This document describes when and how to add assembly code to the Go cryptography packages.

In general, the rules are:

- We prefer portable Go, not assembly. Code in assembly means (N packages * M architectures) to maintain, rather than just N packages.
- Minimize use of assembly. We'd rather have a small amount of assembly for a 50% speedup rather than twice as much assembly for a 55% speedup. Explain the decision to place the assembly/Go boundary where it is in the commit message, and support it with benchmarks.
- Use higher level programs to generate non-trivial amounts of assembly, either standalone Go programs or `go get`-able programs, like avo. Output of other reproducible processes (like formally verified code generators) will also be considered. Discuss the implementation strategy on the issue tracker in advance.
- Use small, testable units (25–75 lines) called from higher-level logic written in Go. If using small, testable functions called from logic written in Go is too slow, use small, testable assembly units with Go-compatible wrappers, so that Go tests can still test the individual units.
- Any assembly function needs a reference Go implementation, that's tested side-by-side with the assembly. Follow golang.org/wiki/TargetSpecific for structure and testing practices.
- The interface of the assembly units and of the reference Go implementation must be the same across architectures, unless the platforms have fundamentally different capabilities (such as high-level cryptographic instructions).
- Unless the Go Security team explicitly commits to owning the specific implementation, an external contributor must commit to maintaining it. If changes are required (for example as part of a broader refactor) and the maintainer is not available, the assembly will be removed.
- The code must be tested in our CI. This means there need to be builders that support the instructions, and if there are multiple (or fallback) paths they must be tested separately. (Tip: use `GODEBUG=cpu.X=off` to disable detection of CPU features.)
- Document in the Go code why the implementation requires assembly (specific performance benefit, access to instructions, etc), so we can reevaluate as the compiler improves.

Not all assembly currently in the standard library adheres to this policy. Changes to existing assembly will be discouraged until that implementation is updated to be compliant. New assembly must be compliant.