WebAssembly architecture for Go

Document revision 1 - 28th February 2018

This documents describes the design decisions that went into the new WebAssembly architecture (short "wasm") for the Go compiler. The goal is to upstream it to the official Go repository in the current development cycle, targeting Go 1.11. The WebAssembly architecture will allow Go to become an alternative to JavaScript for writing code that runs in a web browser. This new freedom of choice will hopefully have a positive impact on the software engineering ecosystem overall.

Introduction to WebAssembly

WebAssembly is in many regards quite different from all other architectures. It is not intended to be processed by a CPU directly, but instead is an intermediate representation that is compiled to actual machine code by the WebAssembly runtime environment. Thus the design of WebAssembly was not constrained by hardware considerations, and the authors could instead focus on their goals of efficiency and output size.

This puts constraints on the logic that can be efficiently expressed in WebAssembly. These constraints are sometimes hard to reconcile with Go's existing design. One may hope that future versions of WebAssembly will add features to fill these gaps. Nevertheless, the current state of the WebAssembly architecture for Go passes all compiler and package tests and supports all major Go features. There was a focus on keeping changes to existing compiler and runtime code to a minimum.

WebAssembly specification
WebAssembly future features

Linear memory

WebAssembly features a linear memory with load and store instructions. This is quite similar to normal memory pages, however it is currently not possible to reserve pages without already allocating them. This will likely be available in the future.

64-bit architecture

WebAssembly has full support for 64 bit integers (in contrast to JavaScript). However, currently memory can only be addressed with 32 bit integers, thus limiting the memory to 4GB. However, the wasm architecture for Go has 64 bit

pointers and a switchover to 64 bit memory operations is planned when they are available.

Threads

Currently WebAssembly has no threads, but they are on the roadmap. Most Go code can run fine on a single thread. The only drawback is that “sysmon” is not available, thus there is no preemption of goroutines.

WebAssembly is a stack machine

All other architectures are register machines, but WebAssembly is not. Instead, it maintains its own opaque stack and each function can have an arbitrary number of local variables.

Fully using WebAssembly’s stack is currently not an option, since Go needs to be able to inspect the stack for garbage collection, stack traces, etc.

Instead, Go maintains its own stack on the linear memory as usual. Registers get mapped to variables: SP, PC_F and PC_B (see below) are global variables. I0 to I15 (64 bit integers) and F0 to F15 (64 bit floats) are local variables. The WebAssembly runtime is likely to map those local variables to CPU registers.

Go’s SSA instructions only operate on registers, for example an add instruction would read two registers, do the addition and then write to a register. WebAssembly’s instructions on the other hand operate on the stack. The add instruction first pops two values from the stack, does the addition, then pushes the result to the stack. To fulfill Go’s semantics, one needs to map Go’s single add instruction to 4 WebAssembly instructions:

- Push the value of local variable A to the stack
- Push the value of local variable B to the stack
- Do addition
- Write value from stack to local variable C

Now consider that B was set to the constant 42 before the addition:

- Push constant 42 to the stack
- Write value from stack to local variable B

This works, but is inefficient. Instead, the stack is used directly by inlining instructions if possible. With inlining it becomes:

- Push the value of local variable A to the stack
- Push constant 42 to the stack
- Do addition
- Write value from stack to local variable C

Please note that the instruction “Push constant 42 to the stack” is not strictly ordered before the add instruction anymore.

Control structures

WebAssembly has no basic blocks or jump instructions, instead it has more high level control structures resembling if-statements and loops. Unfortunately, it also does not currently have any fallback construct like a goto instruction.

According to the authors of WebAssembly, this should not be an issue since most control flow can be turned into high-level structures by the “relooper” algorithm. Unfortunately this is not possible for Go, since resuming a goroutine continues execution at some arbitrary call in a function, thus all those entry points must be reachable from the start of the function. One can see that the assumption of the authors only holds if one is also fully using WebAssembly’s stacks and their yet-to-be-added coroutine feature. One can hope that a goto instruction will be added in the future.

In the meantime, Go generates the equivalent of a big switch statement and uses the PC_B variable to jump to the desired basic block.

Functions

WebAssembly functions do not live in the same address space as the linear memory. Instead, they have a 0-based index. This is reconciled with Go’s concept of a program counter by splitting it into 2 parts: PC_F and PC_B. PC_F is the index of the function to be executed. PC_B is the index of the basic block to be executed. When a single 64-bit PC value is needed it currently gets built as follows: PC_F<<16 + PC_B.

Garbage collection

Go’s garbage collection is fully supported. WebAssembly is planning to add its own garbage collection, but it is hard to imagine that it would yield a better performance than Go’s own GC which is specifically tailored to Go’s needs.

Syscalls

System calls are implemented via calls to the JS environment. Most file system operations are mapped to Node.js’ “fs” module. In the browser, file system operations are currently not available. Network operations are currently simulated internally, not touching the real network, just like with the nacl architecture used by the Go playground. This needs to be improved in the future.

JavaScript interoperability

Interoperability with the JavaScript environment is still in an early state. A few operations are available in the runtime/js package and are currently used by the syscall implementations. However, this package is likely to change in the future.

Conclusion

It works, supports the full Go specification and most major features that people love with Go. There is still work to do to improve the performance. Especially a “goto” operation in WebAssembly would be very helpful. Still, it is in a state where it should be good enough to be merged upstream so more people can start experimenting and contributing.