RISC-V Tracer

Sanoj S Vijendra[*] Alberto Ros[†] Ricardo Fernandez Pascual[‡]
Juan Manuel Cebrian Gonzalez[§] Jose Ruben Titos[¶]
Manuel Eugenio Sanchez[‖] Sawan Singh[**]

12 July, 2024

[*]Pursuing Bachelors at IIT Bombay
[†]Professor at University of Murcia
[‡]Professor at University of Murcia
[§]Professor at University of Murcia
[¶]Professor at University of Murcia
[‖]Professor at University of Murcia
[**]Pursuing Ph.D. at University of Murcia

Contents

1 Introduction

Computer Architecture is a specification detailing how a set of software and hardware technology standards interact to form a computer system or platform. In short, computer architecture refers to how a computer system is designed and what technologies it is compatible with.

It encompasses several key components, including the **central processing unit (CPU)** that performs most of the processing inside the computer, **memory** that stores data and instructions, **input/output (I/O)** devices for interaction with the user, and the **bus** that provides a communication system to connect all components.

There are three categories of computer architecture:

1. **System Design**, which includes hardware components.
2. **Instruction Set Architecture (ISA)**, which interfaces between the hardware and software.
3. **Microarchitecture**, which describes the data path, processors, memory, and how they should be implemented in the ISA.

2 Instruction Set Architecture (ISA)

Instruction Set Architecture (ISA) is a part of the computer architecture that defines the machine code instructions that a processor can execute. It serves as the boundary between software and hardware, providing a consistent interface that abstracts away the underlying microarchitecture's details.

ISAs encompass aspects such as data types, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. They specify the set of commands that a processor can understand and execute.

There are several ISAs available but they generally fall into two major categories:

1. **Complex Instruction Set Computing (CISC):** CISC architectures aim to minimize the number of instructions per program, sacrificing the number of cycles per instruction. The main idea is that a single instruction will do all loading, evaluating, and storing operations just like a multiplication command will do stuff like loading data, evaluating, and storing it, hence it's complex. Common example of CISC which is widely used in PCs is **x86**.
2. **Reduced Instruction Set Computing (RISC):** RISC architectures aim to minimize the computational load on each instruction, optimizing the number of cycles per instruction. The main idea behind this is to simplify hardware by using an instruction set composed of a few basic steps for loading, evaluating, and storing operations just like a load command will load data, a store command will store the data. Instructions used are simple, so instruction decoding is also simple. Common example includes **ARM** (Advanced RISC Machines) and **MIPS**.

3 RISC-V ISA

RISC-V Logo

RISC-V (pronounced “risk-five”) is an open standard instruction set architecture (ISA) based on established reduced instruction set computing (RISC) principles.

RISC-V includes a *base integer* ISA, which must be present in any implementation, along with optional extensions for more complex arithmetic operations. The ISA has been designed with small, fast, and low-power real-world implementations in mind, but without over-architecting for a particular microarchitecture style.

The key features of *RISC-V* include a small standard base with variable-length instruction encoding and modular, optional extensions. It also allows for custom extensions and custom coprocessors. It has a load-store architecture, where data operations can only act on registers. For more information about *RISC-V*, click here.

4 Why RISC-V?

RISC-V, an open standard instruction set architecture, is rapidly gaining traction in the world of computing. Its design principles offer a blend of simplicity, efficiency, and flexibility that make it a compelling choice for a wide range of applications. Here are some key advantages that *RISC-V* brings to the table:

- **Open Source:** As an open-source ISA, *RISC-V* is freely available for use, modification, and distribution, fostering innovation and rapid development.
- **Modular Design:** The *RISC-V ISA* is modular, meaning it has a small base of instructions that can be extended with optional modules. This allows for a high degree of customization and scalability.
- **Simplicity and Efficiency:** In line with RISC principles, *RISC-V* uses a simple and efficient instruction set, enabling fast execution and efficient use of resources.
- **Customizability:** *RISC-V* allows for custom extensions. This means that designers can add custom instructions to optimize their hardware for specific tasks or applications.
- **Strong Community Support:** The open nature of *RISC-V* has led to a strong, vibrant community of developers and users, providing a wealth of resources and collaborative opportunities.

5 Our Aim

The objective of our project is to ascertain the number of instructions executed by programs utilizing the *RISC-V ISA*. We aim to identify the various types of operations occurring within the program. A key goal is to enhance efficiency through the use of **vectorization**, and to subsequently test the effectiveness of this vectorization. To accomplish this, we will generate a trace using a simulator, such as **Spike**, to which we will add specific functionalities.

6 Vector Extension

The **Vector Extension**, often referred to as the '**V-Extension**', is standard extension in *RISC-V* designed to provide vector computation capabilities to the RISC-V ecosystem. It is designed to reduce instruction bandwidth, lower energy consumption, and expose data-level parallelism (DLP).

The Vector Extension introduces **32 vector registers**. Each vector register can optionally have an associated type, allowing for polymorphic encoding. The number of registers is variable and can be dynamically changed. The semantics of vector instructions are controlled by the Vector Length (VL) register. All instructions can be executed under a mask, providing an intuitive memory ordering model. Precise exceptions are supported.

The Vector Extension includes all instructions present in the baseline ISA. Vector memory instructions support linear, strided, and gather/scatter access patterns. Optional sets for fixed-point and transcendental operations are also included. For further information, click here.

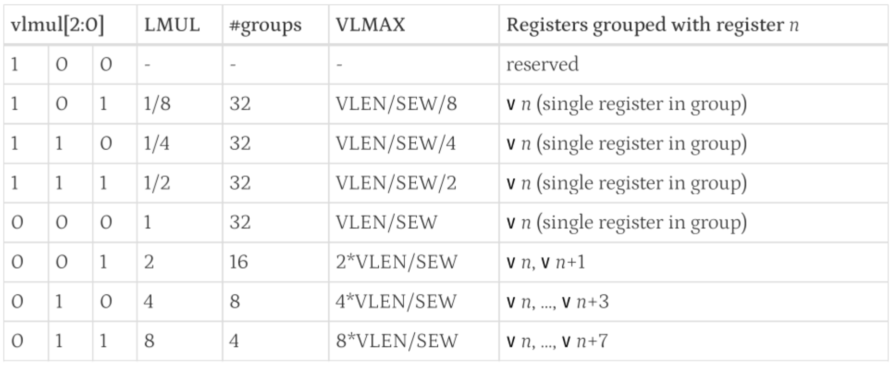
6.1 Vectoriztion in RISC-V

Vectorization in *RISC-V* involves the use of vector instructions that operate on variable-sized vectors, whose length and element size can be set at runtime. This approach exploits application parallelism through deeply pipelined datapaths and single instruction, multiple data (SIMD) computation. The design of a variable-length vector instruction set allows the instructions to be agnostic to the vector register size of a specific CPU implementation.

6.2 Terminologies

Some basic terminologies :

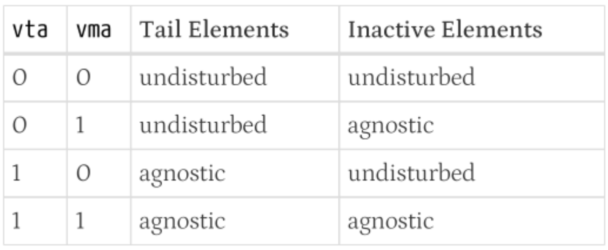
1. **Vector Registers:** The vector extension adds 32 architectural vector registers, **v0-v31** to the base scalar *RISC-V ISA*. Each vector register has a fixed *VLEN* bits of state.
2. **Vector Length (VLEN):** It signifies the number of bits in the vector registers.
3. **Vector Selected Element Width (vsew):** It determines the number of elements per vector. Each vector register is viewed as VLEN/vsew elements, each of which vsew-bits wide.
4. **Vector Length Multiplier (vlmul):** Multiple vector registers can be grouped together, so that a single vector instruction can operate on multiple vector registers. The term vector register group is used herein to refer to one or more vector registers used as a single operand to a vector instruction. The main reason for their inclusion is to allow double-width or larger elements to be operated on with the same vector length as single-width elements.



vlmul[2:0]			LMUL	#groups	VLMAX	Registers grouped with register *n*
1	0	0	-	-	-	reserved
1	0	1	1/8	32	VLEN/SEW/8	v *n* (single register in group)
1	1	0	1/4	32	VLEN/SEW/4	v *n* (single register in group)
1	1	1	1/2	32	VLEN/SEW/2	v *n* (single register in group)
0	0	0	1	32	VLEN/SEW	v *n* (single register in group)
0	0	1	2	16	2*VLEN/SEW	v *n*, v *n*+1
0	1	0	4	8	4*VLEN/SEW	v *n*, ..., v *n*+3
0	1	1	8	4	8*VLEN/SEW	v *n*, ..., v *n*+7

VLMUL Table

5. **Tail Agonistic (vta) and Mask Agonistic (vma):** These two bits modify the behavior of destination tail elements and destination inactive masked-off elements respectively during the execution of vector instructions.



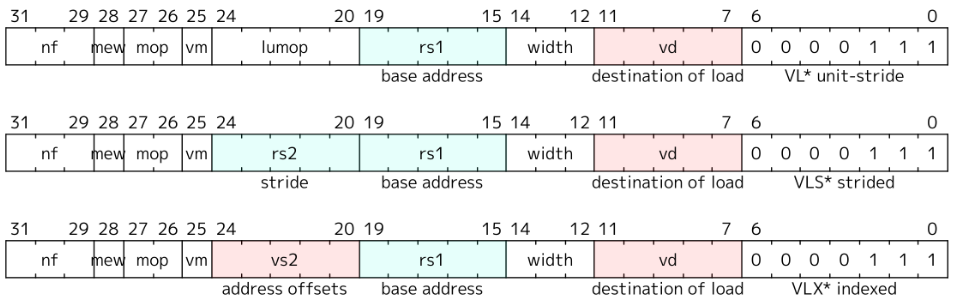
vta	vma	Tail Elements	Inactive Elements
0	0	undisturbed	undisturbed
0	1	undisturbed	agnostic
1	0	agnostic	undisturbed
1	1	agnostic	agnostic

vta & vma

6. **Vector Masking:** Masking is supported on many vector instructions. Element operations that are **masked off** (inactive) **never generate exceptions**. The destination vector register elements corresponding to masked-off elements are handled with either a mask-undisturbed or mask-agnostic policy depending on the setting of the vma bit in vtype. The mask value used to control execution of a masked vector instruction is always supplied by vector register **v0**.

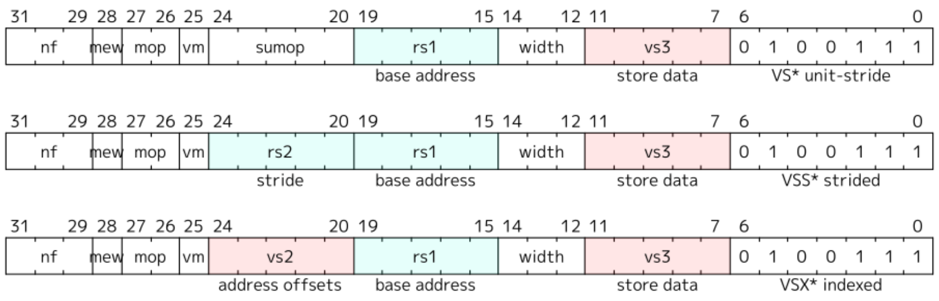
6.3 Vector Instruction Formats

6.3.1 Vector Load Instructions



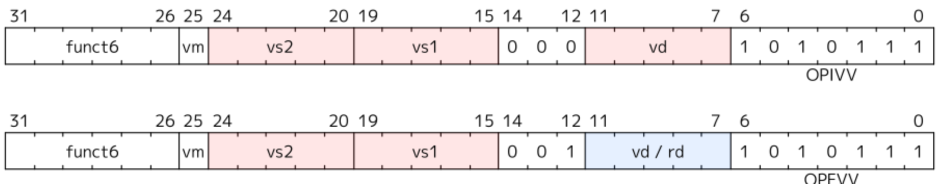
Load Instruction Decoding

6.3.2 Vector Store Instructions


31 29 28 27 26 25 24 20 19 15 14 12 11 7 6 0
nf | mew | mop | vm | sumop | rs1 | width | vs3 | 0 1 0 0 1 1 1
base address | store data | VS* unit-stride
nf | mew | mop | vm | rs2 | rs1 | width | vs3 | 0 1 0 0 1 1 1
stride | base address | store data | VSS* strided
nf | mew | mop | vm | vs2 | rs1 | width | vs3 | 0 1 0 0 1 1 1
address offsets | base address | store data | VSX* indexed

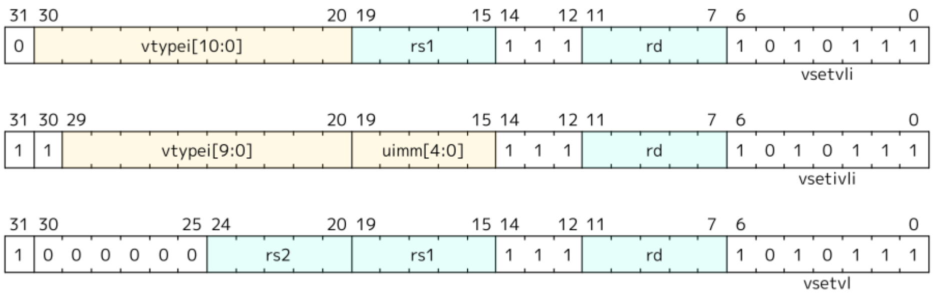
Store Instruction Decoding

6.3.3 Vector Arithmetic Instructions


31 26 25 24 20 19 15 14 12 11 7 6 0
funct6 | vm | vs2 | vs1 | 0 0 0 | vd | 1 0 1 0 1 1 1
OPIVV
funct6 | vm | vs2 | vs1 | 0 0 1 | vd / rd | 1 0 1 0 1 1 1
OPFVV

Arithmetic Instruction Decoding

6.3.4 Vector Configuration Instructions


31 30 20 19 15 14 12 11 7 6 0
0 | vtypei[10:0] | rs1 | 1 1 1 | rd | 1 0 1 0 1 1 1
vsetvli
31 30 29 20 19 15 14 12 11 7 6 0
1 | 1 | vtypei[9:0] | uimm[4:0] | 1 1 1 | rd | 1 0 1 0 1 1 1
vsetivli
31 30 25 24 20 19 15 14 12 11 7 6 0
1 | 0 0 0 0 0 0 | rs2 | rs1 | 1 1 1 | rd | 1 0 1 0 1 1 1
vsetvl

Configuration Instruction Decoding

7 RISC-V Tracer and Spike Simulator

Tracer which we are using is based on spike simulator. It generates trace information in certain format as specified in the code files.

7.1 What it can do?

This tracer can help you in the following:

1. Generate the trace for the in-house simulator.
2. Understand the dynamic instructions in RISC-V ISA.
3. Help you get the stats for various types of instructions.
4. It can help you add new instructions to RISC-V ISA and test it.

There are several more features present in the tracer.

7.2 What it can't?

Our tracer is based on Spike which is a RISC-V ISA emulator and thus it does not provide any timing information. Also, it can not be used to get performance improvements.

7.3 Installation and Dependencies

7.3.1 Dependencies

- RISC-V Toolchain
- RISC-V Proxy Kernel (pk) (*Optional*)
- RISC-V ISA Simulator (rss-spike-sdk) (*Optional*)

7.3.2 Installation

By following steps given below, we can install and build the tracer (assuming RISC-V toolchain is already installed):
Change the working directory to the directory of RISC-V tracer.

```
(sudo) apt-get install device-tree-compiler
mkdir build | cd build
../configure --prefix=$RISCV  (note: $RISCV is the installation path of RISC-V toolchain)
make -jX  (here X is number of CPU to be used)
(sudo) make install
```

7.4 Generating Traces

7.4.1 Using Proxy Kernel (pk)

1. Generate the executable binary by compiling your code using RISC-V Toolchain.
2. Change the working directory to RISC-V Tracer.
3. Follow the commands given below:

```
cd build
./spike --isa=rv64gcv --varch=vlen:512,elen:64 --trace-usermode --trace-location [trace-
    location] [pk-location] [executable-binary-location]
```

7.5 Trace Format

Let's take a simple *C++* code :

```
#include <cstdio>
#include <cstdlib>
#include <cmath>
static inline __attribute__((always_inline)) void Start_ROI()
{
    __asm__ volatile("srai zero, zero, 0");
}

static inline __attribute__((always_inline)) void End_ROI()
{
    __asm__ volatile("srai zero, zero, 1");
}
const int n = 100000;
double a[n], b[n], c[n];
double k = 7.23;
int main(){
  for(int i = 0 ; i < n ; i++) {  // Initialize
     a[i] = 0.2;
     b[i] = 0.2;
     c[i] = 0;
  }
  Start_ROI();
  for(int i = 0 ; i < n ; i++)  { // ROI
     c[i] = a[i] + b[i] * k;
  }
  End_ROI();
  return 0;
}
```

So, trace generated to this above *C++* code (for region of interest **(ROI)**) will be:

```
104F0
L0x3z45 7d8b0 8
4x12z15
L4x14z47 7dc10 8
L4x13z46 7d8f0 8
4x13z13 4x14z14 A4x45x46x47z47 4x15z15
S4x15x47 7df30 8
B4x10x13t66808*
L-28x14z47 7dc18 8
L4x13z46 7d8f8 8
4x13z13 4x14z14 A4x45x46x47z47 4x15z15
S4x15x47 7df38 8
B4x10x13t66808*
.
.
.
L-28x14z47 7df28 8
L4x13z46 7dc08 8
4x13z13 4x14z14 A4x45x46x47z47 4x15z15
S4x15x47 7e248 8
B4x10x13t66808
END 10514
```

7.6 Trace Mapping

- L : load
- LA : atomic load
- LE : load with intent to modify
- S : store
- SA : atomic store
- RMW : read-modify-write
- B : conditional branch (direct)
- C : call direct
- c : call indirect
- J : jump direct
- j: jump indirect
- r : return (indirect)
- A : fp/vector_addsub
- M : fp/vector_mul
- D : fp/vector_div
- Q : fp/vector_sqrt
- [] : generic
- d : reg dependence
- m : mem dependence
- a : addr dependence
- t : target address
- x : src register
- z : dst register
- * : taken (only for conditional)
- CLEAR : point for reseting stats
- ACQ : lock acquire
- REL : lock release
- BAR : barrier
- CV_SIGNAL : conditional variable signal
- CV_BCAST : conditional variable broadcast
- CV_WAIT : conditional variable wait
- e : number of elements (for vector ins. only)
- s : size of each element (for vector ins. only)
- + : stride length

7.7 Vector Instructions added to Tracer

Several vector instructions have been added to the tracer. Some of them are:

1. vsetvli : It sets both vector configuration and vector length. It sets the vl to the $min(VLMAX, asked_length)$.

   ```
   vsetvli rd, rs, eX, mY, ma, ta
   ```

   Here, rs will store the requested vector length, rd will store $min(VLMAX, rs)$. eX is the element length in bits (X can be 32, 64, 128 ...). mY is the length multiplier which can have values from 1/8 to 8 (fractional values are represented using mf for eg; 1/4 is $mf4$). $maandta$ are mask agnostic and tail agnostic values.
2. Vector Loads and Stores (along with its several variants) : Standard vector loads/stores along with strided, gather/scatter, faulty-first and segmented.

   ```
   vle64.v vd, (rs1), vm
   vse64.v vs3, (rs1), vm
   ```

3. Vector addition/multiplication/division and square roots.

   ```
   vadd.vv vd, vs2, vs1, vm
   vfadd.vv vd, vs2, vs1, vm
   ```

4. Vector Single-Width Integer Multiply-Add along with their different variants are added.

   ```
   vmadd.vv vd, vs1, vs2, vm
   ```

5. Vector Single-Width Floating-Point Fused Multiply-Add Instructions along with their different variants.

   ```
   vfmadd.vv vd, vs1, vs2, vm
   ```

6. Vector and/or/Xor instructions have been added. (along with variants).

   ```
   vand.vv vd, vs2, vs1, vm
   vor.vv vd, vs2, vs1, vm
   vxor.vv vd, vs2, vs1, vm
   ```

7. Vector Element Index Instruction

   ```
   vid.v vd, vm
   ```

8. Vector Integer Move Instructions (along with variants).

   ```
   vmv.v.v vd, vs1 # vd[i] = vs1[i]
   ```

9. Different comparison instructions in vector.
10. Vector Shift Instructions (along with variants).
11. Vector integer extension (along with variants).
12. Vector Integer Merge Instruction.
13. Whole Vector Register Move (along with variants).
14. Single-Width Floating-Point/Integer Type-Convert Instructions (along with variants).
15. Widening/Narrowing Floating-Point/Integer Type-Convert Instructions (along with variants).
16. Vector Slide Instructions (along with variants).
17. Vector Single-Width Floating-Point Reduction Instructions (along with variants).

7.8 Additional Functionality

Along with the trace, *RISC-V* Tracer can also produce log file which contains the count of total instructions executed and in ROI, count of vector instructions, vector loads/stores, branches and branches in ROI.

8 Examples and Results

To view some of the examples and results, visit my github page.

9 References

1. The RISC-V Instruction Set Manual Volume I
2. Geeks for Geeks
3. RISC-V Official Website
4. RISC-V Toolchain
5. RISC-V Spike ISA
6. RISC-V CAPS