

# GHC's RISC-V Native Code Generation Backend

---

- Haskell Implementors' Workshop 2025
- Sven Tennie

# RISC-V Overview

# RISC-V

- 32bit **R**educed **I**nstruction **S**et as base
  - RV32I *Base Integer Instruction Set* → ~40 instructions, ~6 formats
  - Basic interpreter can be built in an afternoon
- Augmented by many extensions (sub-standards)
  - ISA like playing with Lego bricks
- Custom extensions are anticipated by the ISA
- Ideal research vehicle for computer architectures

# RISC-V

- **ISA is open source**, implementations (SOCs) not necessarily
  - License: *Creative Commons Attribution 4.0 International*
  - Development on GitHub
  - Vibrant community
  - Conceptualization in working groups at *RISC-V International* foundation
    - Free membership for individuals
- Everyone is free to build a RISC-V processor:
  - Several vendors
  - Hobbyists

# RISC-V Status

- Standard (ISA, Calling Convention, ...) pretty complete
- Lack of powerful hardware
  - No high performance cloud options → No native cloud CI (for us)
    - Most projects use a mixture of Docker & Qemu → Too slow for us
  - Cores comparable to ARM A55 (2017) or A72 (2016)
    - Speed market their *LicheePi 4A* as slightly slower than a *Raspberry Pi 4B*
    - Your smartphone might be more powerful than RISC-V SBCs

# RISC-V Status

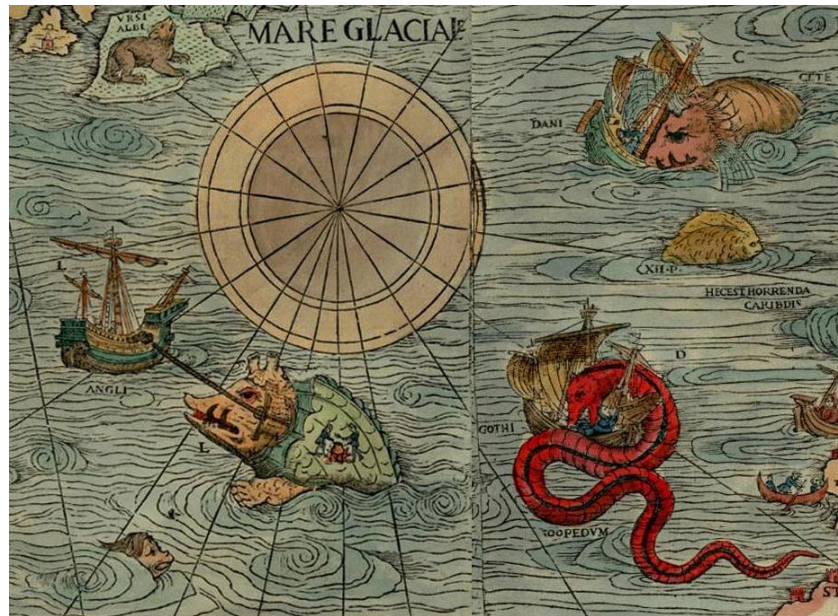
- Lot's of movement though
  - New boards and chips appear frequently
  - Many manufacturers
  - Research all over the world
    - EU grant for RISC-V HPC research
      - DARE (*Digital Autonomy with RISC-V in Europe*)
      - Funding: ~240 Million Euros
    - SHAKTI CPU by IIT-Madras (India)
    - many more

# RISC-V Status

- Here be dragons ...
  - Tools don't support the full instruction set
  - Tools sometimes still have bugs ...
  - Cores may have bugs
  - Core may not adhere to the ratified standards because it pre-dates it

## Warning

Use latest releases and be very precise about the hardware and build target!



Detail of Olaus Magnus's Carta Marina (1539), Wikimedia Commons

# ISA naming scheme

- Clang emits for `-march=rv64g` → `rv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zifencei2p0_zmmul1p0_zaamo1p0_zalrsc1p0` as target in Assembly
  - Base Integer ISA: `rv64i` (64bit)
  - Extension versions: `<extension-name><major>p<minor>`
  - Expansion:
    - `G` → `IMAFDZicsr_Zifencei`
    - `M` → `MZmmul`
    - `F` → `FZicsr`
    - `A` → `AZaamoZalrsc`



# Profiles

- *Profiles* (e.g. RVA23) define minimum requirements to simplify this
  - Otherwise, buying and building for a consumer computer could be a nightmare
  - (It still is, because many vendors don't mention profiles yet on their marketing pages)
  - Linux distributions handle this by relying on a small extension set (usually *RV64GC*)
    - *G*: General
    - *C*: Compressed instructions

# GHC Implementation Status

# GHC RISC-V History

- LLVM backend by Andreas Schwab (merged 2020; GHC 9.2)
- 2023: Moritz Angerman and Sven Tennie accidentally started implementing NCG at the same time
  - Moritz switched to mentor role
  - Sven continued to hack
  - Andreas built CI support at SuSE with patch files
    - <https://build.opensuse.org/package/show/openSUSE:Factory/ghc>
- Available from **GHC 9.12** (August 2024)



## Tip

### Reach out and team up

- I wouldn't have imagined that such great collaboration between former strangers would be possible.

# GHC RISC-V status

- LLVM Backend
- RTS Linker
- Native Code Generation Backend
  - Fullfills whole testsuite (minus SIMD tests)
- Tier 3 platform
  - Due to lack of powerful hardware (CI), there are no official binary distributions, yet
  - Probably not much in use, yet
    - Happy to receive bug reports!
- SIMD (Vector) in NCG support WIP

# Vector (SIMD) Support

# Vector Register Configuration

- Problem: Applications need very different vector sizes
  - Embedded chips should save silicon
  - HPC may need big vectors
  - usually a tradeoff
  - usually max vector sizes are bound to ISA features
  - RISC-V ISA allows 32 (*Zv32b*) to 65,536 bits per vector register

# Vector Register Configuration

- RISC-V approach:
  1. Make effective register width configurable → **grouping**
    - Combine multiple vector registers to one effective
  2. Tell when a configuration doesn't fit → **strip mining**
    - Iterate over vector chunks
- Benefits:
  - Application can dynamically react on the vector register width (VLEN)
  - HPC software can run on embedded CPUs and vice versa without recompilation

# Vector Register Configuration Instruction(s)

```
vsetivli <VL>, <AVL>, <SEW>, <LMUL>, <tail>, <mask>
```

- **VL** : New, effective **V**ector **L**ength (in elements)
- **AVL** : **A**pplication **V**ector **L**ength
  - The desired VL
- **SEW** : **S**ingle **E**lement **W**idth
  - Width of an element: **e8** , **e16** , **e32** , **e64** (bits)
- **LMUL** : **L**ength **M**ultiplier
  - **mf8** (LMUL=1/8), **mf4** (LMUL=1/4), **mf2** (LMUL=1/2)
  - **m1** (LMUL=1), **m2** (LMUL=2), **m4** (LMUL=4), **m8** (LMUL=8)



# Vector configuration - Grouping

- Increment each element of a *8bit* x *8* vector by one (128bit register width)

```
void plus_one(uint8_t b[8]) {  
    for(int i = 0; i < 8; i++) {  
        b[i]++;  
    }  
}
```

- `mf2` grouping:  $1/2 * 128 = 64$
- required bits:  $8 * 8 = 64$

```
plus_one:  
    vsetivli zero, 8, e8, mf2, ta, ma  
    # Load v8 as 8-bit elements at address in a0  
    vle8.v v8, (a0)  
    # v8[i] = v8[i] + 1  
    vadd.vi v8, v8, 1  
    # Store to address in a0  
    vse8.v v8, (a0)  
    ret
```

# Vector configuration - Grouping (2)

- Increment each element of a *8bit x 16* vector by one (128bit register width)

```
void plus_one(uint8_t b[16]) {  
    for(int i = 0; i < 16; i++) {  
        b[i]++;  
    }  
}
```

- `m1` grouping:  $1 * 128 = 128$
- required bits:  $8 * 16 = 128$

```
plus_one:  
    vsetivli zero, 16, e8, m1, ta, ma  
    # Load v8 as 8-bit elements at address in a0  
    vle8.v v8, (a0)  
    # v8[i] = v8[i] + 1  
    vadd.vi v8, v8, 1  
    # Store to address in a0  
    vse8.v v8, (a0)  
    ret
```

# Vector configuration - Grouping (3)

- Increment each element of a *8bit*  $\times$  32 vector by one (128bit register width)

```
void plus_one(uint8_t b[32]) {  
    for(int i = 0; i < 32; i++) {  
        b[i]++;  
    }  
}
```

- `m2` grouping:  $2 * 128 = 256$
- required bits:  $8 * 32 = 256$

```
plus_one:  
    # 32 doesn't fit into an immediate, use a register  
    li a1, 32  
    vsetvli zero, a1, e8, m2, ta, ma  
    # Load v8 as 8-bit elements at address in a0  
    vle8.v v8, (a0)  
    # v8[i] = v8[i] + 1  
    vadd.vi v8, v8, 1  
    # Store to address in a0  
    vse8.v v8, (a0)  
    ret
```

# Vector configuration - Strip-Mining

- Increment each element of a  $8\text{bit} \times 32$  vector by one (128bit register width)

```
void plus_one(uint8_t b[32]) {  
    for(int i = 0; i < 32; i++) {  
        b[i]++;  
    }  
}
```

- Iterations (after `vsetvli`):

- `t0` = 16; `a1` = 32; `a0` = `&b[0]` = `b`
- `t0` = 16; `a1` = 16; `a0` = `&b[16]` = `b + 16`

```
plus_one:  
    # Start with 32 elements  
    li a1, 32  
  
loop:  
    # Configure to get the real VL (16) in t0  
    vsetvli t0, a1, e8, m1, ta, ma  
    # Perform computation on chunk  
    vle8.v v8, (a0)  
    vadd.vi v8, v8, 1  
    vse8.v v8, (a0)  
    # Update pointers and counters for next chunk  
    # Move pointer forward: a0 += VL  
    add a0, a0, t0  
    # Reduce remaining elements (a1 -= VL)  
    sub a1, a1, t0  
    # Repeat if there are remaining elements  
    bnez a1, loop  
  
end:  
    ret
```

# Vector configuration - Strip-Mining (2)

- Increment each element of a  $8\text{bit} \times 17$  vector by one (128bit register width)

```
void plus_one(uint8_t b[17]) {  
    for(int i = 0; i < 17; i++) {  
        b[i]++;  
    }  
}
```

- Iterations (after `vsetvli`):

- `t0 = 16; a1 = 17; a0 = &b[0] = b`
- `t0 = 1; a1 = 1; a0 = &b[16] = b + 16`

```
plus_one:  
    # Start with 17 elements  
    li a1, 17  
  
loop:  
    # Configure to get the real VL in t0  
    vsetvli t0, a1, e8, m1, ta, ma  
    # Perform computation on chunk  
    vle8.v v8, (a0)  
    vadd.vi v8, v8, 1  
    vse8.v v8, (a0)  
    # Update pointers and counters for next chunk  
    # Move pointer forward: a0 += VL  
    add a0, a0, t0  
    # Reduce remaining elements (a1 -= VL)  
    sub a1, a1, t0  
    # Repeat if there are remaining elements  
    bnez a1, loop  
  
end:  
    ret
```

# Vectors: Questions to investigate

- How can we allocate register groups? (Virtual registers that cover multiple consecutive registers)
  - This would require the register allocator to be aware of grouped registers
- Would it be better to apply strip-mining (if the VLEN is too small)?
  - Would that work for all `MachOp`s?

## Note

The first edition of SIMD / vectors support in NCG will:

- have a GHC parameter for a minimum VLEN (vector register width)
- expect the machine to have at least that VLEN
- panic when a bigger vector is requested

# Vectors: Questions to investigate (2)

- How to optimize for minimal vector re-configuration?
  - My naive approach is to:
    - fold over the final instructions in each block in the Assembly emitting stage ( `Ppr.hs` )
      - keep the current vector configuration in the accumulator (state)
      - drop configuration statements that would set (duplicate) the current state
      - drop the state at jumps
  - This ignores optimizations by moving (re-ordering) instructions with the same configuration requirements.

# General future tasks

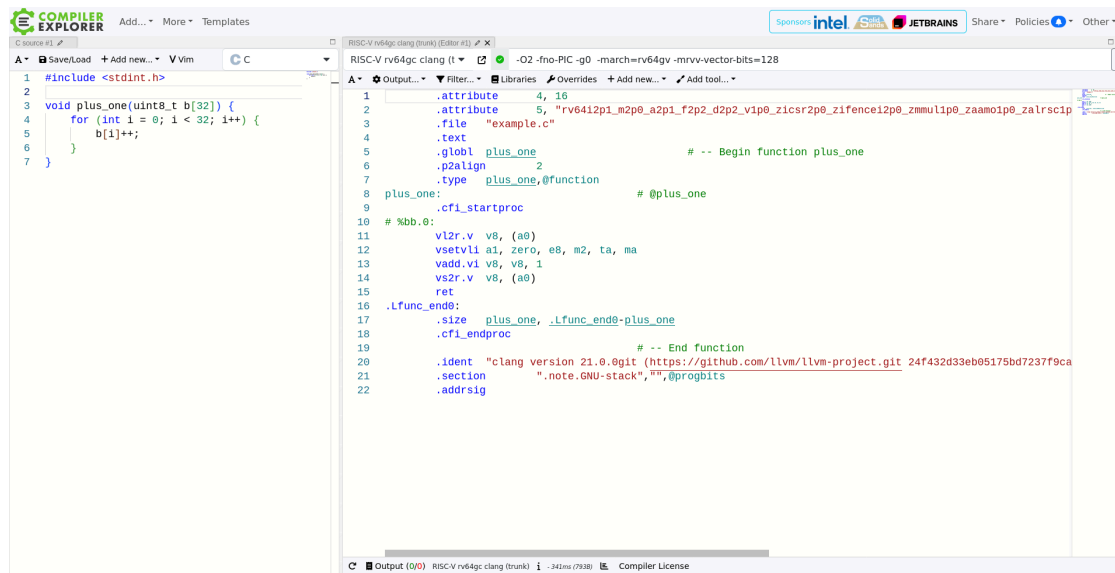
- Investigate ISA standard extensions beyond RV64GV
  - Good candidates may be:
    - **B**: Extension for Bit Manipulation
    - **Zicond**: Extension for Integer Conditional Operations
- Let GHC understand the target machine string
  - The *naming scheme* we discussed in the beginning
  - Make additional extensions optional
- Check if applying GADTs couldn't make NCGs saver
  - The current pattern is often "(pattern) match or panic"



# NCG development: Tipps & Tricks

# Compiler Explorer (Godbolt)

- <https://godbolt.org>
- Learn from others
- C and LLVM IR are good choices
- Intrinsics are a typed way to play with Assembly



The screenshot displays the Compiler Explorer (Godbolt) interface. The left pane shows the source code in C, and the right pane shows the generated assembly code for the RISC-V rv64gc clang (trunk) compiler.

**Source Code (C):**

```
1 #include <stdint.h>
2
3 void plus_one(uint8_t b[32]) {
4     for (int i = 0; i < 32; i++) {
5         b[i]++;
6     }
7 }
```

**Assembly Code (RISC-V rv64gc clang (trunk)):**

```
1      .attribute      4, 16
2      .attribute      5, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_v1p0_zicsr2p0_zifence12p0_zmmul1p0_zaamo1p0_zalrsc1p
3      .file           "example.c"
4      .text
5      .globl plus_one      # -- Begin function plus_one
6      .p2align         2
7      .type            plus_one,@function
8      plus_one:
9      .cfi_startproc      # @plus_one
10     # %bb.0:
11     vl2r.v v8, (a0)
12     vsetvli a1, zero, e8, m2, ta, ma
13     vadd.vi v8, v8, 1
14     vs2r.v v8, (a0)
15     ret
16 .lfunc_end0:
17 .size plus_one, .lfunc_end0-plus_one
18 .cfi_endproc
19
20 # -- End function
21 .ident "clang version 21.0.0git (https://github.com/llvm/llvm-project.git 24f432d33eb85175bd7237f9ca
22 .section ".note.GNU-stack","",@progbits
23 .addrsig
```

# ghc.nix

- <https://gitlab.haskell.org/ghc/ghc.nix>
- Nix env to build GHC
  - Cross-compiler envs possible

```
cd $MY_GHC_SRC_DIR
nix develop "git+https://gitlab.haskell.org/ghc/ghc.nix#riscv64-linux-cross"
./boot && configure_ghc
```

- More convenient with `direnv` `.envrc` file
  - `direnv` automatically provides the environment when you change into the directory

```
use flake git+https://gitlab.haskell.org/ghc/ghc.nix\#riscv64-linux-cross
```

# Run tests emulated with Qemu

- Most tests can be executed with an emulator (e.g. Qemu)
  - You don't have access to real hardware
  - Your workstation is faster
  - ...

```
CROSS_EMULATOR=qemu-riscv64 hadrian/build -j --docs=none --flavour=devel2 test
```

- `-fexternal-interpreter` and `-pgmi` can be used to run tests with Template Haskell

# test-primops

- <https://gitlab.haskell.org/ghc/test-primops>
- QuickCheck tests for GHC's C-- pipeline and code generators
- Generates Cmm expressions and compares your GHC's to an interpreter's results
  - Cross possible
  - Great to verify your NCG changes/implementation



Tip

test-primops doesn't test all MachOps (CmmExprs). Adding (some of) them would be a great newcomers' task.

# Build GHC and libs with LLVM

- Focus on small bits: Don't attempt to write a whole NCG in one go

- Build GHC itself and libraries with `-fllvm`

- Build tests with `-fasm`

- `EXTRA_HC_OPTS=-fasm hadrian/build test ...`

- `hadrian` provides:

- a flavour transformer `<your-flavour>+llvm`

- a flavour that uses LLVM `quick-cross`

# Reduce problems

```
{-# LANGUAGE MagicHash, GHCForeignImportPrim, UnliftedFFITypes #-}
module Main where

import GHC.Exts

foreign import prim "f5149" f :: Int# -> Int# -> Double# -> Int#

main = print (I# (f 1# 2# 1.0##))
```

```
f5149 (W_ x, W_ y, D_ z)
{
  ... // details don't matter here
}
```

(T5149 test)

- Reading a lot of Assembly or Cmm can be very exhausting
- Write small Cmm reproducers by hand
  - E.g. write a small Haskell driver and call it via FFI
  - Examples of this are in the testsuite

# Reduce problems (2)

- Adjust tests
  - Focus on one test / feature at a time
  - Build the smallest reproducer possible
    - Often tests can be simplified by deleting some code
  - Add dump options to test config:
    - `-ddump-to-file -dppr-debug -ddump-cmm -ddump-asm`
    - Run `hadrian` with `-k` to keep those files
- Run testsuite subsets with `hadrian`
  - `--only="test1 test2"` or `--test-root-dirs="./testsuite/tests/..."`



# Your are not alone!

- Matrix group (with IRC bridge): <https://matrix.to/#/#GHC:matrix.org>
- Mailing list: <https://mail.haskell.org/cgi-bin/mailman/listinfo/ghc-devs>
- Discourse: <https://discourse.haskell.org>

# Hunting Heisenbugs

- Bugs that disappear when you "look" at them
  - Trace logs and debuggers (GDB) change the timing of programs and execution at CPU-level
- Trace instructions and/or CPU state with Qemu
  - `qemu-riscv64 -d in_asm,cpu -one-insn-per-tb`
- My worst Heisenbug was a missing memory barrier (after program cache flush, `fence.i` instruction) in the linker
  - Illegal instruction exceptions at weird places
  - Gone when the timing changed e.g. by adding trace logs
  - Staring at Qemu traces gives hints what happened shortly before
    - Though, it slows the execution down immensely!

# AI / LLMs

- Large Language Models are pretty good in explaining Assembly code

