A Quick Guide to Go's Assembler

A Quick Guide to Go's Assembler

This document is a quick outline of the unusual form of assembly language used by the `gc` Go compiler. The document is not comprehensive.

The assembler is based on the input style of the Plan 9 assemblers, which is documented in detail elsewhere. If you plan to write assembly language, you should read that document although much of it is Plan 9-specific. The current document provides a summary of the syntax and the differences with what is explained in that document, and describes the peculiarities that apply when writing assembly code to interact with Go.

The most important thing to know about Go's assembler is that it is not a direct representation of the underlying machine. Some of the details map precisely to the machine, but some do not. This is because the compiler suite (see this description) needs no assembler pass in the usual pipeline. Instead, the compiler operates on a kind of semi-abstract instruction set, and instruction selection occurs partly after code generation. The assembler works on the semi-abstract form, so when you see an instruction like `MOV` what the toolchain actually generates for that operation might not be a move instruction at all, perhaps a clear or load. Or it might correspond exactly to the machine instruction with that name. In general, machine-specific operations tend to appear as themselves, while more general concepts like memory move and subroutine call and return are more abstract. The details vary with architecture, and we apologize for the imprecision; the situation is not well-defined.

The assembler program is a way to parse a description of that semi-abstract instruction set and turn it into instructions to be input to the linker. If you want to see what the instructions look like in assembly for a given architecture, say amd64, there are many examples in the sources of the standard library, in packages such as `runtime` and `math/big`. You can also examine what the compiler emits as assembly code (the actual output may differ from what you see here):

```
$ cat x.go
package main

func main() {
        println(3)
}
$ GOOS=linux GOARCH=amd64 go tool compile -S x.go        # or: go build -gcflags -S
x.go
"".main STEXT size=74 args=0x0 locals=0x10
        0x0000 00000 (x.go:3)   TEXT    "".main(SB), $16-0
        0x0000 00000 (x.go:3)   MOVQ    (TLS), CX
        0x0009 00009 (x.go:3)   CMPQ    SP, 16(CX)
        0x000d 00013 (x.go:3)   JLS     67
        0x000f 00015 (x.go:3)   SUBQ    $16, SP
        0x0013 00019 (x.go:3)   MOVQ    BP, 8(SP)
        0x0018 00024 (x.go:3)   LEAQ    8(SP), BP
        0x001d 00029 (x.go:3)   FUNCDATA        $0,
gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        0x001d 00029 (x.go:3)   FUNCDATA        $1,
gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        0x001d 00029 (x.go:3)   FUNCDATA        $2,
gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        0x001d 00029 (x.go:4)   PCDATA  $0, $0
        0x001d 00029 (x.go:4)   PCDATA  $1, $0
        0x001d 00029 (x.go:4)   CALL    runtime.printlock(SB)
        0x0022 00034 (x.go:4)   MOVQ    $3, (SP)
        0x002a 00042 (x.go:4)   CALL    runtime.printint(SB)
        0x002f 00047 (x.go:4)   CALL    runtime.printnl(SB)
        0x0034 00052 (x.go:4)   CALL    runtime.printunlock(SB)
        0x0039 00057 (x.go:5)   MOVQ    8(SP), BP
        0x003e 00062 (x.go:5)   ADDQ    $16, SP
        0x0042 00066 (x.go:5)   RET
        0x0043 00067 (x.go:5)   NOP
        0x0043 00067 (x.go:3)   PCDATA  $1, $-1
        0x0043 00067 (x.go:3)   PCDATA  $0, $-1
        0x0043 00067 (x.go:3)   CALL    runtime.morestack_noctxt(SB)
        0x0048 00072 (x.go:3)   JMP     0
...
```

The `FUNCDATA` and `PCDATA` directives contain information for use by the garbage collector; they are introduced by the compiler.

To see what gets put in the binary after linking, use `go tool objdump`:

```
$ go build -o x.exe x.go
$ go tool objdump -s main.main x.exe
TEXT main.main(SB) /tmp/x.go
  x.go:3                0x10501c0               65488b0c2530000000      MOVQ GS:0x30,
CX
  x.go:3                0x10501c9               483b6110                CMPQ 0x10(CX),
SP
  x.go:3                0x10501cd               7634                    JBE 0x1050203
  x.go:3                0x10501cf               4883ec10                SUBQ $0x10, SP
  x.go:3                0x10501d3               48896c2408              MOVQ BP,
0x8(SP)
  x.go:3                0x10501d8               488d6c2408              LEAQ 0x8(SP),
BP
  x.go:4                0x10501dd               e86e45fdff              CALL
runtime.printlock(SB)
  x.go:4                0x10501e2               48c7042403000000        MOVQ $0x3,
0(SP)
  x.go:4                0x10501ea               e8e14cfdff              CALL
runtime.printint(SB)
  x.go:4                0x10501ef               e8ec47fdff              CALL
runtime.printnl(SB)
  x.go:4                0x10501f4               e8d745fdff              CALL
runtime.printunlock(SB)
  x.go:5                0x10501f9               488b6c2408              MOVQ 0x8(SP),
BP
  x.go:5                0x10501fe               4883c410                ADDQ $0x10, SP
  x.go:5                0x1050202               c3                      RET
  x.go:3                0x1050203               e83882ffff              CALL
runtime.morestack_noctxt(SB)
  x.go:3                0x1050208               ebb6                    JMP
main.main(SB)
```

Constants

Although the assembler takes its guidance from the Plan 9 assemblers, it is a distinct program, so there are some differences. One is in constant evaluation. Constant expressions in the assembler are parsed using Go's operator precedence, not the C-like precedence of the original. Thus `3&1<<2` is 4, not 0—it parses as `(3&1)<<2` not `3&(1<<2)`. Also, constants are always evaluated as 64-bit unsigned integers. Thus `-2` is not the integer value minus two, but the unsigned 64-bit integer with the same bit pattern. The distinction rarely matters but to avoid ambiguity, division or right shift where the right operand's high bit is set is rejected.

Symbols

Some symbols, such as `R1` or `LR`, are predefined and refer to registers. The exact set depends on the architecture.

There are four predeclared symbols that refer to pseudo-registers. These are not real registers, but rather virtual registers maintained by the toolchain, such as a frame pointer. The set of pseudo-registers is the same for all architectures:

- FP: Frame pointer: arguments and locals.
- PC: Program counter: jumps and branches.
- SB: Static base pointer: global symbols.
- SP: Stack pointer: top of stack.

All user-defined symbols are written as offsets to the pseudo-registers FP (arguments and locals) and SB (globals).

The SB pseudo-register can be thought of as the origin of memory, so the symbol `foo(SB)` is the name `foo` as an address in memory. This form is used to name global functions and data. Adding `<>` to the name, as in `foo<>(SB)`, makes the name visible only in the current source file, like a top-level `static` declaration in a C file. Adding an offset to the name refers to that offset from the symbol's address, so `foo+4(SB)` is four bytes past the start of `foo`.

The FP pseudo-register is a virtual frame pointer used to refer to function arguments. The compilers maintain a virtual frame pointer and refer to the arguments on the stack as offsets from that pseudo-register. Thus `0(FP)` is the first argument to the function, `8(FP)` is the second (on a 64-bit machine), and so on. However, when referring to a function argument this way, it is necessary to place a name at the beginning, as in `first_arg+0(FP)` and `second_arg+8(FP)`. (The meaning of the offset—offset from the frame pointer—distinct from its use with SB, where it is an offset from the symbol.) The assembler enforces this convention, rejecting plain `0(FP)` and `8(FP)`. The actual name is semantically irrelevant but should be used to document the argument's name. It is worth stressing that FP is always a pseudo-register, not a hardware register, even on architectures with a hardware frame pointer.

For assembly functions with Go prototypes, `go vet` will check that the argument names and offsets match. On 32-bit systems, the low and high 32 bits of a 64-bit value are distinguished by adding a `_lo` or `_hi` suffix to the name, as in `arg_lo+0(FP)` or `arg_hi+4(FP)`. If a Go prototype does not name its result, the expected assembly name is `ret`.

The SP pseudo-register is a virtual stack pointer used to refer to frame-local variables and the arguments being prepared for function calls. It points to the top of the local stack frame, so references should use negative offsets in the range [−framesize, 0): `x−8(SP)`, `y−4(SP)`, and so on.

On architectures with a hardware register named SP, the name prefix distinguishes references to the virtual stack pointer from references to the architectural SP register. That is, `x−8(SP)` and `−8(SP)` are different memory locations: the first refers to the virtual stack pointer pseudo-register, while the second refers to the hardware's SP register.

On machines where SP and PC are traditionally aliases for a physical, numbered register, in the Go assembler the names SP and PC are still treated specially; for instance, references to SP require a symbol, much like FP. To access the actual hardware register use the true R name. For example, on the ARM architecture the hardware SP and PC are accessible as R13 and R15.

Branches and direct jumps are always written as offsets to the PC, or as jumps to labels:

```
label:
        MOVW $0, R1
        JMP label
```

Each label is visible only within the function in which it is defined. It is therefore permitted for multiple functions in a file to define and use the same label names. Direct jumps and call instructions can target text symbols, such as `name(SB)`, but not offsets from symbols, such as `name+4(SB)`.

Instructions, registers, and assembler directives are always in UPPER CASE to remind you that assembly programming is a fraught endeavor. (Exception: the g register renaming on ARM.)

In Go object files and binaries, the full name of a symbol is the package path followed by a period and the symbol name: `fmt.Printf` or `math/rand.Int`. Because the assembler's parser treats period and slash as punctuation, those strings cannot be used directly as identifier names. Instead, the assembler allows the middle dot character U+00B7 and the division slash U+2215 in identifiers and rewrites them to plain period and slash.

Within an assembler source file, the symbols above are written as `fmt·Printf` and `math/rand·Int`. The assembly listings generated by the compilers when using the –S flag show the period and slash directly instead of the Unicode replacements required by the assemblers.

Most hand-written assembly files do not include the full package path in symbol names, because the linker inserts the package path of the current object file at the beginning of any name starting with a period: in an assembly source file within the math/rand package implementation, the package's Int function can be referred to as `·Int`. This convention avoids the need to hard-code a package's import path in its own source code, making it easier to move the code from one location to another.

Directives

The assembler uses various directives to bind text and data to symbol names. For example, here is a simple complete function definition. The `TEXT` directive declares the symbol `runtime·profileloop` and the instructions that follow form the body of the function. The last instruction in a `TEXT` block must be some sort of jump, usually a `RET` (pseudo-)instruction. (If it's not, the linker will append a jump-to-itself instruction; there is no fallthrough in `TEXT`s.) After the symbol, the arguments are flags (see below) and the frame size, a constant (but see below):

```
TEXT runtime·profileloop(SB),NOSPLIT,$8
        MOVQ    $runtime·profileloop1(SB), CX
        MOVQ    CX, 0(SP)
        CALL    runtime·externalthreadhandler(SB)
        RET
```

In the general case, the frame size is followed by an argument size, separated by a minus sign. (It's not a subtraction, just idiosyncratic syntax.) The frame size `$24–8` states that the function has a 24-byte frame and is called with 8 bytes of argument, which live on the caller's frame. If `NOSPLIT` is not specified for the `TEXT`, the argument size must be provided. For assembly functions with Go prototypes, `go vet` will check that the argument size is correct.

Note that the symbol name uses a middle dot to separate the components and is specified as an offset from the static base pseudo-register `SB`. This function would be called from Go source for package `runtime` using the simple name `profileloop`.

Global data symbols are defined by a sequence of initializing `DATA` directives followed by a `GLOBL` directive. Each `DATA` directive initializes a section of the corresponding memory. The memory not explicitly initialized is zeroed. The general form of the `DATA` directive is

```
DATA    symbol+offset(SB)/width, value
```

which initializes the symbol memory at the given offset and width with the given value. The `DATA` directives for a given symbol must be written with increasing offsets.

The `GLOBL` directive declares a symbol to be global. The arguments are optional flags and the size of the data being declared as a global, which will have initial value all zeros unless a `DATA` directive has initialized it. The `GLOBL` directive must follow any corresponding `DATA` directives.

For example,

```
DATA divtab<>+0x00(SB)/4, $0xf4f8fcff
DATA divtab<>+0x04(SB)/4, $0xe6eaedf0
...
DATA divtab<>+0x3c(SB)/4, $0x81828384
GLOBL divtab<>(SB), RODATA, $64
```

```
GLOBL runtime·tlsoffset(SB), NOPTR, $4
```

declares and initializes `divtab<>`, a read-only 64-byte table of 4-byte integer values, and declares `runtime·tlsoffset`, a 4-byte, implicitly zeroed variable that contains no pointers.
There may be one or two arguments to the directives. If there are two, the first is a bit mask of flags, which can be written as numeric expressions, added or or-ed together, or can be set symbolically for easier absorption by a human. Their values, defined in the standard `#include` file `textflag.h`, are:

- `NOPROF` = 1
  (For `TEXT` items.) Don't profile the marked function. This flag is deprecated.
- `DUPOK` = 2
  It is legal to have multiple instances of this symbol in a single binary. The linker will choose one of the duplicates to use.
- `NOSPLIT` = 4
  (For `TEXT` items.) Don't insert the preamble to check if the stack must be split. The frame for the routine, plus anything it calls, must fit in the spare space at the top of the stack segment. Used to protect routines such as the stack splitting code itself.
- `RODATA` = 8
  (For `DATA` and `GLOBL` items.) Put this data in a read-only section.
- `NOPTR` = 16
  (For `DATA` and `GLOBL` items.) This data contains no pointers and therefore does not need to be scanned by the garbage collector.
- `WRAPPER` = 32
  (For `TEXT` items.) This is a wrapper function and should not count as disabling `recover`.
- `NEEDCTXT` = 64
  (For `TEXT` items.) This function is a closure so it uses its incoming context register.

Runtime Coordination

For garbage collection to run correctly, the runtime must know the location of pointers in all global data and in most stack frames. The Go compiler emits this information when compiling Go source files, but assembly programs must define it explicitly.

A data symbol marked with the `NOPTR` flag (see above) is treated as containing no pointers to runtime-allocated data. A data symbol with the `RODATA` flag is allocated in read-only memory and is therefore treated as implicitly marked `NOPTR`. A data symbol with a total size smaller than a pointer is also treated as implicitly marked `NOPTR`. It is not possible to define a symbol containing pointers in an assembly source file; such a symbol must be defined in a Go source file instead. Assembly source can still refer to the symbol by name even without `DATA` and `GLOBL` directives. A good general rule of thumb is to define all non-`RODATA` symbols in Go instead of in assembly.

Each function also needs annotations giving the location of live pointers in its arguments, results, and local stack frame. For an assembly function with no pointer results and either no local stack frame or no function calls, the only requirement is to define a Go prototype for the function in a Go source file in the same package. The name of the assembly function must not contain the package name component (for example, function `Syscall` in package `syscall` should use the name `·Syscall` instead of the equivalent name `syscall·Syscall` in its `TEXT` directive). For more complex situations, explicit annotation is needed. These annotations use pseudo-instructions defined in the standard `#include` file `funcdata.h`.

If a function has no arguments and no results, the pointer information can be omitted. This is indicated by an argument size annotation of `$`*`n`*`-0` on the `TEXT` instruction. Otherwise, pointer information must be provided by a Go prototype for the function in a Go source file, even for assembly functions not called directly from Go. (The prototype will also let `go vet` check the argument references.) At the start of the function, the arguments are assumed to be initialized but the results are assumed uninitialized. If the results will hold live pointers during a call instruction, the function should start by zeroing the results and then executing the pseudo-instruction `GO_RESULTS_INITIALIZED`. This instruction records that the results are now initialized and should be scanned during stack movement and garbage collection. It is typically easier to arrange that assembly functions do not

during stack movement and garbage collection. It is typically easier to arrange that assembly functions do not return pointers or do not contain call instructions; no assembly functions in the standard library use `GO_RESULTS_INITIALIZED`.

If a function has no local stack frame, the pointer information can be omitted. This is indicated by a local frame size annotation of `$0-`*n* on the `TEXT` instruction. The pointer information can also be omitted if the function contains no call instructions. Otherwise, the local stack frame must not contain pointers, and the assembly must confirm this fact by executing the pseudo-instruction `NO_LOCAL_POINTERS`. Because stack resizing is implemented by moving the stack, the stack pointer may change during any function call: even pointers to stack data must not be kept in local variables.

Assembly functions should always be given Go prototypes, both to provide pointer information for the arguments and results and to let `go vet` check that the offsets being used to access them are correct.

Architecture-specific details

It is impractical to list all the instructions and other details for each machine. To see what instructions are defined for a given machine, say ARM, look in the source for the `obj` support library for that architecture, located in the directory `src/cmd/internal/obj/arm`. In that directory is a file `a.out.go`; it contains a long list of constants starting with `A`, like this:

```
const (
        AAND = obj.ABaseARM + obj.A_ARCHSPECIFIC + iota
        AEOR
        ASUB
        ARSB
        AADD
        ...
```

This is the list of instructions and their spellings as known to the assembler and linker for that architecture. Each instruction begins with an initial capital `A` in this list, so `AAND` represents the bitwise and instruction, `AND` (without the leading `A`), and is written in assembly source as `AND`. The enumeration is mostly in alphabetical order. (The architecture-independent `AXXX`, defined in the `cmd/internal/obj` package, represents an invalid instruction). The sequence of the `A` names has nothing to do with the actual encoding of the machine instructions. The `cmd/internal/obj` package takes care of that detail.

The instructions for both the 386 and AMD64 architectures are listed in `cmd/internal/obj/x86/a.out.go`.

The architectures share syntax for common addressing modes such as `(R1)` (register indirect), `4(R1)` (register indirect with offset), and `$foo(SB)` (absolute address). The assembler also supports some (not necessarily all) addressing modes specific to each architecture. The sections below list these.

One detail evident in the examples from the previous sections is that data in the instructions flows from left to right: `MOVQ $0, CX` clears `CX`. This rule applies even on architectures where the conventional notation uses the opposite direction.

Here follow some descriptions of key Go-specific details for the supported architectures.

32-bit Intel 386

The runtime pointer to the `g` structure is maintained through the value of an otherwise unused (as far as Go is concerned) register in the MMU. An OS-dependent macro `get_tls` is defined for the assembler if the source is in the `runtime` package and includes a special header, `go_tls.h`:

```
#include "go_tls.h"
```

Within the runtime, the `get_tls` macro loads its argument register with a pointer to the `g` pointer, and the `g` struct contains the `m` pointer. There's another special header containing the offsets for each element of `g`, called `go_asm.h`. The sequence to load `g` and `m` using `CX` looks like this:

```
#include "go_tls.h"
#include "go_asm.h"
...
get_tls(CX)
MOVL    g(CX), AX     // Move g into AX.
MOVL    g_m(AX), BX   // Move g.m into BX.
```

Note: The code above works only in the `runtime` package, while `go_tls.h` also applies to arm, amd64 and amd64p32, and `go_asm.h` applies to all architectures.

Addressing modes:

- `(DI)(BX*2)`: The location at address `DI` plus `BX*2`.
- `64(DI)(BX*2)`: The location at address `DI` plus `BX*2` plus 64. These modes accept only 1, 2, 4, and 8 as scale factors.

When using the compiler and assembler's `-dynlink` or `-shared` modes, any load or store of a fixed memory location such as a global variable must be assumed to overwrite `CX`. Therefore, to be safe for use with these modes, assembly sources should typically avoid CX except between memory references.

64-bit Intel 386 (a.k.a. amd64)

The two architectures behave largely the same at the assembler level. Assembly code to access the `m` and `g` pointers on the 64-bit version is the same as on the 32-bit 386, except it uses `MOVQ` rather than `MOVL`:

```
get_tls(CX)
MOVQ    g(CX), AX     // Move g into AX.
MOVQ    g_m(AX), BX   // Move g.m into BX.
```

ARM

The registers `R10` and `R11` are reserved by the compiler and linker.

`R10` points to the `g` (goroutine) structure. Within assembler source code, this pointer must be referred to as `g`; the name `R10` is not recognized.

To make it easier for people and compilers to write assembly, the ARM linker allows general addressing forms and pseudo-operations like `DIV` or `MOD` that may not be expressible using a single hardware instruction. It implements these forms as multiple instructions, often using the `R11` register to hold temporary values. Hand-written assembly can use `R11`, but doing so requires being sure that the linker is not also using it to implement any of the other instructions in the function.

When defining a `TEXT`, specifying frame size `$-4` tells the linker that this is a leaf function that does not need to save `LR` on entry.

The name `SP` always refers to the virtual stack pointer described earlier. For the hardware register, use `R13`.

Condition code syntax is to append a period and the one- or two-letter code to the instruction, as in `MOVW.EQ`. Multiple codes may be appended: `MOVM.IA.W`. The order of the code modifiers is irrelevant.

Addressing modes:

- `R0->16`
- `R0>>16`

R0>>16
R0<<16
R0@>16: For <<, left shift R0 by 16 bits. The other codes are –> (arithmetic right shift), >> (logical right shift), and @> (rotate right).

- R0–>R1
  R0>>R1
  R0<<R1
  R0@>R1: For <<, left shift R0 by the count in R1. The other codes are –> (arithmetic right shift), >> (logical right shift), and @> (rotate right).
- [R0,g,R12–R15]: For multi-register instructions, the set comprising R0, g, and R12 through R15 inclusive.
- (R5, R6): Destination register pair.

ARM64

The ARM64 port is in an experimental state.

R18 is the "platform register", reserved on the Apple platform. To prevent accidental misuse, the register is named R18_PLATFORM. R27 and R28 are reserved by the compiler and linker. R29 is the frame pointer. R30 is the link register.

Instruction modifiers are appended to the instruction following a period. The only modifiers are P (postincrement) and W (preincrement): MOVW.P, MOVW.W

Addressing modes:

- R0–>16
  R0>>16
  R0<<16
  R0@>16: These are the same as on the 32-bit ARM.
- $(8<<12): Left shift the immediate value 8 by 12 bits.
- 8(R0): Add the value of R0 and 8.
- (R2)(R0): The location at R0 plus R2.
- R0.UXTB
  R0.UXTB<<imm: UXTB: extract an 8-bit value from the low-order bits of R0 and zero-extend it to the size of R0. R0.UXTB<<imm: left shift the result of R0.UXTB by imm bits. The imm value can be 0, 1, 2, 3, or 4. The other extensions include UXTH (16-bit), UXTW (32-bit), and UXTX (64-bit).
- R0.SXTB
  R0.SXTB<<imm: SXTB: extract an 8-bit value from the low-order bits of R0 and sign-extend it to the size of R0. R0.SXTB<<imm: left shift the result of R0.SXTB by imm bits. The imm value can be 0, 1, 2, 3, or 4. The other extensions include SXTH (16-bit), SXTW (32-bit), and SXTX (64-bit).
- (R5, R6): Register pair for LDAXP/LDP/LDXP/STLXP/STP/STP.

Reference: Go ARM64 Assembly Instructions Reference Manual

PPC64

This assembler is used by GOARCH values ppc64 and ppc64le.

Reference: Go PPC64 Assembly Instructions Reference Manual

IBM z/Architecture, a.k.a. s390x

The registers R10 and R11 are reserved. The assembler uses them to hold temporary values when assembling some instructions.

R13 points to the g (goroutine) structure. This register must be referred to as g; the name R13 is not recognized.

R15 points to the stack frame and should typically only be accessed using the virtual registers SP and FP.

R15 points to the stack frame and should typically only be accessed using the virtual registers SP and FP.

Load- and store-multiple instructions operate on a range of registers. The range of registers is specified by a start register and an end register. For example, `LMG (R9), R5, R7` would load R5, R6 and R7 with the 64-bit values at `0(R9)`, `8(R9)` and `16(R9)` respectively.

Storage-and-storage instructions such as `MVC` and `XC` are written with the length as the first argument. For example, `XC $8, (R9), (R9)` would clear eight bytes at the address specified in R9.

If a vector instruction takes a length or an index as an argument then it will be the first argument. For example, `VLEIF $1, $16, V2` will load the value sixteen into index one of V2. Care should be taken when using vector instructions to ensure that they are available at runtime. To use vector instructions a machine must have both the vector facility (bit 129 in the facility list) and kernel support. Without kernel support a vector instruction will have no effect (it will be equivalent to a `NOP` instruction).

Addressing modes:

- `(R5)(R6*1)`: The location at R5 plus R6. It is a scaled mode as on the x86, but the only scale allowed is 1.

MIPS, MIPS64

General purpose registers are named `R0` through `R31`, floating point registers are `F0` through `F31`.

`R30` is reserved to point to `g`. `R23` is used as a temporary register.

In a `TEXT` directive, the frame size `$-4` for MIPS or `$-8` for MIPS64 instructs the linker not to save `LR`.

`SP` refers to the virtual stack pointer. For the hardware register, use `R29`.

Addressing modes:

- `16(R1)`: The location at `R1` plus 16.
- `(R1)`: Alias for `0(R1)`.

The value of `GOMIPS` environment variable (`hardfloat` or `softfloat`) is made available to assembly code by predefining either `GOMIPS_hardfloat` or `GOMIPS_softfloat`.

The value of `GOMIPS64` environment variable (`hardfloat` or `softfloat`) is made available to assembly code by predefining either `GOMIPS64_hardfloat` or `GOMIPS64_softfloat`.

Unsupported opcodes

The assemblers are designed to support the compiler so not all hardware instructions are defined for all architectures: if the compiler doesn't generate it, it might not be there. If you need to use a missing instruction, there are two ways to proceed. One is to update the assembler to support that instruction, which is straightforward but only worthwhile if it's likely the instruction will be used again. Instead, for simple one-off cases, it's possible to use the `BYTE` and `WORD` directives to lay down explicit data into the instruction stream within a `TEXT`. Here's how the 386 runtime defines the 64-bit atomic load function.

```
// uint64 atomicload64(uint64 volatile* addr);
// so actually
// void atomicload64(uint64 *res, uint64 volatile *addr);
TEXT runtime·atomicload64(SB), NOSPLIT, $0-12
        MOVL    ptr+0(FP), AX
        TESTL   $7, AX
        JZ      2(PC)
        MOVL    0, AX // crash with nil ptr deref
        LEAL    ret_lo+4(FP), BX
        // MOVQ (%EAX), %MM0
        BYTE $0x0f; BYTE $0x6f; BYTE $0x00
        // MOVQ %MM0, 0(%EBX)
        BYTE $0x0f; BYTE $0x7f; BYTE $0x03
        // EMMS
        BYTE $0x0F; BYTE $0x77
        RET
```