Go internal ABI specification

Self-link: go.dev/s/regabi

This document describes Go's internal application binary interface (ABI), known as ABIInternal. Go's ABI defines the layout of data in memory and the conventions for calling between Go functions. This ABI is *unstable* and will change between Go versions. If you're writing assembly code, please instead refer to Go's assembly documentation, which describes Go's stable ABI, known as ABI0.

All functions defined in Go source follow ABIInternal. However, ABIInternal and ABI0 functions are able to call each other through transparent *ABI wrappers*, described in the internal calling convention proposal.

Go uses a common ABI design across all architectures. We first describe the common ABI, and then cover per-architecture specifics.

Rationale: For the reasoning behind using a common ABI across architectures instead of the platform ABI, see the register-based Go calling convention proposal.

Memory layout

Go's built-in types have the following sizes and alignments. Many, though not all, of these sizes are guaranteed by the language specification. Those that aren't guaranteed may change in future versions of Go (for example, we've considered changing the alignment of int64 on 32-bit).

Type	64-bit		32-bit	
	Size	Align	Size	Align
bool, uint8, int8	1	1	1	1
uint16, int16	2	2	2	2
uint32, int32	4	4	4	4
uint64, int64	8	8	8	4
int, uint	8	8	4	4
float32	4	4	4	4
float64	8	8	8	4
complex64	8	4	8	4
complex128	16	8	16	4
uintptr, *T, unsafe.Pointer	8	8	4	4

The types `byte` and `rune` are aliases for `uint8` and `int32`, respectively, and hence have the same size and alignment as these types.

The layout of `map`, `chan`, and `func` types is equivalent to *T.

To describe the layout of the remaining composite types, we first define the layout of a *sequence* S of N fields with types t_1, t_2, ..., t_N. We define the byte offset at which each field begins relative to a base address of 0, as well as the size and alignment of the sequence as follows:

```
offset(S, i) = 0  if i = 1
             = align(offset(S, i-1) + sizeof(t_(i-1)), alignof(t_i))
alignof(S)   = 1  if N = 0
             = max(alignof(t_i) | 1 <= i <= N)
sizeof(S)    = 0  if N = 0
             = align(offset(S, N) + sizeof(t_N), alignof(S))
```

Where sizeof(T) and alignof(T) are the size and alignment of type T, respectively, and align(x, y) rounds x up to a multiple of y.

The `interface{}` type is a sequence of 1. a pointer to the runtime type description for the interface's dynamic type and 2. an `unsafe.Pointer` data field. Any other interface type (besides the empty interface) is a sequence of 1. a pointer to the runtime "itab" that gives the method pointers and the type of the data field and 2. an `unsafe.Pointer` data field. An interface can be "direct" or "indirect" depending on the dynamic type: a direct interface stores the value directly in the data field, and an indirect interface stores a pointer to the value in the data field. An interface can only be direct if the value consists of a single pointer word.

An array type `[N]T` is a sequence of N fields of type T.

The slice type `[]T` is a sequence of a `*[cap]T` pointer to the slice backing store, an `int` giving the `len` of the slice, and an `int` giving the `cap` of the slice.

The `string` type is a sequence of a `*[len]byte` pointer to the string backing store, and an `int` giving the `len` of the string.

A struct type `struct { f1 t1; ...; fM tM }` is laid out as the sequence t1, ..., tM, tP, where tP is either:

- Type `byte` if sizeof(tM) = 0 and any of sizeof(t*i*) ≠ 0.
- Empty (size 0 and align 1) otherwise.

The padding byte prevents creating a past-the-end pointer by taking the address of the final, empty fN field.

Note that user-written assembly code should generally not depend on Go type layout and should instead use the constants defined in `go_asm.h`.

Function call argument and result passing

Function calls pass arguments and results using a combination of the stack and machine registers. Each argument or result is passed either entirely in registers or entirely on the stack. Because access to registers is generally faster than access to the stack, arguments and results are preferentially passed in registers. However, any argument or result that contains a non-trivial array or does not fit entirely in the remaining available registers is passed on the stack.

Each architecture defines a sequence of integer registers and a sequence of floating-point registers. At a high level, arguments and results are recursively broken down into values of base types and these base values are assigned to registers from these sequences.

Arguments and results can share the same registers, but do not share the same stack space. Beyond the arguments and results passed on the stack, the caller also reserves spill space on the stack for all register-based arguments (but does not populate this space).

The receiver, arguments, and results of function or method F are assigned to registers or the stack using the following algorithm:

1. Let NI and NFP be the length of integer and floating-point register sequences defined by the architecture. Let I and FP be 0; these are the indexes of the next integer and floating-pointer register. Let S, the type sequence defining the stack frame, be empty.
2. If F is a method, assign F's receiver.
3. For each argument A of F, assign A.
4. Add a pointer-alignment field to S. This has size 0 and the same alignment as `uintptr`.
5. Reset I and FP to 0.
6. For each result R of F, assign R.
7. Add a pointer-alignment field to S.
8. For each register-assigned receiver and argument of F, let T be its type and add T to the stack sequence S. This is the argument's (or receiver's) spill space and will be uninitialized at the call.
9. Add a pointer-alignment field to S.

Assigning a receiver, argument, or result V of underlying type T works as follows:

1. Remember I and FP.
2. If T has zero size, add T to the stack sequence S and return.
3. Try to register-assign V.
4. If step 3 failed, reset I and FP to the values from step 1, add T to the stack sequence S, and assign V to this field in S.

Register-assignment of a value V of underlying type T works as follows:

1. If T is a boolean or integral type that fits in an integer register, assign V to register I and increment I.
2. If T is an integral type that fits in two integer registers, assign the least significant and most significant halves of V to registers I and I+1, respectively, and increment I by 2
3. If T is a floating-point type and can be represented without loss of precision in a floating-point register, assign V to register FP and increment FP.
4. If T is a complex type, recursively register-assign its real and imaginary parts.
5. If T is a pointer type, map type, channel type, or function type, assign V to register I and increment I.
6. If T is a string type, interface type, or slice type, recursively register-assign V's components (2 for strings and interfaces, 3 for slices).
7. If T is a struct type, recursively register-assign each field of V.
8. If T is an array type of length 0, do nothing.
9. If T is an array type of length 1, recursively register-assign its one element.
10. If T is an array type of length > 1, fail.
11. If I > NI or FP > NFP, fail.
12. If any recursive assignment above fails, fail.

The above algorithm produces an assignment of each receiver, argument, and result to registers or to a field in the stack sequence. The final stack sequence looks like: stack-assigned receiver, stack-assigned arguments, pointer-alignment, stack-assigned results, pointer-alignment, spill space for each register-assigned argument, pointer-alignment. The following diagram shows what this stack frame looks like on the stack, using the typical convention where address 0 is at the bottom:

```
+------------------------------+
|             . . .            |
| 2nd reg argument spill space |
| 1st reg argument spill space |
| <pointer-sized alignment>    |
|             . . .            |
| 2nd stack-assigned result    |
| 1st stack-assigned result    |
```

```
| <pointer-sized alignment>    |
|             . . .            |
| 2nd stack-assigned argument  |
| 1st stack-assigned argument  |
| stack-assigned receiver      |
+------------------------------+ ↓ lower addresses
```

To perform a call, the caller reserves space starting at the lowest address in its stack frame for the call stack frame, stores arguments in the registers and argument stack fields determined by the above algorithm, and performs the call. At the time of a call, spill space, result stack fields, and result registers are left uninitialized. Upon return, the callee must have stored results to all result registers and result stack fields determined by the above algorithm.

There are no callee-save registers, so a call may overwrite any register that doesn't have a fixed meaning, including argument registers.

Example

Consider the function `func f(a1 uint8, a2 [2]uintptr, a3 uint8) (r1 struct { x uintptr; y [2]uintptr }, r2 string)` on a 64-bit architecture with hypothetical integer registers R0–R9.

On entry, `a1` is assigned to `R0`, `a3` is assigned to `R1` and the stack frame is laid out in the following sequence:

```
a2      [2]uintptr
r1.x    uintptr
r1.y    [2]uintptr
a1Spill uint8
a3Spill uint8
_       [6]uint8  // alignment padding
```

In the stack frame, only the `a2` field is initialized on entry; the rest of the frame is left uninitialized.

On exit, `r2.base` is assigned to `R0`, `r2.len` is assigned to `R1`, and `r1.x` and `r1.y` are initialized in the stack frame.

There are several things to note in this example. First, `a2` and `r1` are stack-assigned because they contain arrays. The other arguments and results are register-assigned. Result `r2` is decomposed into its components, which are individually register-assigned. On the stack, the stack-assigned arguments appear at lower addresses than the stack-assigned results, which appear at lower addresses than the argument spill area. Only arguments, not results, are assigned a spill area on the stack.

Rationale

Each base value is assigned to its own register to optimize construction and access. An alternative would be to pack multiple sub-word values into registers, or to simply map an argument's in-memory layout to registers (this is common in C ABIs), but this typically adds cost to pack and unpack these values. Modern architectures have more than enough registers to pass all arguments and results this way for nearly all functions (see the appendix), so there's little downside to spreading base values across registers.

Arguments that can't be fully assigned to registers are passed entirely on the stack in case the callee takes the address of that argument. If an argument could be split across the stack and registers and the callee took its address, it would need to be reconstructed in memory, a process that would be proportional to the size of the argument.

Non-trivial arrays are always passed on the stack because indexing into an array typically requires a computed offset, which generally isn't possible with registers. Arrays in general are rare in function signatures (only 0.7% of functions

in the Go 1.15 standard library and 0.2% in kubelet). We considered allowing array fields to be passed on the stack while the rest of an argument's fields are passed in registers, but this creates the same problems as other large structs if the callee takes the address of an argument, and would benefit <0.1% of functions in kubelet (and even these very little).

We make exceptions for 0 and 1-element arrays because these don't require computed offsets, and 1-element arrays are already decomposed in the compiler's SSA representation.

The ABI assignment algorithm above is equivalent to Go's stack-based ABI0 calling convention if there are zero architecture registers. This is intended to ease the transition to the register-based internal ABI and make it easy for the compiler to generate either calling convention. An architecture may still define register meanings that aren't compatible with ABI0, but these differences should be easy to account for in the compiler.

The assignment algorithm assigns zero-sized values to the stack (assignment step 2) in order to support ABI0-equivalence. While these values take no space themselves, they do result in alignment padding on the stack in ABI0. Without this step, the internal ABI would register-assign zero-sized values even on architectures that provide no argument registers because they don't consume any registers, and hence not add alignment padding to the stack.

The algorithm reserves spill space for arguments in the caller's frame so that the compiler can generate a stack growth path that spills into this reserved space. If the callee has to grow the stack, it may not be able to reserve enough additional stack space in its own frame to spill these, which is why it's important that the caller do so. These slots also act as the home location if these arguments need to be spilled for any other reason, which simplifies traceback printing.

There are several options for how to lay out the argument spill space. We chose to lay out each argument according to its type's usual memory layout but to separate the spill space from the regular argument space. Using the usual memory layout simplifies the compiler because it already understands this layout. Also, if a function takes the address of a register-assigned argument, the compiler must spill that argument to memory in its usual memory layout and it's more convenient to use the argument spill space for this purpose.

Alternatively, the spill space could be structured around argument registers. In this approach, the stack growth spill path would spill each argument register to a register-sized stack word. However, if the function takes the address of a register-assigned argument, the compiler would have to reconstruct it in memory layout elsewhere on the stack.

The spill space could also be interleaved with the stack-assigned arguments so the arguments appear in order whether they are register- or stack-assigned. This would be close to ABI0, except that register-assigned arguments would be uninitialized on the stack and there's no need to reserve stack space for register-assigned results. We expect separating the spill space to perform better because of memory locality. Separating the space is also potentially simpler for `reflect` calls because this allows `reflect` to summarize the spill space as a single number. Finally, the long-term intent is to remove reserved spill slots entirely – allowing most functions to be called without any stack setup and easing the introduction of callee-save registers – and separating the spill space makes that transition easier.

Closures

A func value (e.g., `var x func()`) is a pointer to a closure object. A closure object begins with a pointer-sized program counter representing the entry point of the function, followed by zero or more bytes containing the closed-over environment.

Closure calls follow the same conventions as static function and method calls, with one addition. Each architecture specifies a *closure context pointer* register and calls to closures store the address of the closure object in the closure context pointer register prior to the call.

Software floating-point mode

In "softfloat" mode, the ABI simply treats the hardware as having zero floating-point registers. As a result, any arguments containing floating-point values will be passed on the stack.

Rationale: Softfloat mode is about compatibility over performance and is not commonly used. Hence, we keep the ABI as simple as possible in this case, rather than adding additional rules for passing floating-point values in integer registers.

Architecture specifics

This section describes per-architecture register mappings, as well as other per-architecture special cases.

amd64 architecture

The amd64 architecture uses the following sequence of 9 registers for integer arguments and results:

```
RAX, RBX, RCX, RDI, RSI, R8, R9, R10, R11
```

It uses X0 – X14 for floating-point arguments and results.

Rationale: These sequences are chosen from the available registers to be relatively easy to remember.

Registers R12 and R13 are permanent scratch registers. R15 is a scratch register except in dynamically linked binaries.

Rationale: Some operations such as stack growth and reflection calls need dedicated scratch registers in order to manipulate call frames without corrupting arguments or results.

Special-purpose registers are as follows:

Register	Call meaning	Return meaning	Body meaning
RSP	Stack pointer	Same	Same
RBP	Frame pointer	Same	Same
RDX	Closure context pointer	Scratch	Scratch
R12	Scratch	Scratch	Scratch
R13	Scratch	Scratch	Scratch
R14	Current goroutine	Same	Same
R15	GOT reference temporary if dynlink	Same	Same
X15	Zero value (*)	Same	Scratch

(*) Except on Plan 9, where X15 is a scratch register because SSE registers cannot be used in note handlers (so the compiler avoids using them except when absolutely necessary).

Rationale: These register meanings are compatible with Go's stack-based calling convention except for R14 and X15, which will have to be restored on transitions from ABI0 code to ABIInternal code. In ABI0, these are undefined, so transitions from ABIInternal to ABI0 can ignore these registers.

Rationale: For the current goroutine pointer, we chose a register that requires an additional REX byte. While this adds one byte to every function prologue, it is hardly ever accessed outside the function prologue and we expect making more single-byte registers available to be a net win.

Rationale: We could allow R14 (the current goroutine pointer) to be a scratch register in function bodies because it can always be restored from TLS on amd64. However, we designate it as a fixed register for simplicity and for consistency with other architectures that may not have a copy of the current goroutine pointer in TLS.

Rationale: We designate X15 as a fixed zero register because functions often have to bulk zero their stack frames, and this is more efficient with a designated zero register.

Implementation note: Registers with fixed meaning at calls but not in function bodies must be initialized by "injected" calls such as signal-based panics.

Stack layout

The stack pointer, RSP, grows down and is always aligned to 8 bytes.

The amd64 architecture does not use a link register.

A function's stack frame is laid out as follows:

```
+------------------------------+
| return PC                    |
| RBP on entry                 |
| ... locals ...               |
| ... outgoing arguments ...   |
+------------------------------+ ↓ lower addresses
```

The "return PC" is pushed as part of the standard amd64 `CALL` operation. On entry, a function subtracts from RSP to open its stack frame and saves the value of RBP directly below the return PC. A leaf function that does not require any stack space may omit the saved RBP.

The Go ABI's use of RBP as a frame pointer register is compatible with amd64 platform conventions so that Go can inter-operate with platform debuggers and profilers.

Flags

The direction flag (D) is always cleared (set to the “forward” direction) at a call. The arithmetic status flags are treated like scratch registers and not preserved across calls. All other bits in RFLAGS are system flags.

At function calls and returns, the CPU is in x87 mode (not MMX technology mode).

Rationale: Go on amd64 does not use either the x87 registers or MMX registers. Hence, we follow the SysV platform conventions in order to simplify transitions to and from the C ABI.

At calls, the MXCSR control bits are always set as follows:

Flag	Bit	Value	Meaning
FZ	15	0	Do not flush to zero
RC	14/13	0 (RN)	Round to nearest
PM	12	1	Precision masked
UM	11	1	Underflow masked

OM	10	1	Overflow masked
ZM	9	1	Divide-by-zero masked
DM	8	1	Denormal operations masked
IM	7	1	Invalid operations masked
DAZ	6	0	Do not zero de-normals

The MXCSR status bits are callee-save.

Rationale: Having a fixed MXCSR control configuration allows Go functions to use SSE operations without modifying or saving the MXCSR. Functions are allowed to modify it between calls (as long as they restore it), but as of this writing Go code never does. The above fixed configuration matches the process initialization control bits specified by the ELF AMD64 ABI.

The x87 floating-point control word is not used by Go on amd64.

arm64 architecture

The arm64 architecture uses R0 – R15 for integer arguments and results.

It uses F0 – F15 for floating-point arguments and results.

Rationale: 16 integer registers and 16 floating-point registers are more than enough for passing arguments and results for practically all functions (see Appendix). While there are more registers available, using more registers provides little benefit. Additionally, it will add overhead on code paths where the number of arguments are not statically known (e.g. reflect call), and will consume more stack space when there is only limited stack space available to fit in the nosplit limit.

Registers R16 and R17 are permanent scratch registers. They are also used as scratch registers by the linker (Go linker and external linker) in trampolines.

Register R18 is reserved and never used. It is reserved for the OS on some platforms (e.g. macOS).

Registers R19 – R25 are permanent scratch registers. In addition, R27 is a permanent scratch register used by the assembler when expanding instructions.

Floating-point registers F16 – F31 are also permanent scratch registers.

Special-purpose registers are as follows:

Register	Call meaning	Return meaning	Body meaning
RSP	Stack pointer	Same	Same
R30	Link register	Same	Scratch (non-leaf functions)
R29	Frame pointer	Same	Same
R28	Current goroutine	Same	Same
R27	Scratch	Scratch	Scratch
R26	Closure context pointer	Scratch	Scratch
R18	Reserved (not used)	Same	Same

ZR	Zero value	Same	Same

Rationale: These register meanings are compatible with Go's stack-based calling convention.

Rationale: The link register, R30, holds the function return address at the function entry. For functions that have frames (including most non-leaf functions), R30 is saved to stack in the function prologue and restored in the epilogue. Within the function body, R30 can be used as a scratch register.

Implementation note: Registers with fixed meaning at calls but not in function bodies must be initialized by "injected" calls such as signal-based panics.

Stack layout

The stack pointer, RSP, grows down and is always aligned to 16 bytes.

Rationale: The arm64 architecture requires the stack pointer to be 16-byte aligned.

A function's stack frame, after the frame is created, is laid out as follows:

```
+------------------------------+
| ... locals ...               |
| ... outgoing arguments ...   |
| return PC                    | ← RSP points to
| frame pointer on entry       |
+------------------------------+ ↓ lower addresses
```

The "return PC" is loaded to the link register, R30, as part of the arm64 `CALL` operation.

On entry, a function subtracts from RSP to open its stack frame, and saves the values of R30 and R29 at the bottom of the frame. Specifically, R30 is saved at 0(RSP) and R29 is saved at -8(RSP), after RSP is updated.

A leaf function that does not require any stack space may omit the saved R30 and R29.

The Go ABI's use of R29 as a frame pointer register is compatible with arm64 architecture requirement so that Go can inter-operate with platform debuggers and profilers.

This stack layout is used by both register-based (ABIInternal) and stack-based (ABI0) calling conventions.

Flags

The arithmetic status flags (NZCV) are treated like scratch registers and not preserved across calls. All other bits in PSTATE are system flags and are not modified by Go.

The floating-point status register (FPSR) is treated like scratch registers and not preserved across calls.

At calls, the floating-point control register (FPCR) bits are always set as follows:

Flag	Bit	Value	Meaning
DN	25	0	Propagate NaN operands
FZ	24	0	Do not flush to zero
RC	23/22	0 (RN)	Round to nearest, choose even if tied
IDE	15	0	Denormal operations trap disabled
IXE	12	0	Inexact trap disabled

UFE	11	0	Underflow trap disabled
OFE	10	0	Overflow trap disabled
DZE	9	0	Divide-by-zero trap disabled
IOE	8	0	Invalid operations trap disabled
NEP	2	0	Scalar operations do not affect higher elements in vector registers
AH	1	0	No alternate handling of de-normal inputs
FIZ	0	0	Do not zero de-normals

Rationale: Having a fixed FPCR control configuration allows Go functions to use floating-point and vector (SIMD) operations without modifying or saving the FPCR. Functions are allowed to modify it between calls (as long as they restore it), but as of this writing Go code never does.

ppc64 architecture

The ppc64 architecture uses R3 – R10 and R14 – R17 for integer arguments and results.

It uses F1 – F12 for floating-point arguments and results.

Register R31 is a permanent scratch register in Go.

Special-purpose registers used within Go generated code and Go assembly code are as follows:

Register	Call meaning	Return meaning	Body meaning
R0	Zero value	Same	Same
R1	Stack pointer	Same	Same
R2	TOC register	Same	Same
R11	Closure context pointer	Scratch	Scratch
R12	Function address on indirect calls	Scratch	Scratch
R13	TLS pointer	Same	Same
R20,R21	Scratch	Scratch	Used by duffcopy, duffzero
R30	Current goroutine	Same	Same
R31	Scratch	Scratch	Scratch
LR	Link register	Link register	Scratch
Rationale: These register meanings are compatible with Go's			
stack-based calling convention.			

The link register, LR, holds the function return address at the function entry and is set to the correct return address before exiting the function. It is also used in some cases as the function address when doing an indirect call.

The register R2 contains the address of the TOC (table of contents) which contains data or code addresses used when generating position independent code. Non-Go code generated when using cgo contains TOC-relative addresses which depend on R2 holding a valid TOC. Go code compiled with -shared or -dynlink initializes and maintains R2 and uses it in some cases for function calls; Go code compiled without these options does not modify R2.

When making a function call R12 contains the function address for use by the code to generate R2 at the beginning of the function. R12 can be used for other purposes within the body of the function, such as trampoline generation.

R20 and R21 are used in duffcopy and duffzero which could be generated before arguments are saved so should not be used for register arguments.

The Count register CTR can be used as the call target for some branch instructions. It holds the return address when preemption has occurred.

On PPC64 when a float32 is loaded it becomes a float64 in the register, which is different from other platforms and that needs to be recognized by the internal implementation of reflection so that float32 arguments are passed correctly.

Registers R18 - R29 and F13 - F31 are considered scratch registers.

Stack layout

The stack pointer, R1, grows down and is aligned to 8 bytes in Go, but changed to 16 bytes when calling cgo.

A function's stack frame, after the frame is created, is laid out as follows:

```
+------------------------------+
| ... locals ...               |
| ... outgoing arguments ...   |
| 24  TOC register R2 save     | When compiled with -shared/-dynlink
| 16  Unused in Go             | Not used in Go
|  8  CR save                  | nonvolatile CR fields
|  0  return PC                | ← R1 points to
+------------------------------+ ↓ lower addresses
```

The "return PC" is loaded to the link register, LR, as part of the ppc64 `BL` operations.

On entry to a non-leaf function, the stack frame size is subtracted from R1 to create its stack frame, and saves the value of LR at the bottom of the frame.

A leaf function that does not require any stack space does not modify R1 and does not save LR.

NOTE: We might need to save the frame pointer on the stack as in the PPC64 ELF v2 ABI so Go can inter-operate with platform debuggers and profilers.

This stack layout is used by both register-based (ABIInternal) and stack-based (ABI0) calling conventions.

Flags

The condition register consists of 8 condition code register fields CR0-CR7. Go generated code only sets and uses CR0, commonly set by compare functions and use to determine the target of a conditional branch. The generated code does not set or use CR1-CR7.

The floating point status and control register (FPSCR) is initialized to 0 by the kernel at startup of the Go program and not changed by the Go generated code.

riscv64 architecture

The riscv64 architecture uses X10 – X17, X8, X9, X18 – X23 for integer arguments and results.

It uses F10 – F17, F8, F9, F18 – F23 for floating-point arguments and results.

Special-purpose registers used within Go generated code and Go assembly code are as follows:

Register	Call meaning	Return meaning	Body meaning
X0	Zero value	Same	Same
X1	Link register	Link register	Scratch
X2	Stack pointer	Same	Same
X3	Global pointer	Same	Used by dynamic linker
X4	TLS (thread pointer)	TLS	Scratch
X24,X25	Scratch	Scratch	Used by duffcopy, duffzero
X26	Closure context pointer	Scratch	Scratch
X27	Current goroutine	Same	Same
X31	Scratch	Scratch	Scratch

Rationale: These register meanings are compatible with Go's stack-based calling convention. Context register X20 will change to X26, duffcopy, duffzero register will change to X24, X25 before this register ABI been adopted. X10 – X17, X8, X9, X18 – X23, is the same order as A0 – A7, S0 – S7 in platform ABI. F10 – F17, F8, F9, F18 – F23, is the same order as FA0 – FA7, FS0 – FS7 in platform ABI. X8 – X23, F8 – F15 are used for compressed instruction (RVC) which will benefit code size in the future.

Stack layout

The stack pointer, X2, grows down and is aligned to 8 bytes.

A function's stack frame, after the frame is created, is laid out as follows:

```
+------------------------------+
| ... locals ...               |
| ... outgoing arguments ...   |
| return PC                    | ← X2 points to
+------------------------------+ ↓ lower addresses
```

The "return PC" is loaded to the link register, X1, as part of the riscv64 `CALL` operation.

Flags

The riscv64 has Zicsr extension for control and status register (CSR) and treated as scratch register. All bits in CSR are system flags and are not modified by Go.

Future directions

Spill path improvements

The ABI currently reserves spill space for argument registers so the compiler can statically generate an argument spill path before calling into `runtime.morestack` to grow the stack. This ensures there will be sufficient spill space even when the stack is nearly exhausted and keeps stack growth and stack scanning essentially unchanged from ABI0.

However, this wastes stack space (the median wastage is 16 bytes per call), resulting in larger stacks and increased cache footprint. A better approach would be to reserve stack space only when spilling. One way to ensure enough space is available to spill would be for every function to ensure there is enough space for the function's own frame *as well as* the spill space of all functions it calls. For most functions, this would change the threshold for the prologue stack growth check. For `nosplit` functions, this would change the threshold used in the linker's static stack size check.

Allocating spill space in the callee rather than the caller may also allow for faster reflection calls in the common case where a function takes only register arguments, since it would allow reflection to make these calls directly without allocating any frame.

The statically-generated spill path also increases code size. It is possible to instead have a generic spill path in the runtime, as part of `morestack`. However, this complicates reserving the spill space, since spilling all possible register arguments would, in most cases, take significantly more space than spilling only those used by a particular function. Some options are to spill to a temporary space and copy back only the registers used by the function, or to grow the stack if necessary before spilling to it (using a temporary space if necessary), or to use a heap-allocated space if insufficient stack space is available. These options all add enough complexity that we will have to make this decision based on the actual code size growth caused by the static spill paths.

Clobber sets

As defined, the ABI does not use callee-save registers. This significantly simplifies the garbage collector and the compiler's register allocator, but at some performance cost. A potentially better balance for Go code would be to use *clobber sets*: for each function, the compiler records the set of registers it clobbers (including those clobbered by functions it calls) and any register not clobbered by function F can remain live across calls to F.

This is generally a good fit for Go because Go's package DAG allows function metadata like the clobber set to flow up the call graph, even across package boundaries. Clobber sets would require relatively little change to the garbage collector, unlike general callee-save registers. One disadvantage of clobber sets over callee-save registers is that they don't help with indirect function calls or interface method calls, since static information isn't available in these cases.

Large aggregates

Go encourages passing composite values by value, and this simplifies reasoning about mutation and races. However, this comes at a performance cost for large composite values. It may be possible to instead transparently pass large composite values by reference and delay copying until it is actually necessary.

Appendix: Register usage analysis

In order to understand the impacts of the above design on register usage, we analyzed the impact of the above ABI on a large code base: cmd/kubelet from Kubernetes at tag v1.18.8.

The following table shows the impact of different numbers of available integer and floating-point registers on argument assignment:

			stack args			spills			stack total		
ints	floats	% fit	p50	p95	p99	p50	p95	p99	p50	p95	p99
0	0	6.3%	32	152	256	0	0	0	32	152	256
0	8	6.4%	32	152	256	0	0	0	32	152	256
1	8	21.3%	24	144	248	8	8	8	32	152	256
2	8	38.9%	16	128	224	8	16	16	24	136	240
3	8	57.0%	0	120	224	16	24	24	24	136	240
4	8	73.0%	0	120	216	16	32	32	24	136	232
5	8	83.3%	0	112	216	16	40	40	24	136	232
6	8	87.5%	0	112	208	16	48	48	24	136	232
7	8	89.8%	0	112	208	16	48	56	24	136	232
8	8	91.3%	0	112	200	16	56	64	24	136	232
9	8	92.1%	0	112	192	16	56	72	24	136	232
10	8	92.6%	0	104	192	16	56	72	24	136	232
11	8	93.1%	0	104	184	16	56	80	24	128	232
12	8	93.4%	0	104	176	16	56	88	24	128	232
13	8	94.0%	0	88	176	16	56	96	24	128	232
14	8	94.4%	0	80	152	16	64	104	24	128	232
15	8	94.6%	0	80	152	16	64	112	24	128	232
16	8	94.9%	0	16	152	16	64	112	24	128	232
∞	8	99.8%	0	0	0	24	112	216	24	120	216

The first two columns show the number of available integer and floating-point registers. The first row shows the results for 0 integer and 0 floating-point registers, which is equivalent to ABI0. We found that any reasonable number of floating-point registers has the same effect, so we fixed it at 8 for all other rows.

The “% fit” column gives the fraction of functions where all arguments and results are register-assigned and no arguments are passed on the stack. The three “stack args” columns give the median, 95th and 99th percentile number of bytes of stack arguments. The “spills” columns likewise summarize the number of bytes in on-stack spill space. And “stack total” summarizes the sum of stack arguments and on-stack spill slots. Note that these are three different distributions; for example, there’s no single function that takes 0 stack argument bytes, 16 spill bytes, and 24 total stack bytes.

From this, we can see that the fraction of functions that fit entirely in registers grows very slowly once it reaches about 90%, though curiously there is a small minority of functions that could benefit from a huge number of registers. Making 9 integer registers available on amd64 puts it in this realm. We also see that the stack space required for most functions is fairly small. While the increasing space required for spills largely balances out the decreasing space required for stack arguments as the number of available registers increases, there is a general reduction in the total stack space required with more available registers. This does, however, suggest that eliminating spill slots in the future would noticeably reduce stack requirements.