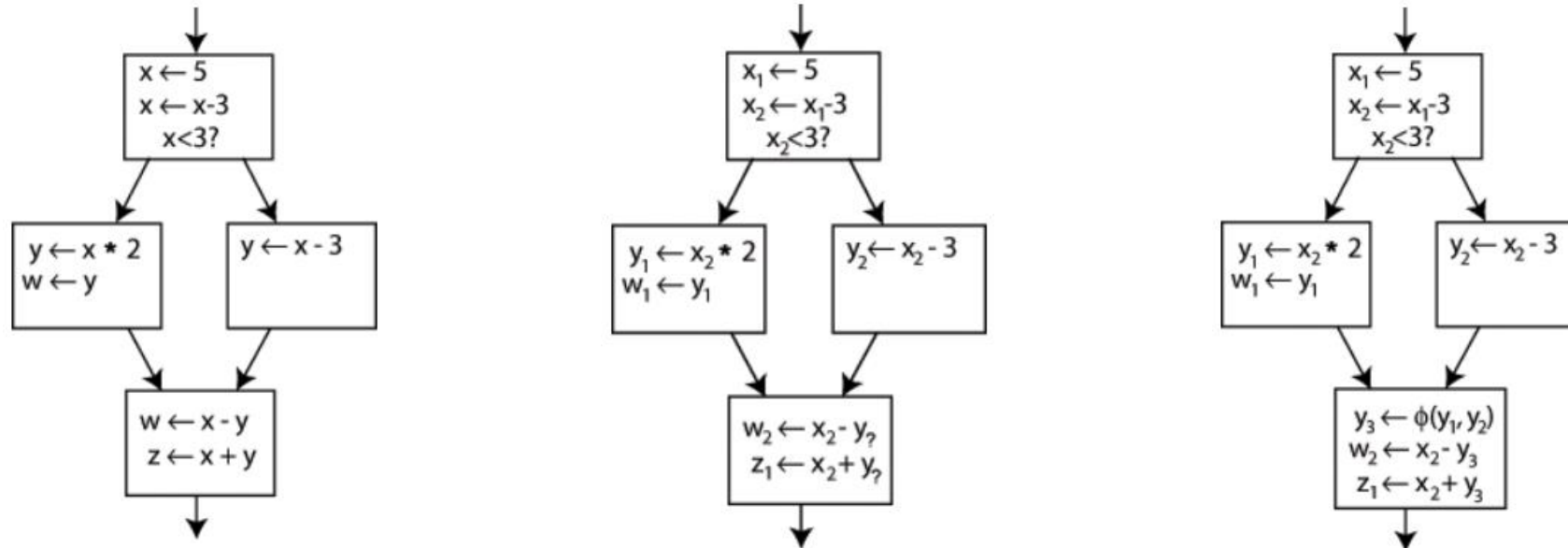
Static Single Assignment form (often abbreviated as **SSA form** or simply **SSA**) is a property of an [intermediate representation](#) (IR), which requires that each variable is assigned exactly once, and every variable is defined before it is used. Existing variables in the original IR are split into *versions*, new variables typically indicated by the original name with a subscript in textbooks, so that every definition gets its own version. In SSA form, [use-def chains](#) are explicit and each contains a single element.

The primary usefulness of SSA comes from how it simultaneously simplifies and improves the results of a variety of [compiler optimizations](#), by simplifying the properties of variables.

```
y1 := 1  
y2 := 2  
x1 := y2
```

Convert to SSA

Converting ordinary code into SSA form is primarily a simple matter of replacing the target of each assignment with a new variable, and replacing each use of a variable with the "version" of the variable [reaching](#) that point.



[Efficiently computing static single assignment form and the control dependence graph](#)

[cmd/compile/internal/gc/ssa.go](#)

```
func buildssa(fn *Node, worker int) *ssa.Func
```

SSA Value

cmd/compile/internal/ssa/value.go

```
19 type Value struct {
20     // A unique identifier for the value. For performance we allocate these IDs
21     // densely starting at 1. There is no guarantee that there won't be occasional holes, though.
22     ID ID
23
24     // The operation that computes this value. See op.go.
25     Op Op
26
27     // The type of this value. Normally this will be a Go type, but there
28     // are a few other pseudo-types, see type.go.
29     Type *types.Type
30
31     // Auxiliary info for this value. The type of this information depends on the opcode and type.
32     // AuxInt is used for integer values, Aux is used for other values.
33     // Floats are stored in AuxInt using math.Float64bits(f).
34     AuxInt int64
35     Aux    interface{}
36
37     // Arguments of this value
38     Args []*Value
39
40     // Containing basic block
41     Block *Block
```

Generic SSA Ops

cmd/compile/internal/ssa/gen/genericOps.go

```
var genericOps = []opData{  
  
    // 2-input arithmetic  
    // Types must be consistent with Go typing. Add, for example, must take two values  
    // of the same type and produces that same type.  
    {name: "Add8", argLength: 2, commutative: true}, // arg0 + arg1  
    {name: "StaticCall", argLength: 1, aux: "SymOff", call: true, symEffect: "None"}, // call function aux.(*obj.LSym), arg0=memory.    auxint=arg size. Returns memory.  
  
    {name: "Avg32u", argLength: 2, typ: "UInt32"}, // 32-bit platforms only  
    {name: "Avg64u", argLength: 2, typ: "UInt64"}, // 64-bit platforms only
```

SSA Basic Block

cmd/compile/internal/ssa/block.go

```
99//      kind          control  successors
100// -----
101//      Exit          return mem      []
102//      Plain         nil           [next]
103//      If            a boolean Value [then, else]
104//      Defer         mem          [nopanic, panic]
105type BlockKind int8

82type Edge struct {
83 // block edge goes to (in a Succs list) or from (in a Preds list)
84 b *Block
85 // index of reverse edge. Invariant:
86 //   e := x.Succs[idx]
87 //   e.b.Preds[e.i] = Edge{x,idx}
88 // and similarly for predecessors.
89 i int
90}
```

```
12// Block represents a basic block in the control flow graph of a function.
13type Block struct {
14 // A unique identifier for the block. The system will attempt to allocate
15 // these IDs densely, but no guarantees.
16 ID ID
17
18 // Source position for block's control operation
19 Pos src.XPos
20
21 // The kind of block this is.
22 Kind BlockKind
23
24 // Likely direction for branches.
25 // If BranchLikely, Succs[0] is the most likely branch taken.
26 // If BranchUnlikely, Succs[1] is the most likely branch taken.
27 // Ignored if len(Succs) < 2.
28 // Fatal if not BranchUnknown and len(Succs) > 2.
29 Likely BranchPrediction
30
31 // After flagalloc, records whether flags are live at the end of the block.
32 FlagsLiveAtEnd bool
33
34 // Subsequent blocks, if any. The number and order depend on the block kind.
35 Succs []Edge
36
37 // Inverse of successors.
38 // The order is significant to Phi nodes in the block.
39 // TODO: predecessors is a pain to maintain. Can we somehow order phi
40 // arguments by block id and have this field computed explicitly when needed?
41 Preds []Edge
42
43 // A value that determines how the block is exited. Its value depends on the kind
44 // of the block. For instance, a BlockIf has a boolean control value and BlockExit
45 // has a memory control value.
46 Control *Value
47
48 // Auxiliary info for the block. Its value depends on the Kind.
49 Aux interface{}
50
51 // The unordered set of Values that define the operation of this block.
52 // The list must include the control value, if any. ([TODO: need this last condition?])
53 // After the scheduling pass, this list is ordered.
54 Values []*Value
```


SSA Example

```

var data2 [26]int64

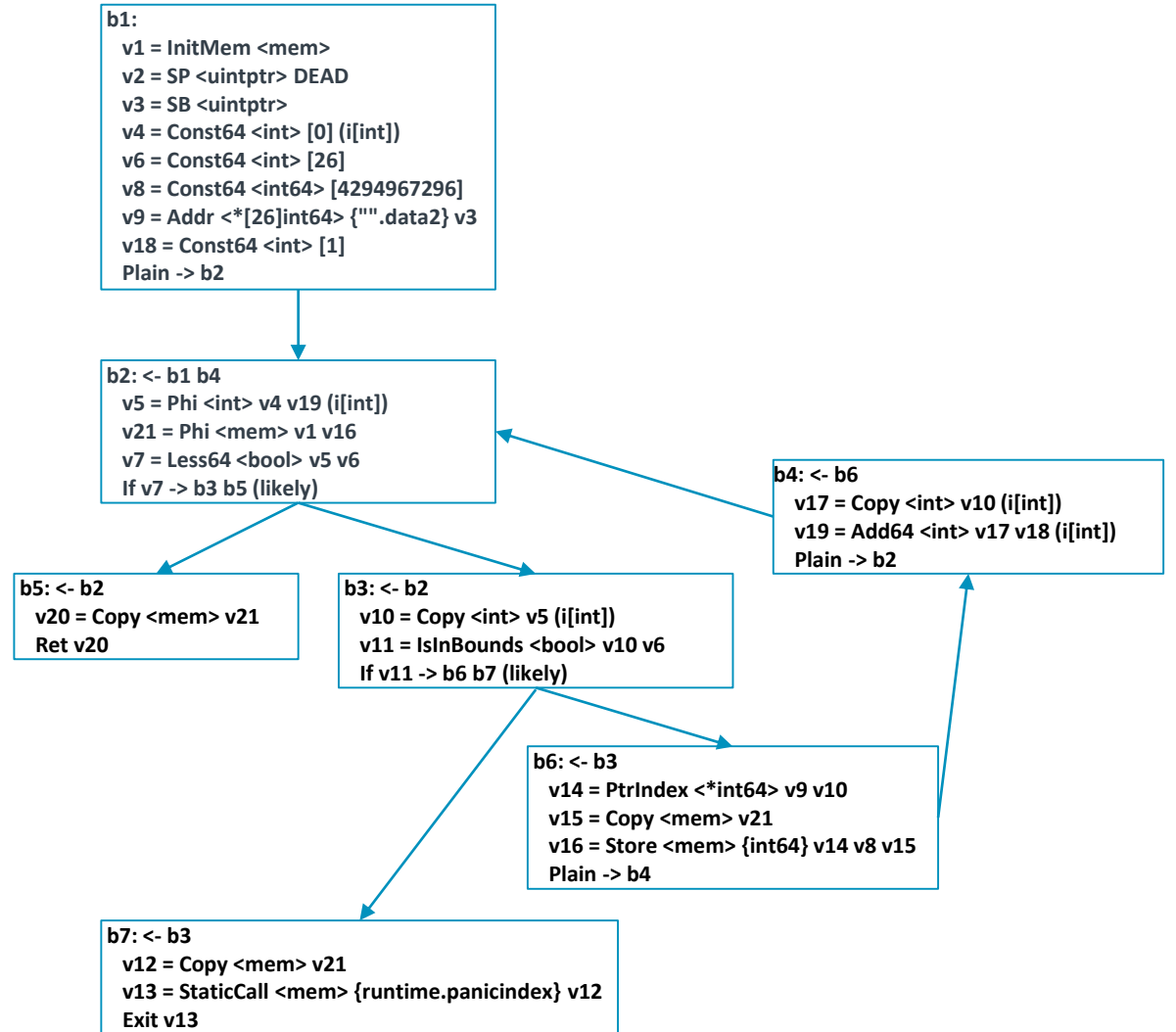
func Init() {
    for i := 0; i < 26; i++ {
        data2[i] = 0x100000000
    }
}

```

```

Init <T>
b1:
v1 = InitMem <mem>
v2 = SP <uintptr> DEAD
v3 = SB <uintptr>
v4 = Const64 <int> [0] (i[int])
v6 = Const64 <int> [26]
v8 = Const64 <int64> [4294967296]
v9 = Addr <*[26]int64> {""}.data2 v3
v18 = Const64 <int> [1]
Plain -> b2
b2: <- b1 b4
v5 = Phi <int> v4 v19 (i[int])
v21 = Phi <mem> v1 v16
v7 = Less64 <bool> v5 v6
If v7 -> b3 b5 (likely)
b3: <- b2
v10 = Copy <int> v5 (i[int])
v11 = IsInBounds <bool> v10 v6
If v11 -> b6 b7 (likely)
b4: <- b6
v17 = Copy <int> v10 (i[int])
v19 = Add64 <int> v17 v18 (i[int])
Plain -> b2
b5: <- b2
v20 = Copy <mem> v21
Ret v20
b6: <- b3
v14 = PtrIndex <*int64> v9 v10
v15 = Copy <mem> v21
v16 = Store <mem> {int64} v14 v8 v15
Plain -> b4
b7: <- b3
v12 = Copy <mem> v21
v13 = StaticCall <mem> {runtime.panicindex} v12
Exit v13
name i[int]: [v4 v5 v10 v17 v19]

```



Dump via env var: GOSSAFUNC
\$ export GOSSAFUNC="Init"

Go SSA viewer

<https://golang-ssaview.herokuapp.com/>

Uses the wonderful golang.org/x/tools/go/ssa package.)

```
status: done
Input
1 package test
2 var data2 [26]int64
3
4 func Init() {
5     for i := 0; i < 26; i++ {
6         data2[i] = 0x100000000
7     }
8 }
9

Result
1 package main.go:
2 func Init func()
3 var data2 [26]int64
4 func init func()
5 var init$guard bool
6
7 # Name: main.go.init
8 # Package: main.go
9 # Synthetic: package initializer
10 func init():
11 0: entry P:0 S:2
12     t0 = *init$guard          bool
13     if t0 goto 2 else 1
14 1: init.start P:1 S:1
15     *init$guard = true:bool
16     jump 2
17 2: init.done P:2 S:0
18     return
19
20 # Name: main.go.Init
21 # Package: main.go
22 # Location: main.go:4:6
23 # Locals:
24 # 0: t0 int
25 func Init():
26 0: entry P:0 S:1
27     t0 = local int (i)        *int
28     *t0 = 0:int
29     jump 3
30 1: for.body P:1 S:1
31     t1 = *t0                  int
32     t2 = &data2[t1]          *int64
33     *t2 = 4294967296:int64
34     t3 = *t0                  int
35     t4 = t3 + 1:int          int
36     *t0 = t4
37     jump 3
38 2: for.done P:1 S:0
39     rundefers
40     return
41 3: for.loop P:2 S:2
42     t5 = *t0                  int
43     t6 = t5 < 26:int         bool
44     if t6 goto 1 else 2
45
```

Compile

cmd/compile/internal/ssa/compile.go
func Compile(f *Func)

Compile modifies f so that on return all Values in f map to 0 or 1 assembly instructions of the target architecture

```
330 // list of passes for the compiler
331 var passes = [...]pass{
332     // TODO: combine phielim and copyelim into a single pass?
333     {name: "early phielim", fn: phielim},
334     {name: "early copyelim", fn: copyelim},
335     {name: "early deadcode", fn: deadcode}, // remove generated dead code to avoid doing pointless work during opt
336     {name: "short circuit", fn: shortcircuit},
337     {name: "decompose user", fn: decomposeUser, required: true},
338     {name: "opt", fn: opt, required: true}, // TODO: split required rules and optimizing rules
339     {name: "zero arg cse", fn: zcse, required: true}, // required to merge OpSB values
340     {name: "opt deadcode", fn: deadcode, required: true}, // remove any blocks orphaned during opt
341     {name: "generic cse", fn: cse},
342     {name: "phiopt", fn: phiopt},
343     {name: "nilcheckelim", fn: nilcheckelim},
344     {name: "prove", fn: prove},
345     {name: "loopbce", fn: loopbce},
346     {name: "decompose builtin", fn: decomposeBuiltin, required: true},
347     {name: "softfloat", fn: softfloat, required: true},
348     {name: "late opt", fn: opt, required: true}, // TODO: split required rules and optimizing rules
349     {name: "generic deadcode", fn: deadcode},
350     {name: "check bce", fn: checkbce},
351     {name: "fuse", fn: fuse},
352     {name: "dse", fn: dse},
353     {name: "writebarrier", fn: writebarrier, required: true}, // expand write barrier ops
354     {name: "insert resched checks", fn: insertLoopReschedChecks,
355         disabled: objabi.PreemptibleLoopsEnabled == 0}, // insert resched checks in loops.
356     {name: "tighten", fn: tighten}, // move values closer to their uses
```

Machine-independent Optimization Example

Before	After
<pre>90 b1: 91 v1 = InitMem <mem> 92 v2 = SP <uintptr> 93 v5 = Addr <*uint64> {~r1} v2 94 v6 = Arg <uint64> {a} 95 v8 = Const64 <uint64> [7] 96 v9 = Div64u <uint64> v6 v8 97 v10 = VarDef <mem> {~r1} v1 98 v11 = Store <mem> {uint64} v5 v9 v10 99 Ret v11 func Div(a uint64) uint64 { return a/7; }</pre>	<pre>103 b1: 104 v1 = InitMem <mem> 105 v2 = SP <uintptr> 106 v5 = Addr <*uint64> {~r1} v2 107 v6 = Arg <uint64> {a} 108 v8 = Const64 <uint64> [7] DEAD 109 v10 = VarDef <mem> {~r1} v1 110 v3 = Const64 <uint64> [2635249153387078803] 111 v12 = Const64 <uint64> [2] 112 v4 = Hmul64u <uint64> v3 v6 113 v7 = Avg64u <uint64> v6 v4 114 v9 = Rsh64Ux64 <uint64> v7 v12 115 v11 = Store <mem> {uint64} v5 v9 v10 116 Ret v11</pre>

Optimization Mathematical Background

Technique from <https://gmplib.org/~tege/divcnst-pldi94.pdf>

$$\lfloor x / c \rfloor = \lfloor x * (2^e/c) / 2^e \rfloor.$$

Dividing by 2^e is easy. $2^e/c$ isn't an integer, unfortunately. So we must approximate it. Let's call its approximation m .

$$e = n + s, \text{ with } s = \lceil \log_2(c) \rceil.$$

An additional complication arises because m has $n+1$ bits in it. Hardware restricts us to n bit by n bit multiplies. We divide into 3 cases:

Case 1: m is even.

$$\begin{aligned}\lfloor x / c \rfloor &= \lfloor x * m / 2^{(n+s)} \rfloor \\ \lfloor x / c \rfloor &= \lfloor x * (m/2) / 2^{(n+s-1)} \rfloor \\ \lfloor x / c \rfloor &= \lfloor x * (m/2) / 2^n / 2^{(s-1)} \rfloor \\ \lfloor x / c \rfloor &= \lfloor \lfloor x * (m/2) / 2^n \rfloor / 2^{(s-1)} \rfloor \\ &\text{multiply + shift}\end{aligned}$$

Case 2: c is even.

$$\begin{aligned}\lfloor x / c \rfloor &= \lfloor (x/2) / (c/2) \rfloor \\ \lfloor x / c \rfloor &= \lfloor \lfloor x/2 \rfloor / (c/2) \rfloor \\ &\text{This is just the original problem, with } x' = \lfloor x/2 \rfloor, c' = c/2, n' = n-1. \\ &s' = s-1 \\ m' &= \lceil 2^{(n'+s')}/c' \rceil \\ &= \lceil 2^{(n+s-1)}/c \rceil \\ &= \lceil m/2 \rceil \\ \lfloor x / c \rfloor &= \lfloor x' * m' / 2^{(n'+s')} \rfloor \\ \lfloor x / c \rfloor &= \lfloor \lfloor x/2 \rfloor * \lceil m/2 \rceil / 2^{(n+s-2)} \rfloor \\ \lfloor x / c \rfloor &= \lfloor \lfloor \lfloor x/2 \rfloor * \lceil m/2 \rceil / 2^n \rfloor / 2^{(s-2)} \rfloor \\ &\text{shift + multiply + shift}\end{aligned}$$

Case 3: everything else

$$\begin{aligned}&\text{let } k = m - 2^n. k \text{ fits in } n \text{ bits.} \\ \lfloor x / c \rfloor &= \lfloor x * m / 2^{(n+s)} \rfloor \\ \lfloor x / c \rfloor &= \lfloor x * (2^n + k) / 2^{(n+s)} \rfloor \\ \lfloor x / c \rfloor &= \lfloor (x + x * k / 2^n) / 2^s \rfloor \\ \lfloor x / c \rfloor &= \lfloor (x + \lfloor x * k / 2^n \rfloor) / 2^s \rfloor \\ \lfloor x / c \rfloor &= \lfloor (x + \lfloor x * k / 2^n \rfloor) / 2^s \rfloor \\ \lfloor x / c \rfloor &= \lfloor \lfloor (x + \lfloor x * k / 2^n \rfloor) / 2 \rfloor / 2^{(s-1)} \rfloor \\ &\text{multiply + avg + shift}\end{aligned}$$

Generic SSA Rules Example

cmd/compile/internal/ssa/gen/generic.rules

1103 // For 64-bit divides on 64-bit machines

1104 // (64-bit divides on 32-bit machines are lowered to a runtime call by the walk pass.)

1105 (Div64u x (Const64 [c])) && umagicOK(64, c) && config.RegSize == 8 && umagic(64,c).m&1 == 0 ->

1106 (Rsh64Ux64 <typ.UInt64>

1107 (Hmul64u <typ.UInt64>

1108 (Const64 <typ.UInt64> [int64(1<<63+umagic(64,c).m/2)])

1109 x)

1110 (Const64 <typ.UInt64> [umagic(64,c).s-1]))

1111 (Div64u x (Const64 [c])) && umagicOK(64, c) && config.RegSize == 8 && c&1 == 0 ->

1112 (Rsh64Ux64 <typ.UInt64>

1113 (Hmul64u <typ.UInt64>

1114 (Const64 <typ.UInt64> [int64(1<<63+(umagic(64,c).m+1)/2)])

1115 (Rsh64Ux64 <typ.UInt64> x (Const64 <typ.UInt64> [1])))

1116 (Const64 <typ.UInt64> [umagic(64,c).s-2]))

1117 (Div64u x (Const64 [c])) && umagicOK(64, c) && config.RegSize == 8 ->

1118 (Rsh64Ux64 <typ.UInt64>

1119 (Avg64u

1120 x

1121 (Hmul64u <typ.UInt64>

1122 (Const64 <typ.UInt64> [int64(umagic(64,c).m)])

1123 x))

1124 (Const64 <typ.UInt64> [umagic(64,c).s-1]))

1125

Case 1: m is even.

$$\lfloor x / c \rfloor = \lfloor x * m / 2^{(n+s)} \rfloor$$

$$\lfloor x / c \rfloor = \lfloor x * (m/2) / 2^{(n+s-1)} \rfloor$$

$$\lfloor x / c \rfloor = \lfloor x * (m/2) / 2^n / 2^{(s-1)} \rfloor$$

$$\lfloor x / c \rfloor = \lfloor \lfloor x * (m/2) / 2^n \rfloor / 2^{(s-1)} \rfloor$$

multiply + shift

Case 2: c is even.

$$\lfloor x / c \rfloor = \lfloor (x/2) / (c/2) \rfloor$$

$$\lfloor x / c \rfloor = \lfloor \lfloor x/2 \rfloor / (c/2) \rfloor$$

This is just the original problem, with $x' = \lfloor x/2 \rfloor$, $c' = c/2$, $n' = n-1$.

$$s' = s-1$$

$$m' = \lceil 2^{(n'+s')} / c' \rceil$$

$$= \lceil 2^{(n+s-1)} / c \rceil$$

$$= \lceil m/2 \rceil$$

$$\lfloor x / c \rfloor = \lfloor x' * m' / 2^{(n'+s')} \rfloor$$

$$\lfloor x / c \rfloor = \lfloor \lfloor x/2 \rfloor * \lceil m/2 \rceil / 2^{(n+s-2)} \rfloor$$

$$\lfloor x / c \rfloor = \lfloor \lfloor \lfloor x/2 \rfloor * \lceil m/2 \rceil / 2^n \rfloor / 2^{(s-2)} \rfloor$$

shift + multiply + shift

Case 3: everything else

let $k = m - 2^n$. k fits in n bits.

$$\lfloor x / c \rfloor = \lfloor x * m / 2^{(n+s)} \rfloor$$

$$\lfloor x / c \rfloor = \lfloor x * (2^n + k) / 2^{(n+s)} \rfloor$$

$$\lfloor x / c \rfloor = \lfloor (x + x * k / 2^n) / 2^s \rfloor$$

$$\lfloor x / c \rfloor = \lfloor (x + \lfloor x * k / 2^n \rfloor) / 2^s \rfloor$$

$$\lfloor x / c \rfloor = \lfloor (x + \lfloor x * k / 2^n \rfloor) / 2^s \rfloor$$

$$\lfloor x / c \rfloor = \lfloor \lfloor (x + \lfloor x * k / 2^n \rfloor) / 2 \rfloor / 2^{(s-1)} \rfloor$$

multiply + avg + shift

Machine-independent Optimization Example

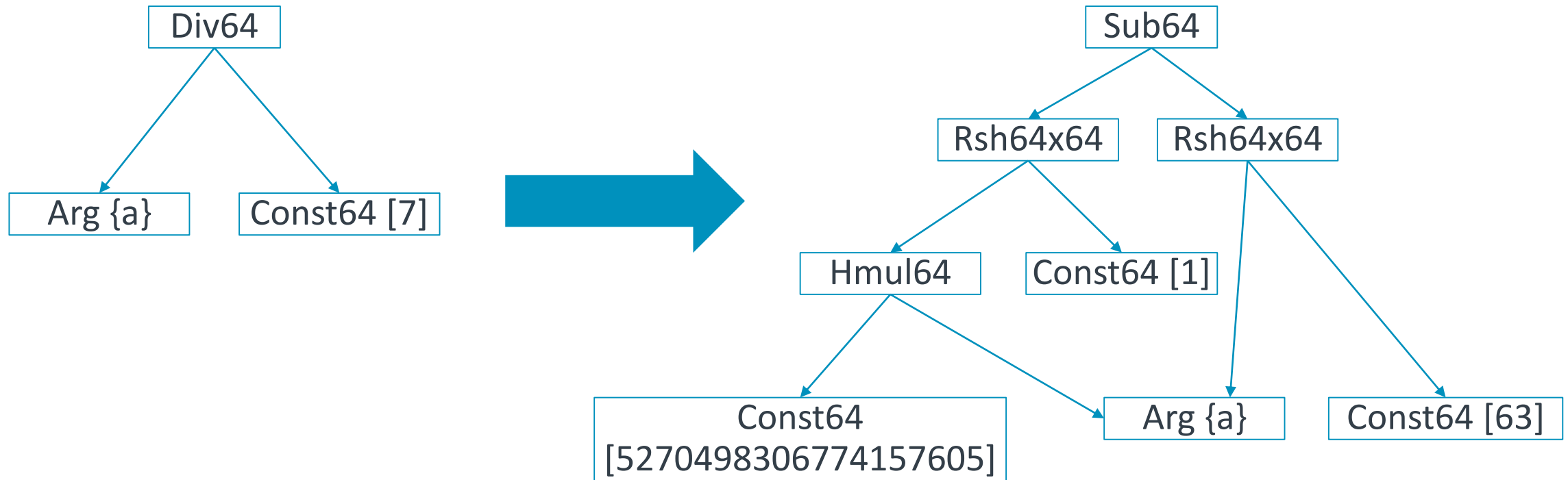
Before	After
<pre>90 b1: 91 v1 = InitMem <mem> 92 v2 = SP <uintptr> 93 v5 = Addr <*int64> {~r1} v2 94 v6 = Arg <int64> {a} 95 v8 = Const64 <int64> [7] 96 v9 = Div64 <int64> v6 v8 97 v10 = VarDef <mem> {~r1} v1 98 v11 = Store <mem> {int64} v5 v9 v10 99 Ret v11 func Div(a int64) int64 { return a/7; }</pre>	<pre>103 b1: 104 v1 = InitMem <mem> 105 v2 = SP <uintptr> 106 v5 = Addr <*int64> {~r1} v2 107 v6 = Arg <int64> {a} 108 v8 = Const64 <int64> [7] DEAD 109 v10 = VarDef <mem> {~r1} v1 110 v3 = Const64 <uint64> [5270498306774157605] 111 v12 = Const64 <uint64> [1] 112 v14 = Const64 <uint64> [63] 113 v4 = Hmul64 <int64> v3 v6 114 v13 = Rsh64x64 <int64> v6 v14 115 v7 = Rsh64x64 <int64> v4 v12 116 v9 = Sub64 <int64> v7 v13 117 v11 = Store <mem> {int64} v5 v9 v10 118 Ret v11</pre>

Generic SSA Rules Implementation Example

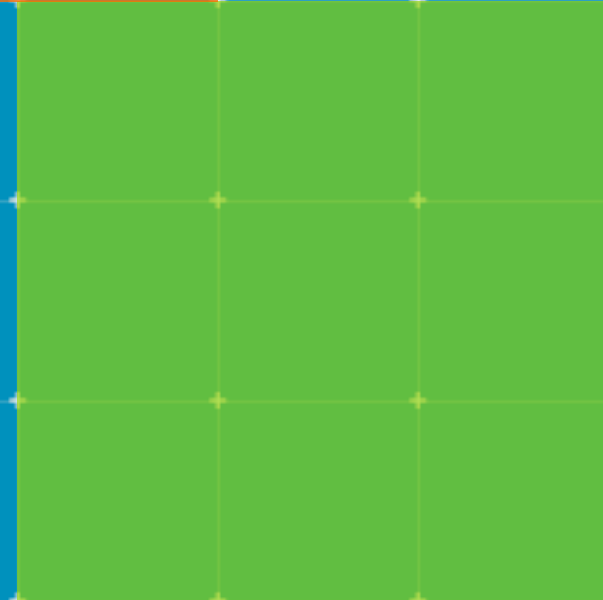
```
1214 (Div64 <t> x (Const64 [c])) && smagicOK(64,c) && smagic(64,c).m&1 == 0 ->
1215 (Sub64 <t>
1216 (Rsh64x64 <t>
1217 (Hmul64 <t>
1218 (Const64 <typ.UInt64> [int64(smagic(64,c).m/2]))
1219 x)
1220 (Const64 <typ.UInt64> [smagic(64,c).s-1]))
1221 (Rsh64x64 <t>
1222 x
1223 (Const64 <typ.UInt64> [63])))
```

```
// match: (Div64 <t> x (Const64 [c]))
// cond: smagicOK(64,c) && smagic(64,c).m&1 == 0
// result: (Sub64 <t> (Rsh64x64 <t> (Hmul64 <t> (Const64 <typ.UInt64> [int64(smagic(64,c).m/2]))
for {
    t := v.Type
    _ = v.Args[1]
    x := v.Args[0]
    v_1 := v.Args[1]
    if v_1.Op != OpConst64 {
        break
    }
    c := v_1.AuxInt
    if !(smagicOK(64, c) && smagic(64, c).m&1 == 0) {
        break
    }
    v.reset(OpSub64)
    v.Type = t
    v0 := b.NewValue0(v.Pos, OpRsh64x64, t)
    v1 := b.NewValue0(v.Pos, OpHmul64, t)
    v2 := b.NewValue0(v.Pos, OpConst64, typ.UInt64)
    v2.AuxInt = int64(smagic(64, c).m / 2)
    v1.AddArg(v2)
    v1.AddArg(x)
    v0.AddArg(v1)
    v3 := b.NewValue0(v.Pos, OpConst64, typ.UInt64)
    v3.AuxInt = smagic(64, c).s - 1
    v0.AddArg(v3)
    v.AddArg(v0)
    v4 := b.NewValue0(v.Pos, OpRsh64x64, t)
    v4.AddArg(x)
    v5 := b.NewValue0(v.Pos, OpConst64, typ.UInt64)
    v5.AuxInt = 63
    v4.AddArg(v5)
    v.AddArg(v4)
    return true
}
```


Machine-independent Optimization Example



Back End



Machine-dependent Passes

```
357 {name: "lower", fn: lower, required: true},
358 {name: "lowered cse", fn: cse},
359 {name: "elim unread autos", fn: elimUnreadAutos},
360 {name: "lowered deadcode", fn: deadcode, required: true},
361 {name: "checkLower", fn: checkLower, required: true},
362 {name: "late phielim", fn: phielim},
363 {name: "late copyelim", fn: copyelim},
364 {name: "phi tighten", fn: phiTighten},
365 {name: "late deadcode", fn: deadcode},
366 {name: "critical", fn: critical, required: true}, // remove critical edges
367 {name: "likelyadjust", fn: likelyadjust},
368 {name: "layout", fn: layout, required: true}, // schedule blocks
369 {name: "schedule", fn: schedule, required: true}, // schedule values
370 {name: "late nilcheck", fn: nilcheckelim2},
371 {name: "flagalloc", fn: flagalloc, required: true}, // allocate flags register
372 {name: "regalloc", fn: regalloc, required: true}, // allocate int & float registers + stack slots
373 {name: "loop rotate", fn: loopRotate},
374 {name: "stackframe", fn: stackframe, required: true},
375 {name: "trim", fn: trim}, // remove empty blocks
```

ARM64 SSA Ops

cmd/compile/internal/ssa/gen/ARM64Ops.go

```
ops := []opData{
    // binary ops
    {name: "ADD", argLength: 2, reg: gp21, asm: "ADD", commutative: true}, // arg0 + arg1
    {name: "ADDconst", argLength: 1, reg: gp11sp, asm: "ADD", aux: "Int64"}, // arg0 + auxInt
    {name: "SUB", argLength: 2, reg: gp21, asm: "SUB"}, // arg0 - arg1
    {name: "SUBconst", argLength: 1, reg: gp11, asm: "SUB", aux: "Int64"}, // arg0 - auxInt
    {name: "MUL", argLength: 2, reg: gp21, asm: "MUL", commutative: true}, // arg0 * arg1
    {name: "MULW", argLength: 2, reg: gp21, asm: "MULW", commutative: true}, // arg0 * arg1, 32-bit
    {name: "MULH", argLength: 2, reg: gp21, asm: "SMULH", commutative: true}, // (arg0 * arg1) >> 64, signed
    {name: "DIV", argLength: 2, reg: gp21, asm: "SDIV"}, // arg0 / arg1, signed
    {name: "UDIV", argLength: 2, reg: gp21, asm: "UDIV"}, // arg0 / arg1, unsigned
    {name: "DIVW", argLength: 2, reg: gp21, asm: "SDIVW"}, // arg0 / arg1, signed, 32 bit
    {name: "UDIVW", argLength: 2, reg: gp21, asm: "UDIVW"}, // arg0 / arg1, unsigned, 32 bit
    {name: "MOD", argLength: 2, reg: gp21, asm: "REM"}, // arg0 % arg1, signed
    {name: "UMOD", argLength: 2, reg: gp21, asm: "UREM"}, // arg0 % arg1, unsigned
    {name: "MODW", argLength: 2, reg: gp21, asm: "REMW"}, // arg0 % arg1, signed, 32 bit
    {name: "UMODW", argLength: 2, reg: gp21, asm: "UREMW"}, // arg0 % arg1, unsigned, 32 bit
}
```

Convert to Machine-dependent Ops

Before	After
410 Div <T> 411 b1: 412 v1 = InitMem <mem> 413 v2 = SP <uintptr> 414 v5 = Addr <*int64> {~r1} v2 415 v6 = Arg <int64> {a} 416 v10 = VarDef <mem> {~r1} v1 417 v3 = Const64 <uint64> [5270498306774157605] 418 v12 = Const64 <uint64> [1] 419 v14 = Const64 <uint64> [63] 420 v4 = Hmul64 <int64> v3 v6 421 v13 = Rsh64x64 <int64> v6 v14 422 v7 = Rsh64x64 <int64> v4 v12 423 v9 = Sub64 <int64> v7 v13 424 v11 = Store <mem> {int64} v5 v9 v10 425 Ret v11	428 Div <T> 429 b1: 430 v1 = InitMem <mem> 431 v2 = SP <uintptr> 432 v5 = MOVDaddr <*int64> {~r1} v2 DEAD 433 v6 = Arg <int64> {a} 434 v10 = VarDef <mem> {~r1} v1 435 v3 = MOVDconst <uint64> [5270498306774157605] 436 v12 = MOVDconst <uint64> [1] DEAD 437 v13 = SRAconst <int64> [63] v6 DEAD 438 v14 = MOVDconst <uint64> [63] DEAD 439 v15 = MOVDconst <uint64> [63] DEAD 440 v16 = FlagLT_ULT <flags> DEAD 441 v18 = MOVDconst <uint64> [63] DEAD 442 v19 = FlagLT_ULT <flags> DEAD 443 v4 = MULH <int64> v3 v6 444 v7 = SRAconst <int64> [1] v4 445 v9 = SUBshiftRA <int64> [63] v7 v6 446 v11 = MOVDstore <mem> {~r1} v2 v9 v10 447 Ret v11

ARM64 Rules Examples

(Hmul64 x y) -> (MULH x y)

(Rsh64x64 x y) -> (SRA x (CSELULT <y.Type> y (MOVDconst <y.Type> [63]) (CMPconst [64] y)))

(CMPconst (MOVDconst [x]) [y]) && int64(x)<int64(y) && uint64(x)<uint64(y) -> (FlagLT_ULT)

(CSELULT x _ (FlagLT_ULT)) -> x

(Sub64 x y) -> (SUB x y)

(SUB x (SRAconst [c] y)) -> (SUBshiftRA x y [c])

Register Allocation

Linear scan register allocator (cmd/compile/internal/ssa/regalloc.go)

- Treat the whole function as a single long basic block and run through it using a greedy register allocator.
- Spill the value whose next use is farthest in the future

Before	After
648 Div <T> 649 b1: 650 v1 = InitMem <mem> 651 v10 = VarDef <mem> {~r1} v1 652 v2 = SP <uintptr> 653 v6 = Arg <int64> {a} 654 v3 = MOVDconst <uint64> [5270498306774157605] 655 v4 = MULH <int64> v3 v6 656 v7 = SRAconst <int64> [1] v4 657 v9 = SUBshiftRA <int64> [63] v7 v6 658 v11 = MOVDstore <mem> {~r1} v2 v9 v10 659 Ret v11	662 Div <T> 663 b1: 664 v1 = InitMem <mem> 665 v10 = VarDef <mem> {~r1} v1 666 v2 = SP <uintptr> : SP 667 v6 = Arg <int64> {a} : a[int64] 668 v19 = MOVDconst <uint64> [5270498306774157605] : R0 669 v18 = LoadReg <int64> v6 : R1 670 v4 = MULH <int64> v19 v18 : R0 671 v7 = SRAconst <int64> [1] v4 : R0 672 v9 = SUBshiftRA <int64> [63] v7 v18 : R0 673 v11 = MOVDstore <mem> {~r1} v2 v9 v10 674 Ret v11

Code Gen

Prog describes a single machine instruction

```
type Prog struct {
    Ctxt    *Link    // linker context
    Link    *Prog    // next Prog in linked list
    From    Addr    // first source operand
    RestArgs []Addr  // can pack any operands that not fit into {Prog.From, Prog.To}
    To      Addr    // destination operand (second is RegTo2 below)
    Pcond   *Prog    // target of conditional jump
    Forwd   *Prog    // for x86 back end
    Rel     *Prog    // for x86, arm back ends
    Pc      int64   // for back ends or assembler: virtual or actual program counter, depending on phase
    Pos     src.XPos // source position of this instruction
    Spadj   int32   // effect of instruction on stack pointer (increment or decrement amount)
    As      As     // assembler opcode
    Reg     int16  // 2nd source operand
    RegTo2  int16  // 2nd destination operand
    Mark    uint16  // bitmask of arch-specific items
    Optab   uint16  // arch-specific opcode index
    Scond   uint8   // condition bits for conditional instruction (e.g., on ARM)
    Back    uint8   // for x86 back end: backwards branch state
    Ft      uint8   // for x86 back end: type index of Prog.From
    Tt      uint8   // for x86 back end: type index of Prog.To
    Isize   uint8   // for x86 back end: size of the instruction in bytes
}
```

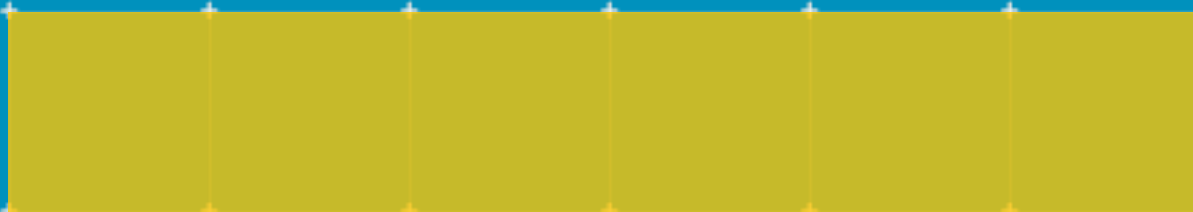
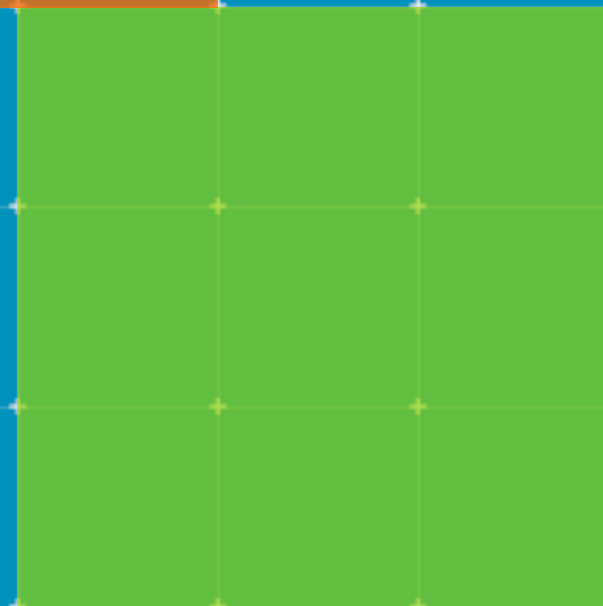

Code Gen

Emit Prog(s) for the Value(s)

[cmd/compile/internal/arm64/ssa.go](#)

Before	After
707 Div <T>	722 00000 (3) TEXT "" .Div(SB)
708 b1:
709 v1 = InitMem <mem>	
710 v10 = VarDef <mem> {~r1} v1	725 v19 00003 (4) MOVD \$5270498306774157605, R0
711 v2 = SP <uintptr> : SP	726 v18 00004 (4) MOVD "" .a(RSP), R1
712 v6 = Arg <int64> {a} : a[int64]	727 v4 00005 (4) SMULH R1, R0, R0
713 v19 = MOVDconst <uint64> [5270498306774157605] : R0	728 v7 00006 (4) ASR \$1, R0, R0
714 v18 = LoadReg <int64> v6 : R1	729 v9 00007 (4) SUB R1->63, R0, R0
715 v4 = MULH <int64> v19 v18 : R0	730 v11 00008 (4) MOVD R0, "" .~r1+8(RSP)
716 v7 = SRAconst <int64> [1] v4 : R0	731 b1 00009 (4) RET
717 v9 = SUBshiftRA <int64> [63] v7 v18 : R0	732 00010 (?) END
718 v11 = MOVDstore <mem> {~r1} v2 v9 v10	
719 Ret v11	

Intrinsic



Intrinsic

If a function is an intrinsic, the code for that function is usually inserted inline, avoiding the overhead of a function call and allowing highly efficient machine instructions to be emitted for that function.

An intrinsic is often faster than the equivalent inline assembly, because the optimizer has a built-in knowledge of how many intrinsics behave, so some optimizations can be available that are not available when inline assembly is used. Also, the optimizer can expand the intrinsic differently, align buffers differently, or make other adjustments depending on the context and arguments of the call.

Go source code	Intrinsic disablement	Intrinsic enablement
<pre>package test import "math" func MySqrt(x float64) float64 { return math.Sqrt(x) }</pre>	<pre>MOVD 16(R28), R1 MOVD RSP, R2 CMP R1, R2 BLS 9(PC) MOVD.W R30, -32(RSP) MOVD 40(RSP), F0 MOVD F0, 8(RSP) CALL math.Sqrt(SB) MOVD 16(RSP), F0 MOVD F0, 48(RSP) MOVD.P 32(RSP), R30 RET</pre>	<pre>MOVD 16(R28), R1 MOVD RSP, R2 CMP R1, R2 BLS 5(PC) MOVD 8(RSP), F0 FSQRTD F0, F0 MOVD F0, 16(RSP) RET</pre>

Intrinsic

1. Register a hook to convert a call node into SSA value that implements that call as an intrinsic

[cmd/compile/internal/gc/ssa.go](#)

```
2912  /***** math *****/
2913  addF("math", "Sqrt",
2914      func(s *state, n *Node, args []*ssa.Value) *ssa.Value {
2915      return s.newValue1(ssa.OpSqrt, types.Types[TFLOAT64], args[0])
2916      },
2917      sys.AMD64, sys.ARM, sys.ARM64, sys.MIPS, sys.PPC64, sys.S390X)
```

```
3017  addF("math/bits", "TrailingZeros16",
3018      func(s *state, n *Node, args []*ssa.Value) *ssa.Value {
3019      x := s.newValue1(ssa.OpZeroExt16to64, types.Types[TUINT64], args[0])
3020      c := s.constInt64(types.Types[TUINT64], 1<<16)
3021      y := s.newValue2(ssa.OpOr64, types.Types[TUINT64], x, c)
3022      return s.newValue1(ssa.OpCtz64, types.Types[TINT], y)
3023      },
3024      sys.AMD64, sys.ARM64, sys.S390X)
```

Intrinsic

2. Add ARM64 ops

```
ops := []opData{
    {name: "MVN", argLength: 1, reg: gp11, asm: "MVN"},    // ^arg0
    {name: "NEG", argLength: 1, reg: gp11, asm: "NEG"},    // -arg0
    {name: "FNEGS", argLength: 1, reg: fp11, asm: "FNEGS"}, // -arg0, float32
    {name: "FNEGD", argLength: 1, reg: fp11, asm: "FNEGD"}, // -arg0, float64
    {name: "FSQRTD", argLength: 1, reg: fp11, asm: "FSQRTD"}, // sqrt(arg0), float64
    {name: "REV", argLength: 1, reg: gp11, asm: "REV"},    // byte reverse, 64-bit
    {name: "REVW", argLength: 1, reg: gp11, asm: "REVW"},  // byte reverse, 32-bit
    {name: "REV16W", argLength: 1, reg: gp11, asm: "REV16W"}, // byte reverse in each 16-bit halfword, 32-bit
    {name: "RBIT", argLength: 1, reg: gp11, asm: "RBIT"},  // bit reverse, 64-bit
    {name: "RBITW", argLength: 1, reg: gp11, asm: "RBITW"}, // bit reverse, 32-bit
    {name: "CLZ", argLength: 1, reg: gp11, asm: "CLZ"},    // count leading zero, 64-bit
    {name: "CLZW", argLength: 1, reg: gp11, asm: "CLZW"},  // count leading zero, 32-bit
}
```

Intrinsic

3. Add rule to convert SSA generic ops to ARM64 ops (or progs)

```
cmd/compile/internal/ssa/gen/ARM64.rules
```

```
(Sqrt x) -> (FSQRTD x)
```

```
(ZeroExt16to64 x) -> (MOVHUreg x)
```

```
(Or64 x y) -> (OR x y)
```

```
(Ctz64 <t> x) -> (CLZ (RBIT <t> x))
```

4. Re-generate Ops and Rules

```
$ cd cmd/compile/internal/ssa/gen; go run *.go
```

Thank You!

Danke!

Merci!

谢谢!

ありがとう!

Gracias!

Kiitos!

감사합니다

धन्यवाद

arm