Golang on RISC-V

The Status and The Future

汪鹏程
ByteDance
Pengcheng Wang

蒙卓
ISCAS
Zhuo Meng

ByteDance字节跳动

Content

Introduction

Go Language

Go (Golang) is a new (not really 😂), general-purpose Programming language feature with:

- Compiled
- Statically typed
- Concurrent
- Simple syntax
- Productive

Flourishing ecosystem:

- Cloud-Native: Kubernetes, k3s, Istio, …
- Container: Docker, Podman, Containerd, …
- Blockchain: Go-ethereum, Hyperledger Fabric, …
- AOB.

Top 10 in TIOBE Index and widely used in top companies like Google, Cloudflare, ByteDance, …

Go is a wise, clean, insightful, fresh thinking approach to the greatest -hits subset of the well understood.

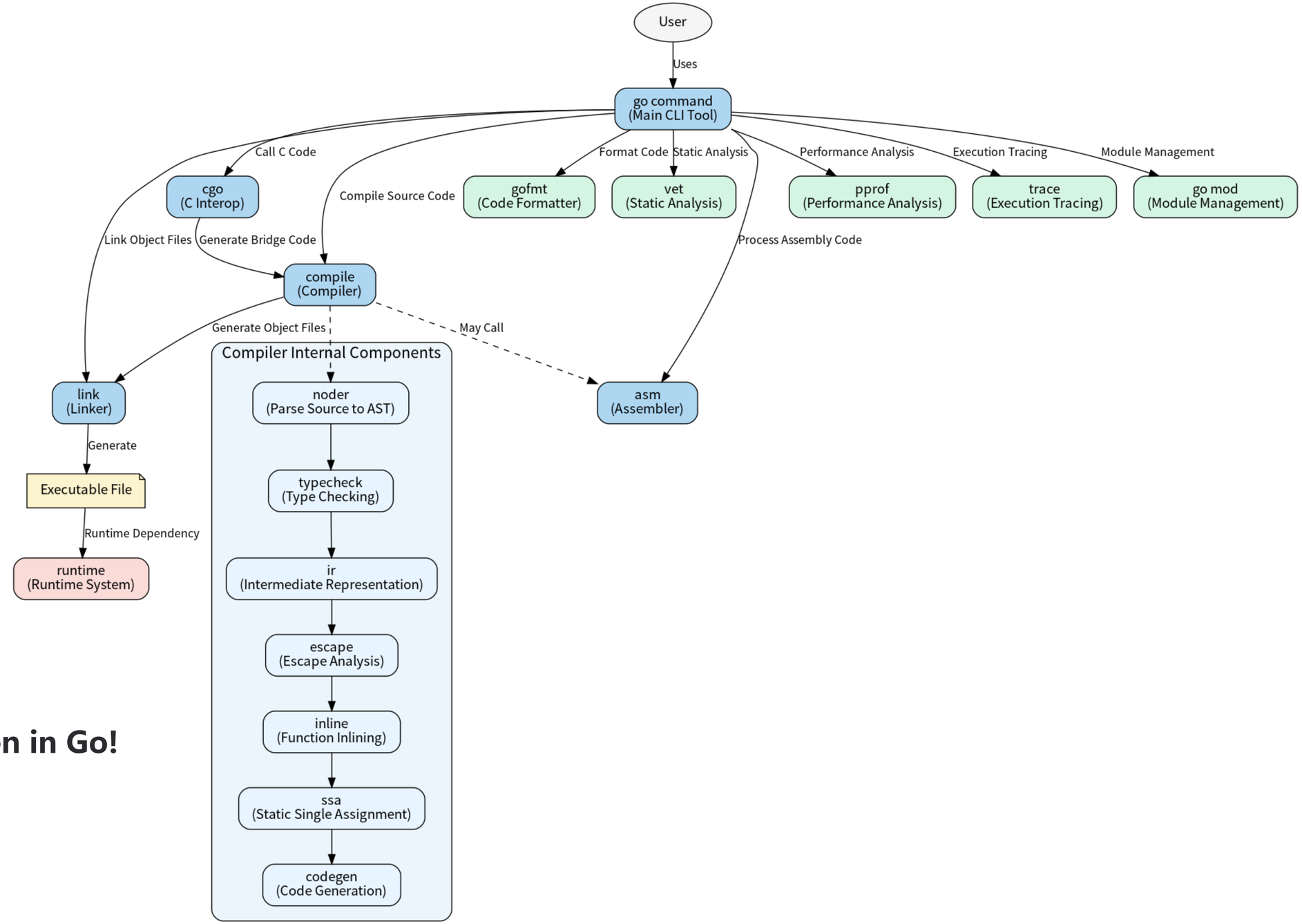
- **Michael** T. Jones

```go
package main

import “fmt”

func main() {
  fmt.Println("Hello! RISC-V!")
}
```

Go example

Go mascot
(by Renee French)

Go Components



Go is bootstrap!
Almost all tools are written in Go!

History

The History

1970 C Language born

2007 Creation of Go

2009 First Go release

2012 Go 1.0

2015 RVI founded

2017 First line to support RISC-V

2020 Go 1.14
First experimental support

2021 RISC-V Vector 1.0 released

2023 RISE founded

2024 Go 1.23
GORISCV64=
rva20u64|rva22u64

2025 Go1.25
Basic RVV
GORISCV64=rva23u64

More ongoing!

1975 First RISC

2010 RISC-V started

RVA23 Profile released

History of Go Language Support for RISC-V

Go 1.14
The first experimental support for RV64G on Linux

Go 1.15
- Improved stability and performance.
- supported asynchronous preemption.

Go 1.16
- Added cgo and PIE support.
- Performance optimizations.
- Code generation improvements.

Go 1.18
Added c-archive and c-shared support.

Go 1.19
Supports passing function arguments and result using registers.

Go 1.20
Added experimental FreeBSD/RISC-V support.

Go 1.23
- Added experimental OpenBSD/RISC-V support.
- Introduced GORISCV64 to select the RISC-V profiles. Allowed values are rva20u64 and rva22u64.

Go 1.24
The objdump tool was supported to disassembly.

Go 1.25
- RISC-V ELF attributes were added.
- Supported plugin build mode.
- The GORISCV64 can accept a rva23u64.
- First version with RVV support.

Go 1.26 (ongoing)
- More RVA23 supports.
- More optimizations.
- More features.

Go on RISC-V keeps evolving!

Status

Supported OS and Core Features

OSs

Stable

Experimental

Experimental

No

No

Build Modes

- standard
- cgo
- pie
- c-archive
- c-shared
- plugin

Libraries & Tools

Most libraries work well, but some needs RISC-V porting.

gopls	works
vim-go	works
huge	works
ollama	works
delve	partial
vscode-go	no
……	

RISC-V Profiles

rva20u64	Fully supported
rva22u64	Fully supported
rva23u64	Partial supported

Note: RVC not support yet!

RISC-V ISA Support in Go

RVA23U64 Requirements	Go 1.23	Go 1.24	Go 1.25	master
I	✓	✓	✓	✓
M	✓	✓	✓	✓
A	✓	✓	✓	✓
F	✓	✓	✓	✓
D	✓	✓	✓	✓
C	✗	✗	✗	🚧
B	✓	✓	✓	✓
V	✗	✗	🚧	🚧
Zihintpause	✗	✗	✗	✗
Zicbom/Zicbop/Zicboz	✗	✗	✗	🚧
Zfhmin	✗	✗	✗	✗
Zvfhmin	✗	✗	✗	🚧
Zvbb	✗	✗	✗	🚧
Zihintntl	✗	✗	✗	🚧
Zicond	✗	✗	✗	🚧
Zimop/Zcmop	✗	✗	✗	✗
Zcb	✗	✗	✗	✗
Zfa	✗	✗	✗	🚧
Zawrs	✗	✗	✗	✗

Assembler support only!

🚧 Lots of WIP patches to review!

The Details of RVV Implementation

- **Go only support PLAN9 assemblies.**
- **vset(i)vl(i):**
  - **VSET(I)VL(I) VL, <VTYPE>, RD**
  - VTYPE can be a scalar register or special operands like ARM's PRFM/TLBI.
- **Format: VOP VS2, VS1/RS1/IMM[, V0], RD**
  - Load/Store instructions:
    - Unit-Stride: VLE8V (X10), V8
    - Strided: VLSE8V (X10), X11, V8
    - Indexed: VLUXEI8V (X10), V4, V8
  - Masked instructions:
    - VADDVV V4, V20, V0, V8
    - VADDVX V4, X10, V0, V8
    - VADDVI V4, $15, V0, V8
  - ALU instructions:
    - VADDVV V4, V20, V8
    - VADDVX V4, X10, V8
    - VADDVI V4, $15, V8
  - Ternary ALU instructions (VOP vs1/rs1, vs2 [, V0], rd)
    - VMACCVV V20, V4, V8
    - VMACCVX X10, V4, V8
    - VMACCVV V20, V4, V0, V8
    - VMACCVX X10, V4, V0, V8
- **Segment load/store instructions have been not supported yet!**

```
// 31.6: Configuration Setting Instructions
VSETVLI  X10, E32, M1, TA, MA, MA, X12
VSETIVLI $0,  E32, M1, TA, MA, X12
VSETIVLI $15, E32, M1, TA, MA, X12
VSETIVLI $31, E32, M1, TA, MA, X12
VSETVL   X10,X11,X12
```

Runtime Detection

Besides RVA23 mandatory extensions, we can detect the CPU features at runtime via riscv_hwprobe and enable compiler/runtime optimizations.

```
var RISCV64 struct {
    _                 CacheLinePad
    HasFastMisaligned bool // Fast misaligned accesses
    HasC              bool // Compressed instruction-set extension
    HasV              bool // Vector extension compatible with RVV 1.0
    HasZba            bool // Address generation instructions extension
    HasZbb            bool // Basic bit-manipulation extension
    HasZbs            bool // Single-bit instructions extension
    HasZvbb           bool // Vector Basic Bit-manipulation
    HasZvbc           bool // Vector Carryless Multiplication
    HasZvkb           bool // Vector Cryptography Bit-manipulation
    HasZvkt           bool // Vector Data-Independent Execution Latency
    HasZvkg           bool // Vector GCM/GMAC
    HasZvkn           bool // NIST Algorithm Suite (AES/SHA256/SHA512)
    HasZvknc          bool // NIST Algorithm Suite with carryless multiply
    HasZvkng          bool // NIST Algorithm Suite with GCM
    HasZvks           bool // ShangMi Algorithm Suite
    HasZvksc          bool // ShangMi Algorithm Suite with carryless multiplication
    HasZvksg          bool // ShangMi Algorithm Suite with GCM
    _                 CacheLinePad
}
```

```
makeOnesCountRISCV64 := func(op ssa.Op) func(s *state, n *ir.CallExpr, args []*ssa.Value) *ssa.Value {
    return func(s *state, n *ir.CallExpr, args []*ssa.Value) *ssa.Value {
        if cfg.goriscv64 >= 22 {
            return s.newValue1(op, types.Types[types.TINT], args[0])
        }

        addr := s.entryNewValue1A(ssa.OpAddr, types.Types[types.TBOOL].PtrTo(), ir.Syms.RISCV64HasZbb, s.sb)
        v := s.load(types.Types[types.TBOOL], addr)
        b := s.endBlock()
        b.Kind = ssa.BlockIf
        b.SetControl(v)
        bTrue := s.f.NewBlock(ssa.BlockPlain)
        bFalse := s.f.NewBlock(ssa.BlockPlain)
        bEnd := s.f.NewBlock(ssa.BlockPlain)
        b.AddEdgeTo(bTrue)
        b.AddEdgeTo(bFalse)
        b.Likely = ssa.BranchLikely // Majority of RISC-V support Zbb.

        // We have the intrinsic - use it directly.
        s.startBlock(bTrue)
        s.vars[n] = s.newValue1(op, types.Types[types.TINT], args[0])
        s.endBlock().AddEdgeTo(bEnd)

        // Call the pure Go version.
        s.startBlock(bFalse)
        s.vars[n] = s.callResult(n, callNormal) // types.Types[TINT]
        s.endBlock().AddEdgeTo(bEnd)

        // Merge results.
        s.startBlock(bEnd)
        return s.variable(n, types.Types[types.TINT])
    }
}
```

Runtime Optimization

Based detected CPU features, we may run the different optimized runtime routines.
Now we have (some of them are ongoing):

- bytealg
- string
- memory
- CRC32
- Crypto
- AOB

```
#ifdef GORISCV64_rva22u64
#define hasZba
#define hasZbb
#define hasZbs
#endif

#ifdef GORISCV64_rva23u64
#define hasV
#define hasZba
#define hasZbb
#define hasZbs
#define hasZfa
#define hasZicond
#endif
```

```
#ifndef hasV
    MOVB    internal/cpu·RISCV64+const_offsetRISCV64HasV(SB), X5
    BEQZ    X5, b_memmove_scalar
#endif
    // Use vector if destination and source are not 8 byte aligned.
    OR  X10, X11, X5
    AND $7, X5
    BNEZ    X5, b_vector_loop
    // Use scalar if destination and source are 8 byte aligned and n <= 64 bytes.
    SUB $32, X12, X6

    BLEZ    X6, b_loop_check
    PCALIGN $16
b_vector_loop:
    VSETVLI X12, E8, M8, TA, MA, X5
    SUB X5, X10
    SUB X5, X11
    VLE8V   (X11), V8
    VSE8V   V8, (X10)
    SUB X5, X12
    BNEZ    X12, b_vector_loop
    RET
b_memmove_scalar:
    // ...
```

Community

- The Go is supported in RISE project: https://lf-rise.atlassian.net/wiki/spaces/HOME/pages/8590598/LR_02+Go
- We need more active contributors!
  - Joel Sing
  - Mark Ryan (RVIOS)
  - Pengcheng Wang (ByteDance)
  - Yaduo Wang (Alibaba)
  - Zhuo Meng (ISCAS)
  - Xianmiao Qu (Alibaba)
  - AOB
- Monthly sync-up meetings at the 4th Thursday every month.
  - Please send an email to markdryan@rivosinc.com if you want to join the meeting!

Future

Roadmap

Go1.26

- Full test on real hardware.
- Fully support of RVA23 profile:
  - RVC is mandatory (default PCQuantum to 2).
  - Assembler support for more extensions.
  - SSA rewrite rules for more extensions.
  - Runtime optimizations (RVV especially).
- Implement race detector and other sanitizers
- Detailed documentation about RISC-V support.

Go1.27

- Implement important optional extensions in RVA23 like Zacas.
- Implement Go atomic mappings.
- Linker Relaxation.
- Compiler/Runtime optimizations.

Go1.28

- Experimental SIMD support.
- RISC-V Specific optimizations:
  - Remove redundant sext instructions.
  - Merge base+offset.
  - etc.

Hopefully Fully support RVA23 and become Tier 2 in 2026.

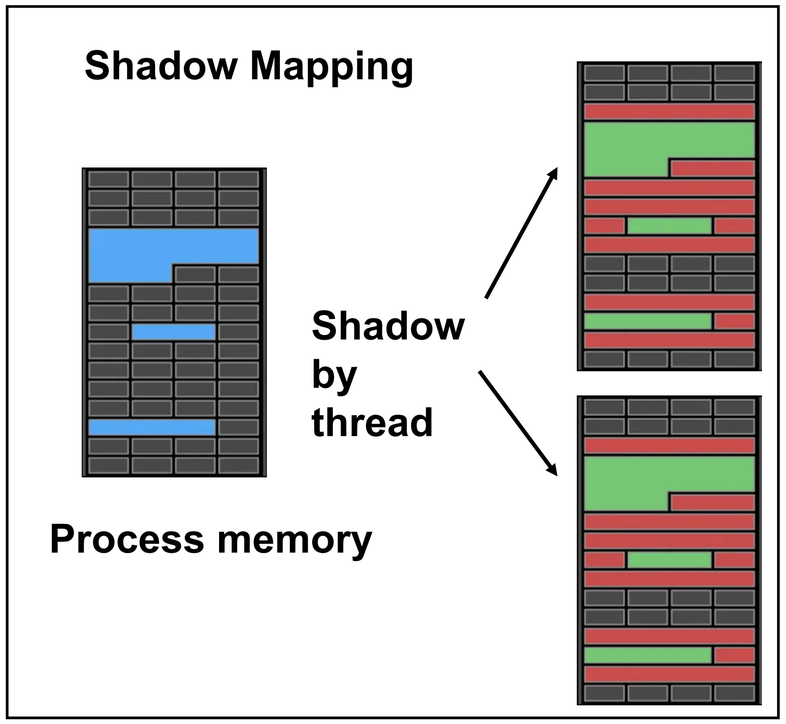
Race detector

Port complete, but require upstream by Joel.

- https://github.com/golang/go/issues/64345
- https://github.com/mengzhuo/go/tree/riscv64-race

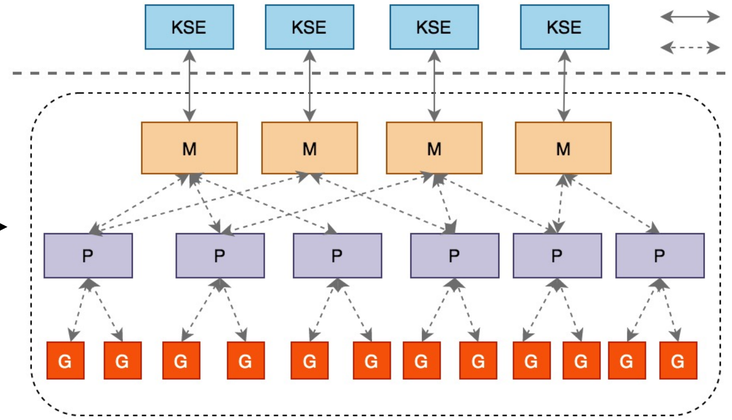
LLVM Threadsanitizer

Compile time Stub



Thread（M）got its shadow After write

Thread（M）drop its shadow after sync



If read on writes shadow before sync, it will panic

Go Atomic Mappings

- The proposed mappings: https://docs.google.com/document/d/1xxRGuISUKN6oNNazQwCUOWTnnG8NrZMMeOYhtqX8qJ0
  - Addresses performance issues.
  - Brings Go into line with the mappings used by the C/C++ compilers.
- Some problems here:
  - Zacas and Zalasr are not mandatory in RVA23.
  - More fine-grained fences are needed to be implemented in Go assembler.

C/C++ Construct	RVWMO AMO Mapping
atomic_<op>(memory_order_relaxed)	amo<op>.{w|d}
atomic_<op>(memory_order_acquire)	amo<op>.{w|d}.aq
atomic_<op>(memory_order_release)	amo<op>.{w|d}.rl
atomic_<op>(memory_order_acq_rel)	amo<op>.{w|d}.aqrl
atomic_<op>(memory_order_seq_cst)	amo<op>.{w|d}.aqrl

C/C++ Construct	RVWMO LR/SC Mapping
atomic_<op>(memory_order_relaxed)	loop:lr.{w|d}; <op>; sc.{w|d}; bnez loop
atomic_<op>(memory_order_acquire)	loop:lr.{w|d}.aq; <op>; sc.{w|d}; bnez loop
atomic_<op>(memory_order_release)	loop:lr.{w|d}; <op>; sc.{w|d}.rl; bnez loop
atomic_<op>(memory_order_acq_rel)	loop:lr.{w|d}.aq; <op>; sc.{w|d}.rl; bnez loop
atomic_<op>(memory_order_seq_cst)	loop:lr.{w|d}.aqrl; <op>; sc.{w|d}.rl; bnez loop
atomic_<op>(memory_order_seq_cst)	loop:lr.{w|d}.aq; <op>; sc.{w|d}.rl; bnez loop

Experimental SIMD support

- Two steps:
  1. First, design low-level architecture-specific vector intrinsics.
  2. Then, design high-level portable vector API.
- The support of scalable vector is still under discusstion.
- Please see [go#73787]

```go
package simd

type Uint32x4 struct { a0, a1, a2, a3 uint32}

func LoadUint32x4(*[4]uint32) Uint32x4
func (Uint32x4) Store(*[4]uint32)

func (Uint32x4) Add(Uint32x4) Uint32x4
func (Uint32x4) Sub(Uint32x4) Uint32x4
func (Uint32x4) Mul(Uint32x4) Uint32x4
func (Uint32x4) Min(Uint32x4) Uint32x4
func (Uint32x4) Max(Uint32x4) Uint32x4
func (Uint32x4) And(Uint32x4) Uint32x4
func (Uint32x4) Or(Uint32x4) Uint32x4
func (Uint32x4) Xor(Uint32x4) Uint32x4
func (Uint32x4) AndNot(Uint32x4) Uint32x4
```

```go
// c = a + b
func add(a *[4]uint32, b *[4]uint32, c *[4]uint32) {
  v0 := LoadUint32x4(a)
  v1 := LoadUint32x4(b)
  v2 := v0.Add(v1)
  v2.Store(c)
}
```

```
add:
  VSETIVLI $4, E32, M1, TA, MA, X0
  VLE32.V (A0), V8
  VLE32.V (A1), V9
  VADD.VV V8, V9, V8
  VSE32.V (A2), V8
  RET
```

Thank You!

References

1. https://riscv.org/about/history
2. https://github.com/riscvarchive/riscv-go
3. https://en.wikipedia.org/wiki/RISC-V
4. https://docs.google.com/document/d/1xxRGuISUKN6oNNazQwCUOWTnnG8NrZMMeOYhtqX8qJ0
5. https://github.com/golang/go/issues/73787
6. https://medium.com/@jjuou2/advanced-debugging-and-the-address-sanitizer-8d6232127f53
7. 《深入Go语言之旅》