arm

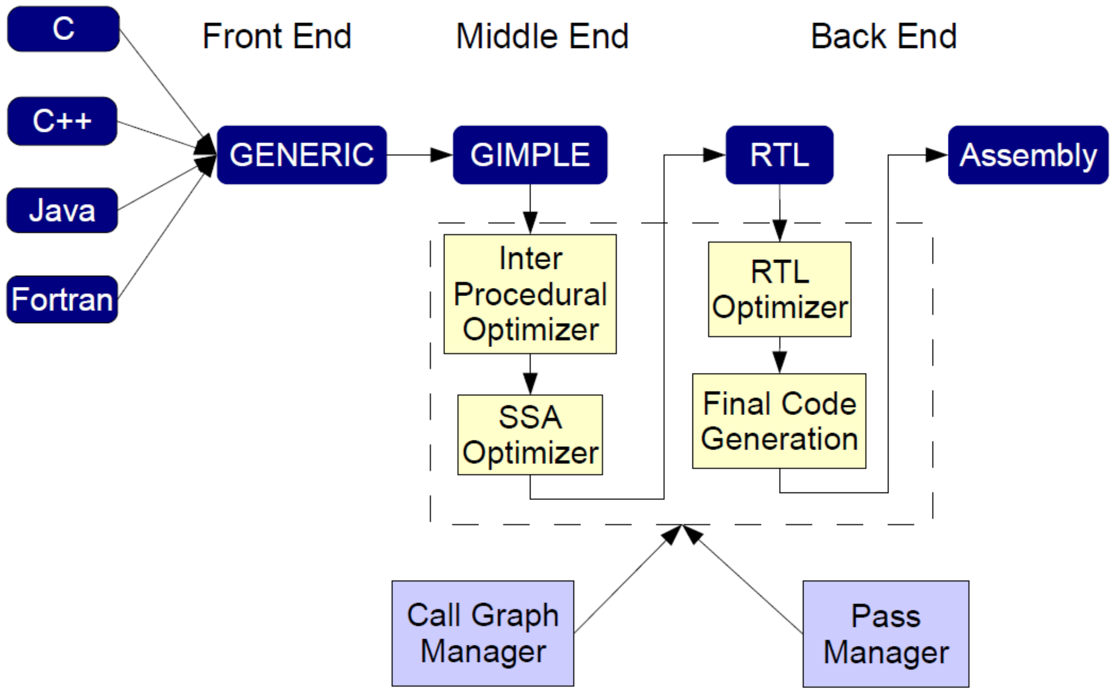
Golang Compiler Internals for arm64

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

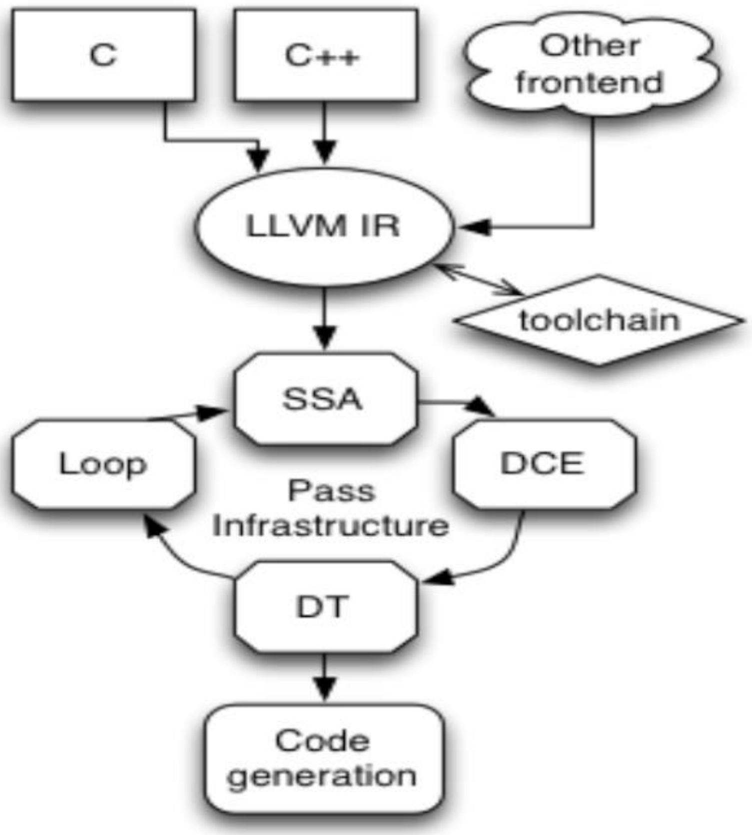
Overview

Classic Compiler Overview

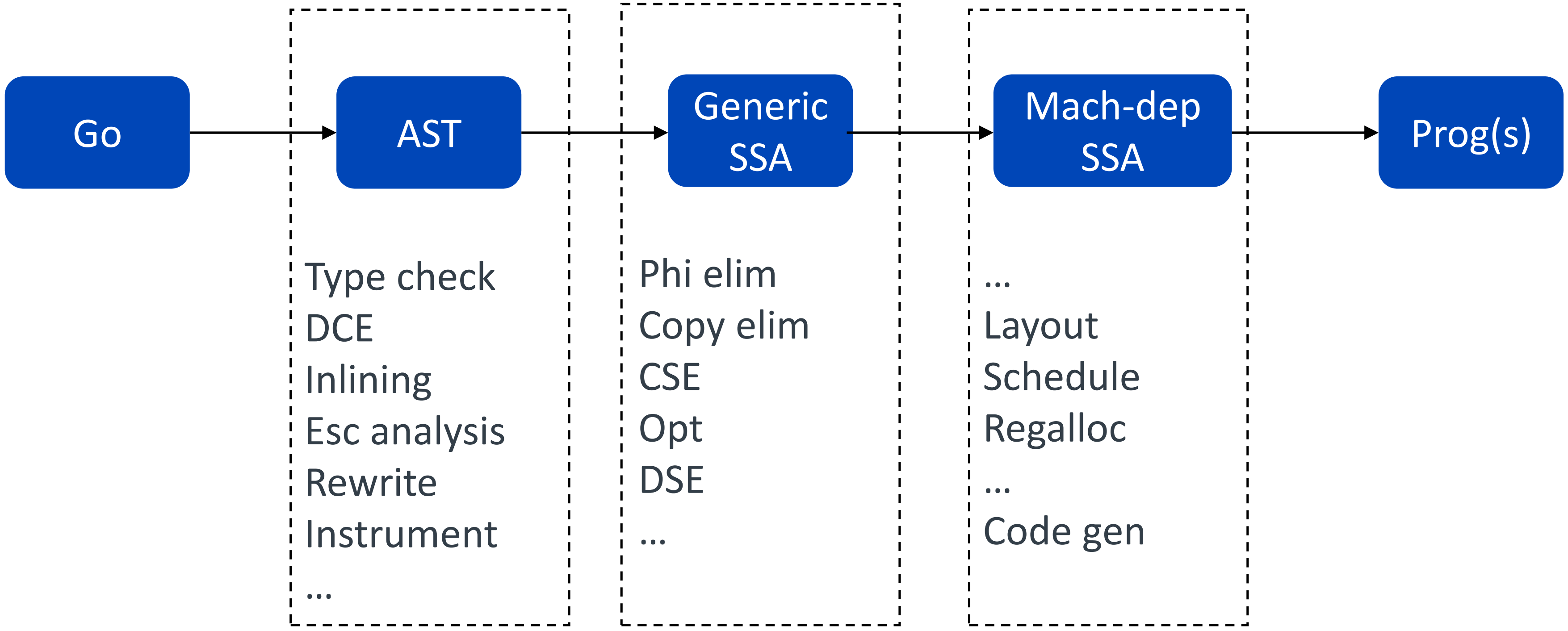
GCC



LLVM



Golang Overview



Front End

arm

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)

```
func (s *Scanner) Scan() (pos token.Pos, tok token.Token, lit string) {
scanAgain:
    s.skipWhitespace()

    // current token start
    pos = s.file.Pos(s.offset)

    // determine token value
    insertSemi := false
    switch ch := s.ch; {
    case isLetter(ch):
        lit = s.scanIdentifier()
        if len(lit) > 1 {
            // keywords are longer than one letter - avoid lookup otherwise
            tok = token.Lookup(lit)
            switch tok {
            case token.IDENT, token.BREAK, token.CONTINUE, token.FALLTHROUGH, token.RETURN:
                insertSemi = true
            }
        } else {
            insertSemi = true
            tok = token.IDENT
        }
    case '0' <= ch && ch <= '9':
        insertSemi = true
        tok, lit = s.scanNumber(false)
    default:
        s.next() // always make progress
        switch ch {
        case -1:
            if s.insertSemi {
                s.insertSemi = false // EOF consumed
                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)

The go/scanner and go/parser are used by various tools (gofmt, gotype, etc.) but they are **not used by the compiler** (cmd/compile). Go compiler uses a new internal package called syntax (cmd/compile/internal/syntax) which contains a new lexer, parser, and AST, all in one package. These are similar to the go/* packages, but streamlined and faster. The compiler receives ASTs from the syntax package which are then translated into the existing compiler-internal node structure (by cmd/compile/internal/gc/noder.go). In other words, **there are two syntax trees**, the new one generated by the syntax package (modern, light-weight), and the (older) one in the compiler, based on the Node data structure. The reason for this is that at some point we might want to replace the Nodes data structure with something more modern and light-weight. (by Robert Griesemer)

Syntax Tree Node

```go
type Node struct {
    // Tree structure.
    // Generic recursive walks should follow these fields.
    Left  *Node
    Right *Node
    Ninit Nodes
    Nbody Nodes
    List  Nodes
    Rlist Nodes

    // most nodes
    Type *types.Type
    Orig *Node // original form, for printing, and tracking copies of ONAMEs

    // func
    Func *Func

    // ONAME, OTYPE, OPACK, OLABEL, some OLITERAL
    Name *Name

    Sym *types.Sym  // various
    E   interface{} // Opt or Val, see methods below

    // Various. Usually an offset into a struct. For example:
    // - ONAME nodes that refer to local variables use it to identify their stack frame position.
    // - ODOT, ODOTPTR, and OINDREGSP use it to indicate offset relative to their base address.
    // - OSTRUCTKEY uses it to store the named field's offset.
    // - Named OLITERALs use it to to store their ambient iota value.
    // Possibly still more uses. If you find any, document them.
    Xoffset int64

    Pos src.XPos

    flags bitset32

    Esc uint16 // EscXXX

    Op    Op
    Etype types.EType // op for OASOP, etype for OTYPE, exclam for export, 6g saved reg, ChanDir for OTCHAN, for OINDEXMAP 1=LHS,0=RHS
}
```

```go
// 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
OARRAYBYTESTR       // Type(Left) (Type is string, Left is a []byte)
OARRAYBYTESTRTMP // 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)
OSTRARRAYBYTETMP // 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)
OCOMPLIT            // 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)
```

arm

Syntax Tree Example

```go
package test

func fact(n int) int {
	if n == 0 {
		return 1
	}
	return n * fact(n-1)
}
```

Dump by compile flag: “-W”

$ go tool compile -w example.go

```
. IF l(4) tc(1)
. . EQ l(4) tc(1) bool
. . . NAME-test.n a(true) g(2) l(3) x(0) class(PPARAM) f(1) tc(1) used int
. . . LITERAL-0 l(4) tc(1) int
. IF-body
. . RETURN l(5) tc(1)
. . RETURN-list
. . . LITERAL-1 l(5) tc(1) int

. AS l(7) tc(1)
. . NAME-test..autotmp_2 a(true) l(7) x(0) class(PAUTO) esc(N) tc(1) assigned used int
. . CALLFUNC l(7) tc(1) int
. . . NAME-test.fact a(true) l(3) x(0) class(PFUNC) tc(1) used FUNC-func(int) int
. . CALLFUNC-list
. . . SUB l(7) tc(1) int
. . . . NAME-test.n a(true) g(2) l(3) x(0) class(PPARAM) f(1) tc(1) used int
. . . . LITERAL-1 l(7) tc(1) int

. RETURN l(7) tc(1)
. RETURN-list
. . MUL l(7) tc(1) int
. . . NAME-test.n a(true) g(2) l(3) x(0) class(PPARAM) f(1) tc(1) used int
. . . NAME-test..autotmp_2 a(true) l(7) x(0) class(PAUTO) esc(N) tc(1) assigned used int

. VARKILL tc(1)
. . 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

arm

Middle End

arm

SSA Form

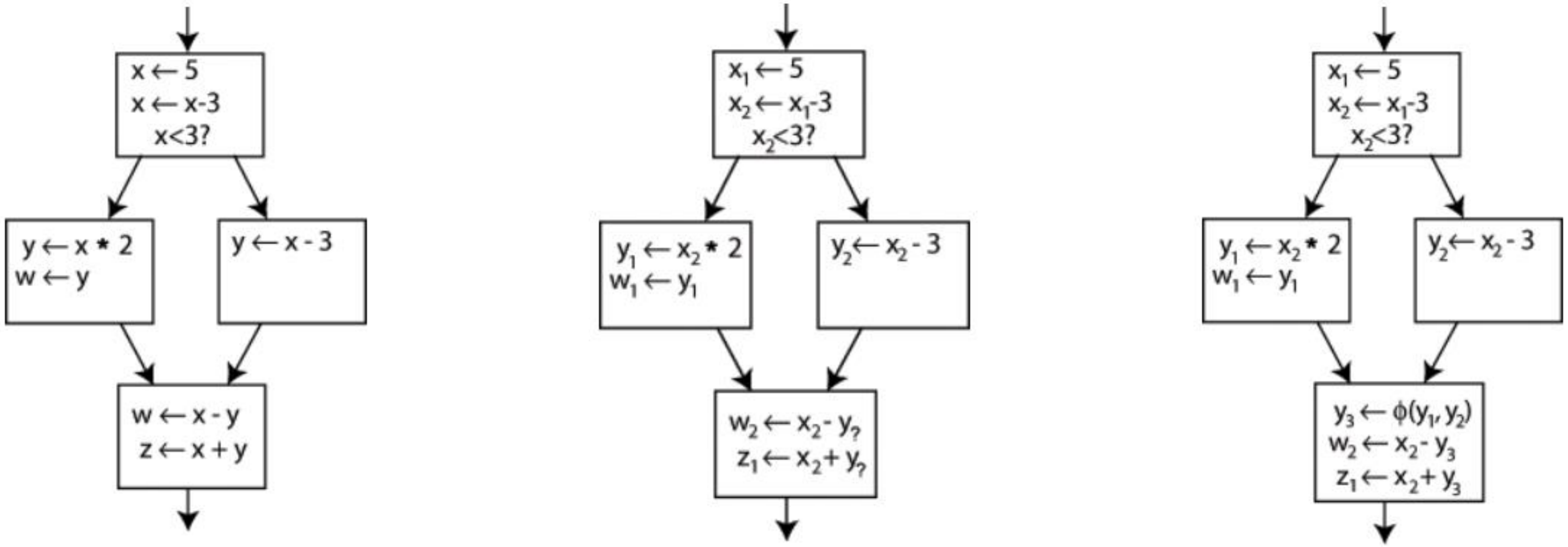
Static Single Assignment form (often abbreviated as **SSA form** or simply **SSA**) is a property of an intermediate representation (IR), which requires that each variable is assigned exactly once, and every variable is defined before it is used. Existing variables in the original IR are split into *versions*, 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. (by wikipedia.org)

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.

$$
\begin{aligned}
y_1 &:= 1 \\
y_2 &:= 2 \\
x_1 &:= y_2
\end{aligned}
$$

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

arm

SSA Value

cmd/compile/internal/ssa/value.go

```go
type Value struct {
	// A unique identifier for the value. For performance we allocate these IDs
	// densely starting at 1.  There is no guarantee that there won't be occasional holes, though.
	ID ID

	// The operation that computes this value. See op.go.
	Op Op

	// The type of this value. Normally this will be a Go type, but there
	// are a few other pseudo-types, see type.go.
	Type *types.Type

	// Auxiliary info for this value. The type of this information depends on the opcode and type.
	// AuxInt is used for integer values, Aux is used for other values.
	// Floats are stored in AuxInt using math.Float64bits(f).
	AuxInt int64
	Aux    interface{}

	// Arguments of this value
	Args []*Value

	// Containing basic block
	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

```
//     kind           control    successors
//  ------------------------------------------
//     Exit          return mem                []
//    Plain                 nil            [next]
//       If     a boolean Value      [then, else]
//    Defer                 mem  [nopanic, panic]
type BlockKind int8
```

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

```
// Block represents a basic block in the control flow graph of a function.
type Block struct {
    // A unique identifier for the block. The system will attempt to allocate
    // these IDs densely, but no guarantees.
    ID ID

    // Source position for block's control operation
    Pos src.XPos

    // The kind of block this is.
    Kind BlockKind

    // Likely direction for branches.
    // If BranchLikely, Succs[0] is the most likely branch taken.
    // If BranchUnlikely, Succs[1] is the most likely branch taken.
    // Ignored if len(Succs) < 2.
    // Fatal if not BranchUnknown and len(Succs) > 2.
    Likely BranchPrediction

    // After flagalloc, records whether flags are live at the end of the block.
    FlagsLiveAtEnd bool

    // Subsequent blocks, if any. The number and order depend on the block kind.
    Succs []Edge

    // Inverse of successors.
    // The order is significant to Phi nodes in the block.
    // TODO: predecessors is a pain to maintain. Can we somehow order phi
    // arguments by block id and have this field computed explicitly when needed?
    Preds []Edge

    // A value that determines how the block is exited. Its value depends on the kind
    // of the block. For instance, a BlockIf has a boolean control value and BlockExit
    // has a memory control value.
    Control *Value

    // Auxiliary info for the block. Its value depends on the Kind.
    Aux interface{}

    // The unordered set of Values that define the operation of this block.
    // The list must include the control value, if any. (TODO: need this last condition?)
    // After the scheduling pass, this list is ordered.
    Values []*Value
```


SSA Example

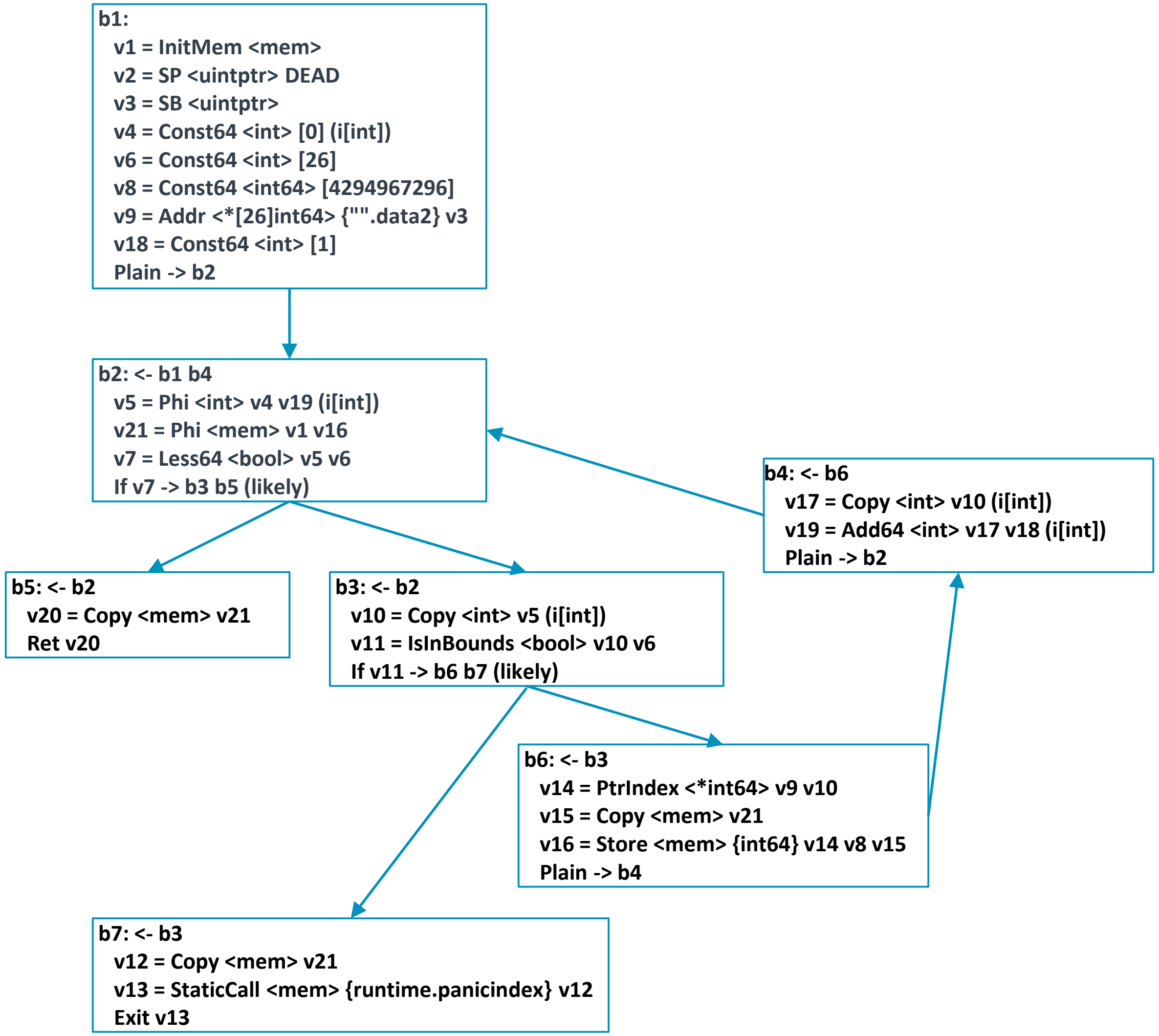
```
var data2 [26]int64

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

Dump by setting $GOSSAFUNC:

$ export GOSSAFUNC="Init“

A side effect of GOSSAFUNC=foo is creation of ssa.html in the directory containing the source for function/method: foo. (by [David Chase](#))

```
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]
```



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

```
// list of passes for the compiler
var passes = [...]pass{
	// TODO: combine phielim and copyelim into a single pass?
	{name: "early phielim", fn: phielim},
	{name: "early copyelim", fn: copyelim},
	{name: "early deadcode", fn: deadcode}, // remove generated dead code to avoid doing pointless work during opt
	{name: "short circuit", fn: shortcircuit},
	{name: "decompose user", fn: decomposeUser, required: true},
	{name: "opt", fn: opt, required: true},               // TODO: split required rules and optimizing rules
	{name: "zero arg cse", fn: zcse, required: true},     // required to merge OpSB values
	{name: "opt deadcode", fn: deadcode, required: true}, // remove any blocks orphaned during opt
	{name: "generic cse", fn: cse},
	{name: "phiopt", fn: phiopt},
	{name: "nilcheckelim", fn: nilcheckelim},
	{name: "prove", fn: prove},
	{name: "loopbce", fn: loopbce},
	{name: "decompose builtin", fn: decomposeBuiltIn, required: true},
	{name: "softfloat", fn: softfloat, required: true},
	{name: "late opt", fn: opt, required: true}, // TODO: split required rules and optimizing rules
	{name: "generic deadcode", fn: deadcode},
	{name: "check bce", fn: checkbce},
	{name: "fuse", fn: fuse},
	{name: "dse", fn: dse},
	{name: "writebarrier", fn: writebarrier, required: true}, // expand write barrier ops
	{name: "insert resched checks", fn: insertLoopReschedChecks,
		disabled: objabi.Preemptibleloops_enabled == 0}, // insert resched checks in loops.
	{name: "tighten", fn: tighten}, // move values closer to their uses
```

Options providing insight into compiler decisions:

- -d=ssa/<phase>/stats
- -m

Machine-independent Optimization Example

Before	After

b1:
  v1 = InitMem <mem>
  v2 = SP <uintptr>
  v5 = Addr <*uint64> {~r1} v2
  v6 = Arg <uint64> {a}
  v8 = Const64 <uint64> [7]
  v9 = Div64u <uint64> v6 v8
  v10 = VarDef <mem> {~r1} v1
  v11 = Store <mem> {uint64} v5 v9 v10
  Ret v11
 	
func Div(a uint64) uint64 {
    return a/7;
}
	b1:
	  v1 = InitMem <mem>
	  v2 = SP <uintptr>
	  v5 = Addr <*uint64> {~r1} v2
	  v6 = Arg <uint64> {a}
	  v8 = Const64 <uint64> [7] DEAD
	  v10 = VarDef <mem> {~r1} v1
	  v3 = Const64 <uint64> [2635249153387078803]
	  v12 = Const64 <uint64> [2]
	  v4 = Hmul64u <uint64> v3 v6
	  v7 = Avg64u <uint64> v6 v4
	  v9 = Rsh64Ux64 <uint64> v7 v12
	  v11 = Store <mem> {uint64} v5 v9 v10
	  Ret v11

Optimization Mathematical Background

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

```
  ⌊x / c⌋ = ⌊x * (2^e/c) / 2^e⌋.
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, with s = ⌈log2(c)⌉.

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.
  ⌊x / c⌋ = ⌊x * m / 2^(n+s)⌋
  ⌊x / c⌋ = ⌊x * (m/2) / 2^(n+s-1)⌋
  ⌊x / c⌋ = ⌊x * (m/2) / 2^n / 2^(s-1)⌋
  ⌊x / c⌋ = ⌊⌊x * (m/2) / 2^n⌋ / 2^(s-1)⌋
  multiply + shift

Case 2: c is even.
  ⌊x / c⌋ = ⌊(x/2) / (c/2)⌋
  ⌊x / c⌋ = ⌊⌊x/2⌋ / (c/2)⌋
    This is just the original problem, with x' = ⌊x/2⌋, c' = c/2, n' = n-1.
      s' = s-1
      m' = ⌈2^(n'+s')/c'⌉
         = ⌈2^(n+s-1)/c⌉
         = ⌈m/2⌉
  ⌊x / c⌋ = ⌊x' * m' / 2^(n'+s')⌋
  ⌊x / c⌋ = ⌊⌊x/2⌋ * ⌈m/2⌉ / 2^(n+s-2)⌋
  ⌊x / c⌋ = ⌊⌊⌊x/2⌋ * ⌈m/2⌉ / 2^n⌋ / 2^(s-2)⌋
  shift + multiply + shift

Case 3: everything else
  let k = m - 2^n. k fits in n bits.
  ⌊x / c⌋ = ⌊x * m / 2^(n+s)⌋
  ⌊x / c⌋ = ⌊x * (2^n + k) / 2^(n+s)⌋
  ⌊x / c⌋ = ⌊(x + x * k / 2^n) / 2^s⌋
  ⌊x / c⌋ = ⌊(x + ⌊x * k / 2^n⌋) / 2^s⌋
  ⌊x / c⌋ = ⌊(x + ⌊x * k / 2^n⌋) / 2^s⌋
  ⌊x / c⌋ = ⌊⌊(x + ⌊x * k / 2^n⌋) / 2⌋ / 2^(s-1)⌋
  multiply + avg + shift
```

Generic SSA Rules Example

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

```
// For 64-bit divides on 64-bit machines
// (64-bit divides on 32-bit machines are lowered to a runtime call by the walk pass.)
(Div64u x (Const64 [c])) && umagicOK(64, c) && config.RegSize == 8 && umagic(64,c).m&1 == 0 ->
  (Rsh64Ux64 <typ.UInt64>
    (Hmul64u <typ.UInt64>
      (Const64 <typ.UInt64> [int64(1<<63+umagic(64,c).m/2)])
      x)
    (Const64 <typ.UInt64> [umagic(64,c).s-1]))
(Div64u x (Const64 [c])) && umagicOK(64, c) && config.RegSize == 8 && c&1 == 0 ->
  (Rsh64Ux64 <typ.UInt64>
    (Hmul64u <typ.UInt64>
      (Const64 <typ.UInt64> [int64(1<<63+(umagic(64,c).m+1)/2)])
      (Rsh64Ux64 <typ.UInt64> x (Const64 <typ.UInt64> [1])))
    (Const64 <typ.UInt64> [umagic(64,c).s-2]))
(Div64u x (Const64 [c])) && umagicOK(64, c) && config.RegSize == 8 ->
  (Rsh64Ux64 <typ.UInt64>
    (Avg64u
      x
      (Hmul64u <typ.UInt64>
        (Const64 <typ.UInt64> [int64(umagic(64,c).m)])
        x))
    (Const64 <typ.UInt64> [umagic(64,c).s-1]))

```

```
Case 1: m is even.
  ⌊x / c⌋ = ⌊x * m / 2^(n+s)⌋
  ⌊x / c⌋ = ⌊x * (m/2) / 2^(n+s-1)⌋
  ⌊x / c⌋ = ⌊x * (m/2) / 2^n / 2^(s-1)⌋
  ⌊x / c⌋ = ⌊⌊x * (m/2) / 2^n⌋ / 2^(s-1)⌋
  multiply + shift

Case 2: c is even.
  ⌊x / c⌋ = ⌊(x/2) / (c/2)⌋
  ⌊x / c⌋ = ⌊⌊x/2⌋ / (c/2)⌋
    This is just the original problem, with x' = ⌊x/2⌋, c' = c/2, n' = n-1.
      s' = s-1
      m' = ⌈2^(n'+s')/c'⌉
         = ⌈2^(n+s-1)/c⌉
         = ⌈m/2⌉
  ⌊x / c⌋ = ⌊x' * m' / 2^(n'+s')⌋
  ⌊x / c⌋ = ⌊⌊x/2⌋ * ⌈m/2⌉ / 2^(n+s-2)⌋
  ⌊x / c⌋ = ⌊⌊⌊x/2⌋ * ⌈m/2⌉ / 2^n⌋ / 2^(s-2)⌋
  shift + multiply + shift

Case 3: everything else
  let k = m - 2^n. k fits in n bits.
  ⌊x / c⌋ = ⌊x * m / 2^(n+s)⌋
  ⌊x / c⌋ = ⌊x * (2^n + k) / 2^(n+s)⌋
  ⌊x / c⌋ = ⌊(x + x * k / 2^n) / 2^s⌋
  ⌊x / c⌋ = ⌊(x + ⌊x * k / 2^n⌋) / 2^s⌋
  ⌊x / c⌋ = ⌊(x + ⌊x * k / 2^n⌋) / 2^s⌋
  ⌊x / c⌋ = ⌊⌊(x + ⌊x * k / 2^n⌋) / 2⌋ / 2^(s-1)⌋
  multiply + avg + shift
```


Machine-independent Optimization Example

Before

```
b1:
  v1 = InitMem <mem>
  v2 = SP <uintptr>
  v5 = Addr <*int64> {~r1} v2
  v6 = Arg <int64> {a}
  v8 = Const64 <int64> [7]
  v9 = Div64 <int64> v6 v8
  v10 = VarDef <mem> {~r1} v1
  v11 = Store <mem> {int64} v5 v9 v10
  Ret v11
```

```
func Div(a int64) int64 {
    return a/7;
}
```

After

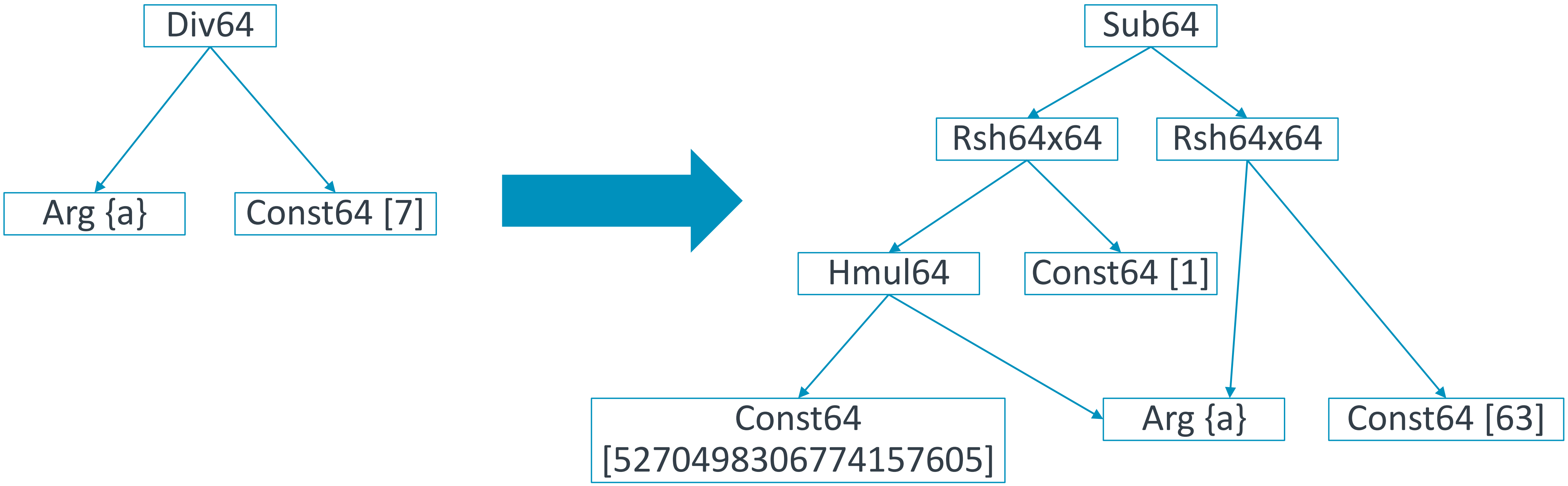
```
b1:
  v1 = InitMem <mem>
  v2 = SP <uintptr>
  v5 = Addr <*int64> {~r1} v2
  v6 = Arg <int64> {a}
  v8 = Const64 <int64> [7] DEAD
  v10 = VarDef <mem> {~r1} v1
  v3 = Const64 <uint64> [5270498306774157605]
  v12 = Const64 <uint64> [1]
  v14 = Const64 <uint64> [63]
  v4 = Hmul64 <int64> v3 v6
  v13 = Rsh64x64 <int64> v6 v14
  v7 = Rsh64x64 <int64> v4 v12
  v9 = Sub64 <int64> v7 v13
  v11 = Store <mem> {int64} v5 v9 v10
  Ret v11
```

Generic SSA Rules Implementation Example

```
(Div64 <t> x (Const64 [c])) && smagicOK(64,c) && smagic(64,c).m&1 == 0 ->
 (Sub64 <t>
  (Rsh64x64 <t>
   (Hmul64 <t>
    (Const64 <typ.UInt64> [int64(smagic(64,c).m/2)])
    x)
   (Const64 <typ.UInt64> [smagic(64,c).s-1]))
  (Rsh64x64 <t>
   x
   (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

```
{name: "lower", fn: lower, required: true},
{name: "lowered cse", fn: cse},
{name: "elim unread autos", fn: elimUnreadAutos},
{name: "lowered deadcode", fn: deadcode, required: true},
{name: "checkLower", fn: checkLower, required: true},
{name: "late phielim", fn: phielim},
{name: "late copyelim", fn: copyelim},
{name: "phi tighten", fn: phiTighten},
{name: "late deadcode", fn: deadcode},
{name: "critical", fn: critical, required: true}, // remove critical edges
{name: "likelyadjust", fn: likelyadjust},
{name: "layout", fn: layout, required: true},      // schedule blocks
{name: "schedule", fn: schedule, required: true}, // schedule values
{name: "late nilcheck", fn: nilcheckelim2},
{name: "flagalloc", fn: flagalloc, required: true}, // allocate flags register
{name: "regalloc", fn: regalloc, required: true},   // allocate int & float registers + stack slots
{name: "loop rotate", fn: loopRotate},
{name: "stackframe", fn: stackframe, required: true},
{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, unsighed
        {name: "DIVW", argLength: 2, reg: gp21, asm: "SDIVW"},                      // arg0 / arg1, signed, 32 bit
        {name: "UDIVW", argLength: 2, reg: gp21, asm: "UDIVW"},                     // arg0 / arg1, unsighed, 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
Div <T>	Div <T>
b1:	b1:
v1 = InitMem <mem>	v1 = InitMem <mem>
v2 = SP <uintptr>	v2 = SP <uintptr>
v5 = Addr <*int64> {~r1} v2	v5 = MOVDaddr <*int64> {~r1} v2 DEAD
v6 = Arg <int64> {a}	v6 = Arg <int64> {a}
v10 = VarDef <mem> {~r1} v1	v10 = VarDef <mem> {~r1} v1
v3 = Const64 <uint64> [5270498306774157605]	v3 = MOVDconst <uint64> [5270498306774157605]
v12 = Const64 <uint64> [1]	v12 = MOVDconst <uint64> [1] DEAD
v14 = Const64 <uint64> [63]	v13 = SRAconst <int64> [63] v6 DEAD
v4 = Hmul64 <int64> v3 v6	v14 = MOVDconst <uint64> [63] DEAD
v13 = Rsh64x64 <int64> v6 v14	v15 = MOVDconst <uint64> [63] DEAD
v7 = Rsh64x64 <int64> v4 v12	v16 = FlagLT_ULT <flags> DEAD
v9 = Sub64 <int64> v7 v13	v18 = MOVDconst <uint64> [63] DEAD
v11 = Store <mem> {int64} v5 v9 v10	v19 = FlagLT_ULT <flags> DEAD
Ret v11	v4 = MULH <int64> v3 v6
	v7 = SRAconst <int64> [1] v4
	v9 = SUBshiftRA <int64> [63] v7 v6
	v11 = MOVDstore <mem> {~r1} v2 v9 v10
	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>	662 Div <T>
649 b1:	663 b1:
650 v1 = InitMem <mem>	664 v1 = InitMem <mem>
651 v10 = VarDef <mem> {~r1} v1	665 v10 = VarDef <mem> {~r1} v1
652 v2 = SP <uintptr>	666 v2 = SP <uintptr> : SP
653 v6 = Arg <int64> {a}	667 v6 = Arg <int64> {a} : a[int64]
654 v3 = MOVDconst <uint64> [5270498306774157605]	668 v19 = MOVDconst <uint64> [5270498306774157605] : R0
655 v4 = MULH <int64> v3 v6	669 v18 = LoadReg <int64> v6 : R1
656 v7 = SRAconst <int64> [1] v4	670 v4 = MULH <int64> v19 v18 : R0
657 v9 = SUBshiftRA <int64> [63] v7 v6	671 v7 = SRAconst <int64> [1] v4 : R0
658 v11 = MOVDstore <mem> {~r1} v2 v9 v10	672 v9 = SUBshiftRA <int64> [63] v7 v18 : R0
659 Ret v11	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> 708 b1: 709 v1 = InitMem <mem> 710 v10 = VarDef <mem> {~r1} v1 711 v2 = SP <uintptr> : SP 712 v6 = Arg <int64> {a} : a[int64] 713 v19 = MOVDconst <uint64> [5270498306774157605] : R0 714 v18 = LoadReg <int64> v6 : R1 715 v4 = MULH <int64> v19 v18 : R0 716 v7 = SRAconst <int64> [1] v4 : R0 717 v9 = SUBshiftRA <int64> [63] v7 v18 : R0 718 v11 = MOVDstore <mem> {~r1} v2 v9 v10 719 Ret v11	722 00000 (3) TEXT "".Div(SB) …… 725 v19 00003 (4) MOVD $5270498306774157605, R0 726 v18 00004 (4) MOVD "".a(RSP), R1 727 v4 00005 (4) SMULH R1, R0, R0 728 v7 00006 (4) ASR $1, R0, R0 729 v9 00007 (4) SUB R1->63, R0, R0 730 v11 00008 (4) MOVD R0, "".~r1+8(RSP) 731 b1 00009 (4) RET 732 00010 (?) END

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

package test
 	
import "math"
 	
func MySqrt(x float64) float64 {
    return math.Sqrt(x)
}
	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
	……
		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
		……

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

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

```
addF("math/bits", "TrailingZeros16",
    func(s *state, n *Node, args []*ssa.Value) *ssa.Value {
        x := s.newValue1(ssa.OpZeroExt16to64, types.Types[TUINT64], args[0])
        c := s.constInt64(types.Types[TUINT64], 1<<16)
        y := s.newValue2(ssa.OpOr64, types.Types[TUINT64], x, c)
        return s.newValue1(ssa.OpCtz64, types.Types[TINT], y)
    },
    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

Acknowledgements

Thank everyone involved in the review of the slides, especially following viewers from Golang-dev forum who provide valuable comments and feedbacks:

Brad Fitzpatrick bradfitz@golang.org

David Chase drchase@google.com

Robert Griesemer gri@golang.org

ilya.tocar@intel.com

Thank You!

Danke!

Merci!

谢谢!

ありがとう!

Gracias!

Kiitos!

감사합니다

ड़ीँ क्रड़