

Bounded Load/Stores in Grammar-based Code Generation for Testing the RISC-V Vector Extension

Manfred Schlägl

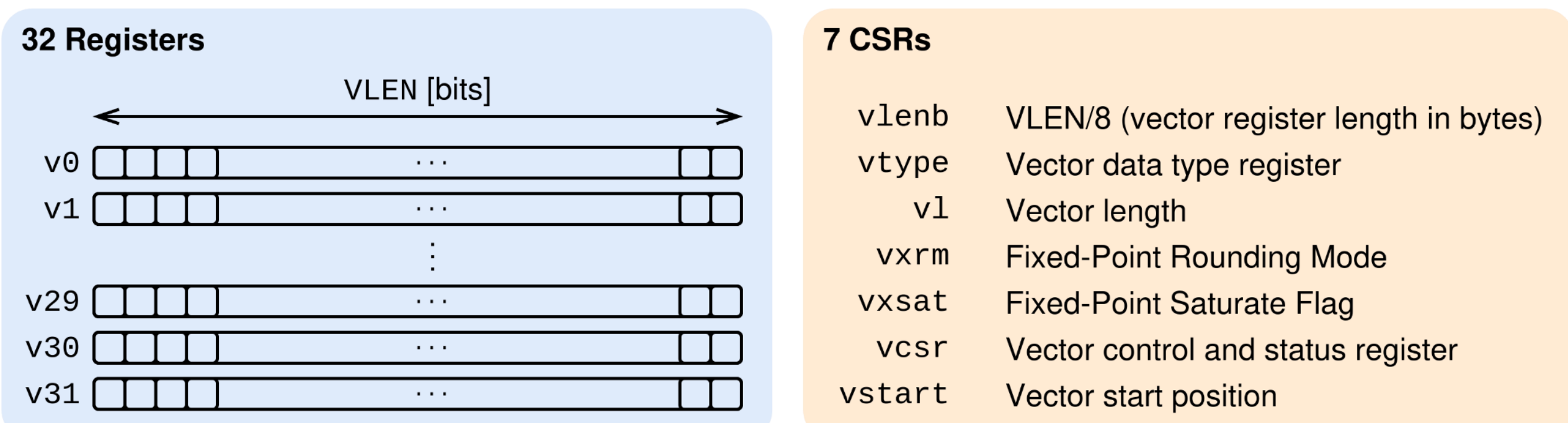
Daniel Große

Institute for Complex Systems, Johannes Kepler University Linz
manfred.schlaegl@jku.at, daniel.grosse@jku.at



Introduction

- RISC-V open-standard and modular ISA
 - Very small base ISA (RV32I / RV64I)
 - Various extensions (e.g. Mult/Div, Float, Compressed, ...)
- RISC-V "V" Vector Extension (RVV)
 - Motivation: Exploiting data-level parallelism** → Working on multiple elements of data (=vector) simultaneously to increase data throughput



624 Instructions		
Configuration	Integer	Reductions, Permutations
Load/Store (Strided, Indexed, etc)	Fixed-Point Floating-Point	Widening / Narrowing Masking

- Extensive *Single Instruction, Multiple Data* (SIMD) capabilities
- Instructions act on vectors in dedicated registers (**v0 - v31**)
- Function of instructions highly dependent on configuration (**vl, vtype**)
- Powerful set of load/store instructions

Very extensive and complex RISC-V extension → Verification challenging

RISC-V VP++ with RVV [1]

- Virtual Prototypes* (VPs): Binary compatible executable SW models of real HW
- RISC-V VP++**
 - Free- and open-source SystemC TLM based RISC-V VP
 - RV32/RV64, Exceptions/Interrupts, MMU, Peripherals
 - Configurations: uC/Application-processor based, Single-/Multi-core
 - RVV supported in all configurations
 - Graphics output, Mouse/Keyboard input via VNC
 - Linux support → GUI-VP Kit [2]

Verification of RVV in RISC-V VP++

Approach presented in original Paper [1]

- Instruction Sequence Generator* (ISG) *FORCE-RISCV*
- Trace comparison *Simulator under Test* (SUT) with *Reference Simulator* (REF)
- **81.44% coverage** → **Limited by ISG and missing coverage feedback**

New Approach:

Grammar-based ISG + Coverage-guided + Machine state comparison
→ **>94% coverage**

Context-free ISG Grammar for RVV

```
1 RVVGrammar = {
2   "<start>": ["<instr_v_config>", "<instr_v_compute>", ... ],
3   ...
4   "<instr_v_compute>": ["<instr_v_vector_int>", ... ],
5   "<instr_v_vector_int>": ["vadd<.vv>", "vadd<.vx>", "vadd<.vi>", ... ],
6   ...
7   "<.vv>": ["<.vvd>", "<.vs2>", "<.vs1><.vm>"],
8   "<.vx>": ["<.vxd>", "<.vs2>", "<.rs1><.vm>"],
9   "<.vi>": ["<.vid>", "<.vs2>", "<.imm5><.vm>"],
10  "<.vm>": ["", "<.v0.t>"],
11  ...
12  "<.vd>": ["<.vreg>"],
13  "<.vs1>": ["<.vreg>"],
14  "<.vs2>": ["<.vreg>"],
15  "<.vreg>": ["<.v0>", "<.v1>", ... "<.v31>"],
16  ...
17 }
```

- Non-terminal symbols:** pointed brackets (e.g. "<start>")
- Terminal symbols:** without pointed brackets (e.g. "<.v0.t>")
- Each line is an *expansion rule* = A list of expansion alternatives
- Expansion: Select alternative randomly, repeat until no *non-terminal sym.* left
- **Examples:** "vadd.vv v2, v3, v4", "vadd.vx v0, v3, x3, v0.t", ...

Resources



RISC-V VP++



Bounded Load/Store Paper

Bounded Load/Stores

Context-free grammar is able to generate RVV load/stores

Example: RVV unit-stride store of vector register **v1** to memory at address in **x5**

```
vse8.v v1, (x5) // RVV unit-stride store
```

Problem: Value of **x5** likely not pointing in a valid address range (esp. RV64)

Solution: Bounding of address values → Generate code to ensure valid **x5**

Problem: Bounding not efficiently expressible in context-free grammar

Solution: Extending grammar with *function symbols* (context-free behavior)

```
1 RVVGrammar = {
2   "<start>": ["<instr_v_config>", "<instr_v_load_store>", ... ],
3   ...
4   "<instr_v_load_store>": ["<instr_v_load>", "<instr_v_store>"],
5   ...
6   "<instr_v_store>": ["<instr_v_store_vse8>", ... ],
7   "<instr_v_store_vse8>": gen_v_store_vse8,
8   ...
9 }
```

Expansion of **<instr_v_store_vse8>** → call of **gen_v_store_vse8**

```
1 # Global values (allowed to change while code generation)
2 <VALID_START> = start address of valid area
3 <VALID_LEN> = length of valid area
4 <MAX_STORE_LEN> = maximum number of bytes in a vector (VLENB * 8)
5
6 # Bounded vse8 generation function (pseudocode)
7 def gen_v_store_vse8():
8   <ireg_rs1> = select random integer register
9   <vreg_vd> = select random vector register
10  <ireg_scratch> = select rand integer register other than <ireg_rs1>
11
12 # mask for upper bound
13 <upper_bound_mask> = 1 << (log2(<VALID_LEN> - <MAX_STORE_LEN>) - 1)
14 # offset to add to meet lower bound
15 <lower_bound_offs> = <VALID_START>
16
17 # ensure address below upper bound by masking
18 code = li <ireg_scratch>, <upper_bound_mask>
19 code += and <ireg_rs1>, <ireg_rs1>, <ireg_scratch>
20
21 # ensure address above lower bound by addition
22 code += li <ireg_scratch>, <lower_bound_offs>
23 code += add <ireg_rs1>, <ireg_rs1>, <ireg_scratch>
24
25 # generate store
26 code += vse8.v <vreg_vd>, (<ireg_rs1>)
27
28 # return generated code
29 return code
```

Example: Generated code for **vse8.v**

VALID_START = 0x801a0000; VALID_LEN = 1.5 MiB; MAX_STORE_LEN = 512 B

```
li x9, 0xffff // set upper_bound_mask
and x5, x5, x9 // apply upper_bound_mask
li x9, 0x801a0000 // set lower_bound_offset
add x5, x5, x23 // apply lower_bound_offset
vse8.v v1, (x5) // RVV unit-stride store
```

→ RVV unit-stride store bounded to memory area [0x801a000:0x811A1FF]

Progress & Outlook

RVV contains powerful load/store instructions supporting complex data structures (e.g. strided, indexed)

→ **Concept is applicable to all RVV load/store instructions**

→ **Implementations for all RV32/64I and RVV load/store instructions included in our new verification framework**

Will be presented and released as open-source later this year

References

[1] M. Schlägl, M. Stockinger, and D. Große, "A RISC-V "V" VP: Unlocking vector processing for evaluation at the system level," in DATE, 2024, pp. 1-6.

[2] M. Schlägl and D. Große, "GUI-VP Kit: A RISC-V VP meets Linux graphics - enabling interactive graphical application development," in GLSVLSI, 2023, pp. 599-605.