An Introduction to RISC-V

Compute Accelerator Forum

Jonas Hahnfeld, EP-SFT

September 13, 2023

What is RISC-V? (on one slide)

- Open Standard Instruction Set Architecture (ISA)
  - Specifications are open source, no royalty fees
  - RISC-V cores can be open or proprietary

What is RISC-V? (on one slide)

- Open Standard Instruction Set Architecture (ISA)
  - Specifications are open source, no royalty fees
  - RISC-V cores can be open or proprietary

- Started at the University of California, Berkley, in 2010
- Since 2020 published by RISC-V International located in Switzerland

What is RISC-V? (on one slide)

- Open Standard Instruction Set Architecture (ISA)
  - Specifications are open source, no royalty fees
  - RISC-V cores can be open or proprietary

- Started at the University of California, Berkley, in 2010
- Since 2020 published by RISC-V International located in Switzerland

- Modular design: base ISA with very few (integer) instructions
  - Many standard extensions and possibility for custom instructions

Background

History of RISC-V

The RISC-V ISA
- Base Integer Instruction Set(s)
- Standard Extensions

RISC-V Vector Extension (RVV)

Conclusions

Background

RISC and CISC

- RISC = Reduced Instruction Set Computer
  - Term coined by David Patterson of the Berkley RISC project (1980s)
  - Very “simple” instructions, that can be executed fast(er)

RISC and CISC

- RISC = Reduced Instruction Set Computer
  - Term coined by David Patterson of the Berkley RISC project (1980s)
  - Very "simple" instructions, that can be executed fast(er)

- CISC = Complex Instruction Set Computer
  - (Some) instructions perform many operations
  - e.g. memory load, arithmetic operation, memory store

RISC and CISC

- RISC = Reduced Instruction Set Computer
  - Term coined by David Patterson of the Berkley RISC project (1980s)
  - Very "simple" instructions, that can be executed fast(er)

- CISC = Complex Instruction Set Computer
  - (Some) instructions perform many operations
  - e.g. memory load, arithmetic operation, memory store (cf. `add [rdi], 2`)

RISC and CISC

- RISC = Reduced Instruction Set Computer
  - Term coined by David Patterson of the Berkley RISC project (1980s)
  - Very "simple" instructions, that can be executed fast(er)
  - Notable architectures today: Arm, Power, RISC-V
- CISC = Complex Instruction Set Computer
  - (Some) instructions perform many operations
  - e.g. memory load, arithmetic operation, memory store (cf. `add [rdi], 2`)

RISC and CISC

- RISC = Reduced Instruction Set Computer
  - Term coined by David Patterson of the Berkley RISC project (1980s)
  - Very "simple" instructions, that can be executed fast(er)
  - Notable architectures today: Arm, Power, RISC-V
- CISC = Complex Instruction Set Computer
  - (Some) instructions perform many operations
  - e.g. memory load, arithmetic operation, memory store (cf. `add [rdi], 2`)
  - Notable architectures today: x86, z/Architecture (IBM Z mainframe)

RISC and CISC – the meaning of "Reduced"

- Does *not* mean "few instructions"
  - Example: Armv9 has 402 "base instructions", 404 "SIMD and FP instructions", 765 "SVE instructions"...

RISC and CISC – the meaning of "Reduced"

- Does *not* mean "few instructions"
  - Example: Armv9 has 402 "base instructions", 404 "SIMD and FP instructions", 765 "SVE instructions"...
- Does *not* mean only "simple operations"
  - Example: instructions for FP square root in Arm and RISC-V
  - Arm even has instructions for hashing (SHA1, SHA256, SHA512)!
    - (RISC-V has them in the `Zk` Standard Extension for Scalar Cryptography)

RISC and CISC – the meaning of "Reduced"

- Does *not* mean "few instructions"
  - Example: Armv9 has 402 "base instructions", 404 "SIMD and FP instructions", 765 "SVE instructions"...
- Does *not* mean only "simple operations"
  - Example: instructions for FP square root in Arm and RISC-V
  - Arm even has instructions for hashing (SHA1, SHA256, SHA512)!
    - (RISC-V has them in the `Zk` Standard Extension for Scalar Cryptography)
- Nowadays mostly refers to "load-store architectures"

Load-Store Architectures

- Principle of load-store architectures:
  - Instructions either load from or store to memory,
  - *OR* perform operations between registers
  - (hence also sometimes called “register-register architecture”)

Load-Store Architectures

- Principle of load-store architectures:
  - Instructions either load from or store to memory,
  - *OR* perform operations between registers
  - (hence also sometimes called “register-register architecture”)

- On the contrary, x86 is a “register-memory architecture”

Load-Store Architectures

- Principle of load-store architectures:
  - Instructions either load from or store to memory,
  - *OR* perform operations between registers
  - (hence also sometimes called "register-register architecture")

- On the contrary, x86 is a "register-memory architecture"

- Compare:
  - x86: `add DWORD PTR [rdi], 2`

Load-Store Architectures

- Principle of load-store architectures:
  - Instructions either load from or store to memory,
  - *OR* perform operations between registers
  - (hence also sometimes called "register-register architecture")

- On the contrary, x86 is a "register-memory architecture"

- Compare:
  - x86: `add DWORD PTR [rdi], 2`
  - RISC-V:

```
lw      a5,0(a0)
addiw   a5,a5,2
sw      a5,0(a0)
```

History of RISC-V

The Beginnings

- Started in May 2010 at the Par Lab at UC Berkley
  - Prof. Krste Asanović, graduate students Yunsup Lee and Andrew Waterman
  - Joined by Prof. David Patterson, Director of Par Lab
    - (who coined the term RISC in 1980s during the Berkley RISC project!)
  - Fifth generation RISC ISA design at Berkley

The Beginnings

- Started in May 2010 at the Par Lab at UC Berkley
  - Prof. Krste Asanović, graduate students Yunsup Lee and Andrew Waterman
  - Joined by Prof. David Patterson, Director of Par Lab
    - (who coined the term RISC in 1980s during the Berkley RISC project!)
  - Fifth generation RISC ISA design at Berkley

- First publication of RISC-V instruction set in May 2011:
  - The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA

The Beginnings

- Started in May 2010 at the Par Lab at UC Berkley
  - Prof. Krste Asanović, graduate students Yunsup Lee and Andrew Waterman
  - Joined by Prof. David Patterson, Director of Par Lab
    - (who coined the term RISC in 1980s during the Berkley RISC project!)
  - Fifth generation RISC ISA design at Berkley
- First publication of RISC-V instruction set in May 2011:
  - The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA
- Later that month, May 2011: first tapeout of a RISC-V chip in 28nm

RISC-V Foundation and RISC-V International

- RISC-V Foundation launched in 2015
  - Steer development and ratification of RISC-V by its members
  - Freely publish the RISC-V ISA documents for unrestricted use

RISC-V Foundation and RISC-V International

- RISC-V Foundation launched in 2015
  - Steer development and ratification of RISC-V by its members
  - Freely publish the RISC-V ISA documents for unrestricted use
- Decision in November 2019 to relocate to Switzerland
  - After reflections of geo-political landscape, to "alleviate uncertainty going forward"

RISC-V Foundation and RISC-V International

- RISC-V Foundation launched in 2015
  - Steer development and ratification of RISC-V by its members
  - Freely publish the RISC-V ISA documents for unrestricted use
- Decision in November 2019 to relocate to Switzerland
  - After reflections of geo-political landscape, to "alleviate uncertainty going forward"
- RISC-V International Association incorporated in Switzerland in March 2020

Current Status of the Standard

- Unprivileged ISA first ratified and frozen in December 2019
  - Included “Base Integer Instruction Sets” and some standard extensions

Current Status of the Standard

- Unprivileged ISA first ratified and frozen in December 2019
  - Included "Base Integer Instruction Sets" and some standard extensions
- A number of extensions ratified in 2021
  - For bit manipulation, half-precision floating-point, vector

Current Status of the Standard

- Unprivileged ISA first ratified and frozen in December 2019
  - Included "Base Integer Instruction Sets" and some standard extensions
- A number of extensions ratified in 2021
  - For bit manipulation, half-precision floating-point, vector
- Some more ratifications in 2022 and 2023
  - Multiply without divide, reduced integer bases, total store ordering

Current Status of the Standard

- Unprivileged ISA first ratified and frozen in December 2019
  - Included "Base Integer Instruction Sets" and some standard extensions
- A number of extensions ratified in 2021
  - For bit manipulation, half-precision floating-point, vector
- Some more ratifications in 2022 and 2023
  - Multiply without divide, reduced integer bases, total store ordering
- Important addition this year: Profiles
  - Group a base ISA with mandatory standard extensions plus options
  - Also includes expectations, for example regarding atomicity

Software Milestones

- First toolchain support in binutils 2.28 (March 2017) and GCC 7.1 (May 2017)

Software Milestones

- First toolchain support in binutils 2.28 (March 2017) and GCC 7.1 (May 2017)

- First Linux port merged in v4.15 (January 2018)
- Followed by userspace support in glibc 2.27 (February 2018)

Software Milestones

- First toolchain support in binutils 2.28 (March 2017) and GCC 7.1 (May 2017)
- First Linux port merged in v4.15 (January 2018)
- Followed by userspace support in glibc 2.27 (February 2018)
- Official Debian architecture since July 2023!

The RISC-V ISA

RV32I Base Integer Instruction Set

- 32 registers, `x0`–`x31`, 32 bits wide (`x0` hardwired to zero)
- 32-bit instruction encoding (except "C" extension, see later)

RV32I Base Integer Instruction Set

- 32 registers, `x0`–`x31`, 32 bits wide (`x0` hardwired to zero)
- 32-bit instruction encoding (except "C" extension, see later)

- 40 instructions:
  - 21x integer computation instructions (add, sub, shift, logical operations)
  - 8x control transfer instructions (unconditional jump, conditional branches)
  - 8x load and store instructions (word, half-word, byte)
  - Memory ordering (fence), environment call, and breakpoint instructions

RV64I Base Integer Instruction Set

- 32 registers widened to 64 bits

RV64I Base Integer Instruction Set

- 32 registers widened to 64 bits
- 15 new instructions:
  - 9x integer computation instructions on 32-bit words
  - 3x shift immediate instructions for 64 bits
  - 3x load and store instructions on doublewords

RV32E, RV64E, RV128I Base Integer Instruction Sets

- RV32E and RV64E are reduced versions of RV32I and RV64I
  - Only 16 general-purpose registers
  - Designed for microcontrollers in embedded systems

RV32E, RV64E, RV128I Base Integer Instruction Sets

- RV32E and RV64E are reduced versions of RV32I and RV64I
  - Only 16 general-purpose registers
  - Designed for microcontrollers in embedded systems
- RV128I extends RV64I to 128 bits
  - For future exploration, specification not frozen at the moment

“M” Standard Extension for Integer Multiplication and Division

- Integer multiplication and division

“M” Standard Extension for Integer Multiplication and Division

- Integer multiplication and division
- 8 / 13 new instructions for RV32 / RV64:
  - 4x integer multiplication (+ 1x multiplication of 32-bit words for RV64)
  - 4x integer division and remainder (+ 4x for 32-bit words for RV64)

“A” Standard Extension for Atomic Instructions

- Instructions for atomic operations:
  - Load-reserved/store-conditional instructions (LR/SC)
  - Atomic fetch-and-op memory instructions

“A” Standard Extension for Atomic Instructions

- Instructions for atomic operations:
  - Load-reserved/store-conditional instructions (LR/SC)
  - Atomic fetch-and-op memory instructions

- 11 / 22 new instructions:
  - 2x LR/SC instructions (+ 2x for doublewords on RV64)
  - 9x atomic memory operations (+ 9x for doublewords on RV64)

“A” Standard Extension for Atomic Instructions

- Instructions for atomic operations:
  - Load-reserved/store-conditional instructions (LR/SC)
  - Atomic fetch-and-op memory instructions

- 11 / 22 new instructions:
  - 2x LR/SC instructions (+ 2x for doublewords on RV64)
  - 9x atomic memory operations (+ 9x for doublewords on RV64)
  - Requirement: naturally aligned addresses

“A” Standard Extension for Atomic Instructions

- Instructions for atomic operations:
  - Load-reserved/store-conditional instructions (LR/SC)
  - Atomic fetch-and-op memory instructions
- 11 / 22 new instructions:
  - 2x LR/SC instructions (+ 2x for doublewords on RV64)
  - 9x atomic memory operations (+ 9x for doublewords on RV64)
  - Requirement: naturally aligned addresses
- Atomic memory operations:
  - Atomically swap, add, and, or, xor two values
  - Atomically compute signed / unsigned maximum / minimum of two values

“A” Standard Extension for Atomic Instructions – LR/SC loops

- RISC-V chooses LR/SC loops over compare-and-set (CAS) instructions
  - Claimed to be a better fit and more efficient, see standard for rationale

"A" Standard Extension for Atomic Instructions – LR/SC loops

- RISC-V chooses LR/SC loops over compare-and-set (CAS) instructions
  - Claimed to be a better fit and more efficient, see standard for rationale
  - CAS can be implemented with LR/SC instructions:

```
    # a0 holds address of memory location
    # a1 holds expected value
    # a2 holds desired value
    # a0 holds return value, 0 if successful, !0 otherwise
cas:
    lr.w t0, (a0)       # Load original value.
    bne t0, a1, fail    # Doesn't match, so fail.
    sc.w t0, a2, (a0)   # Try to update.
    bnez t0, cas        # Retry if store-conditional failed.
    li a0, 0            # Set return to success.
    jr ra               # Return.
fail:
    li a0, 1            # Set return to failure.
    jr ra               # Return.
```

“F” Standard Extension for Single-Precision Floating-Point

- 32 registers, `f0-f31`, 32 bits wide

“F” Standard Extension for Single-Precision Floating-Point

- 32 registers, `f0-f31`, 32 bits wide
- 26 / 30 new instructions:
  - 2x load and store instruction
  - 7x floating point computation (add, sub, mul, div, sqrt, min, max)
  - 4x fused multiply-add instructions
  - 4x conversion instructions to / from integers (+ 4x to / from 64-bit integers)
  - 3x sign-injection instructions (copy, negate, xor)
  - 2x instructions to move bit patterns to / from general registers
  - 3x compare and 1x classify instructions

“D” Standard Extension for Double-Precision Floating-Point

- 32 floating point registers widened to 64 bits
  - Can hold both single- or double-precision values

“D” Standard Extension for Double-Precision Floating-Point

- 32 floating point registers widened to 64 bits
  - Can hold both single- or double-precision values
- 26 / 32 new instructions:
  - 13x analogous load/store/computational instructions
  - 4x analogous conversion instructions to / from int. (+ 4x to / from 64-bit int.)
  - 2x conversion instructions to / from single-precision
  - 3x analogous sign-injection instructions
  - 2x instructions to move bit patterns (only RV64)
  - 4x analogous compare / classify instructions

"G" = "General-Purpose" ISA

- Letter "G" used as abbreviation for: `IMAFDZicsr_Zifencei`

"G" = "General-Purpose" ISA

- Letter "G" used as abbreviation for: `IMAFDZicsr_Zifencei`
- Includes all standard extensions presented so far and:
  - `Zicsr`: Control and Status Register (CSR) Instructions
    - e.g. cycle and timing counters, hardware performance counters, floating-point CSR
  - `Zifencei`: Instruction-Fetch Fence

“G” = “General-Purpose” ISA

- Letter “G” used as abbreviation for: `IMAFDZicsr_Zifencei`
- Includes all standard extensions presented so far and:
  - `Zicsr`: Control and Status Register (CSR) Instructions
    - e.g. cycle and timing counters, hardware performance counters, floating-point CSR
  - `Zifencei`: Instruction-Fetch Fence
- Note: in the future Profiles probably are going to become more important

“C” Standard Extension for Compressed Instructions

- 16-bit encodings of commonly used instructions
  - Also lowers code alignment requirements for 32-bit encodings

“C” Standard Extension for Compressed Instructions

- 16-bit encodings of commonly used instructions
  - Also lowers code alignment requirements for 32-bit encodings
- Trade-offs:
  - Small immediate or address offset
    - Implying smaller target ranges for loads/stores/jumps/branches

“C” Standard Extension for Compressed Instructions

- 16-bit encodings of commonly used instructions
  - Also lowers code alignment requirements for 32-bit encodings
- Trade-offs:
  - Small immediate or address offset
    - Implying smaller target ranges for loads/stores/jumps/branches
  - Restrictions on the register (either special or most popular ones)

“C” Standard Extension for Compressed Instructions

- 16-bit encodings of commonly used instructions
  - Also lowers code alignment requirements for 32-bit encodings
- Trade-offs:
  - Small immediate or address offset
    - Implying smaller target ranges for loads/stores/jumps/branches
  - Restrictions on the register (either special or most popular ones)
- “50%–60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%–30% code-size reduction.”

More Standard Extensions

Unprivileged Architecture:

- `Q`: Quad-Precision Floating-Point
- `B`: Bit Manipulation
- `V`: Vector Operations (will talk about this a bit more)
- `Zk`: Scalar Cryptography
- `Zihintpause`: Pause Hint
- `Ztso`: Total Store Ordering

Even More Standard Extensions

Unprivileged Architecture:

- `Zfh{,min}`: Half-Precision Floating-Point
- `Z{f,d}inx`: {Single,Double}-Precision Floating-Point in Integer Register
- `Zhinx{,min}`: Half-Precision Floating-Point in Integer Register
- `Zmmul`: Multiplication Subset of the M Extension

Privileged Architecture:

- `H`: Hypervisor
- `S`: Supervisor

Some Words on Naming and Non-Standard Extensions

- Standard ISA extensions
  - Single letter (e.g. M, A, F, D)
  - "Z" prefix followed by alphabetical name
    - Second letter conventionally indicates closest standard extension
    - For example `Zicsr` and `Zfh`

Some Words on Naming and Non-Standard Extensions

- Standard ISA extensions
  - Single letter (e.g. M, A, F, D)
  - "Z" prefix followed by alphabetical name
    - Second letter conventionally indicates closest standard extension
    - For example `Zicsr` and `Zfh`
- Non-standard extensions, for example by vendors:
  - "X" prefix followed by alphabetical name
  - Convention by toolchains: start with vendor name
    - For example `XVentanaCondOps` and various `XThead*`

RISC-V Vector Extension (RVV)

Motivation

Motivation

- Can it [RISC-V] run Doom?

Motivation

- Can it [RISC-V] run Doom? Yes.

Motivation

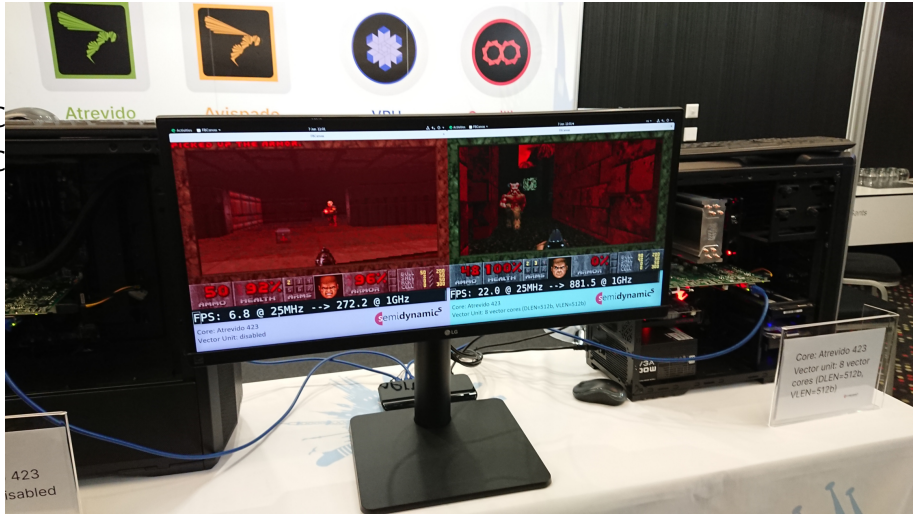
- Can it [RISC-V] run Doom? Yes.
- Can it run Doom faster with RISC-V Vector Extensions?

Motivation

- Can it [RISC-V] run Doom? Yes.
- Can it run Doom faster with RISC-V Vector Extensions? Yes.

Motivation

- C
- C



Motivation

- Can it [RISC-V] run Doom? Yes.
- Can it run Doom faster with RISC-V Vector Extensions? Yes.

- Some grains of salt:
  - FPGAs running at relatively low frequency of 25MHz
  - Hardware already fully optimized? Software fully optimized?
  - Manually inserting vector intrinsics

A Quick Look at SIMD Extensions in Other ISAs

- SIMD = Single Instruction, Multiple Data
  - To exploit Data-Level Parallelism, for example adding elements of two vectors

A Quick Look at SIMD Extensions in Other ISAs

- SIMD = Single Instruction, Multiple Data
  - To exploit Data-Level Parallelism, for example adding elements of two vectors
- Instructions available in SIMD extensions for current ISAs:
  - x86: Streaming SIMD Extensions (SSE), Advanced Vector Extensions (AVX)
  - Arm NEON

A Vector Loop: saxpy

- Single-precision vector operation: $\vec{y} \leftarrow a\vec{x} + \vec{y}$

```
void saxpy(int n, const float a, const float *x, float *y) {
    #pragma clang loop vectorize(assume_safety)
    #pragma clang loop interleave(disable)
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

[0] Clang #pragmas just to shorten the produced assembly code.

Code using SSE instructions (omitting setup code)

```
vectorized:
        movups  xmm2, xmmword ptr [rsi + r8]    # load x
        mulps   xmm2, xmm1                      # multiply with a (in xmm1)
        movups  xmm3, xmmword ptr [rdx + r8]    # load y
        addps   xmm3, xmm2                      # ax + y
        movups  xmmword ptr [rdx + r8], xmm3    # store y
        add     r8, 16              # compute next offset (+ 4 elements)
        cmp     rdi, r8             # compare to final offset to process
        jne     vectorized          # with vectorized loop (pre-computed)
        cmp     rcx, rax            # check if elements remain
        je      done                # otherwise done
scalar:
        movss   xmm1, dword ptr [rsi + 4*rcx]   # load x
        mulss   xmm1, xmm0                      # multiply with a
        addss   xmm1, dword ptr [rdx + 4*rcx]   # load and add y
        movss   dword ptr [rdx + 4*rcx], xmm1   # store y
        inc     rcx
        cmp     rax, rcx
        jne     scalar
done:   ret
```

Issues with "SIMD Instruction Set Extensions for Multimedia"

(according to "Computer Architecture - A Quantative Approach" by Hennessy and Patterson)

- Fixed-size registers: need new instructions for larger vectors
  - On x86: MMX (64 bits); SSE (128 bits); AVX (256 bits); AVX-512 (512 bits*)

Issues with "SIMD Instruction Set Extensions for Multimedia"

(according to "Computer Architecture - A Quantative Approach" by Hennessy and Patterson)

- Fixed-size registers: need new instructions for larger vectors
  - On x86: MMX (64 bits); SSE (128 bits); AVX (256 bits); AVX-512 (512 bits*)
- Partial solution: have "scalable" vector registers
  - Arm Scalable Vector Extensions (SVE)
  - Implementation-defined vector length from 128 bits to 2048 bits

Issues with "SIMD Instruction Set Extensions for Multimedia"

(according to "Computer Architecture - A Quantative Approach" by Hennessy and Patterson)

- Fixed-size registers: need new instructions for larger vectors
  - On x86: MMX (64 bits); SSE (128 bits); AVX (256 bits); AVX-512 (512 bits*)
- Partial solution: have "scalable" vector registers
  - Arm Scalable Vector Extensions (SVE)
  - Implementation-defined vector length from 128 bits to 2048 bits
- Related: how to deal with vectors that are not multiples of the register size?
  - Traditional approach: scalar remainder loop
  - Masking / predication in Arm SVE and x86 AVX-512 can help

Issues with "SIMD Instruction Set Extensions for Multimedia"

(according to "Computer Architecture - A Quantative Approach" by Hennessy and Patterson)

- Fixed-size registers: need new instructions for larger vectors
  - On x86: MMX (64 bits); SSE (128 bits); AVX (256 bits); AVX-512 (512 bits*)
- Partial solution: have "scalable" vector registers
  - Arm Scalable Vector Extensions (SVE)
  - Implementation-defined vector length from 128 bits to 2048 bits
- Related: how to deal with vectors that are not multiples of the register size?
  - Traditional approach: scalar remainder loop
  - Masking / predication in Arm SVE and x86 AVX-512 can help
- Natural solution developed in the 1960s/1970s used in the Cray-1
  - Have a vector-length register set by the application
  - Determines length for following vector operations

“V” Standard Extension for Vector Operations (RVV) – Registers

- 32 vector registers, `v0-v31`
  - Fixed vector length ($\texttt{VLEN} \leq 2^{16}$) bits of state (must be power of 2)

"V" Standard Extension for Vector Operations (RVV) – Registers

- 32 vector registers, `v0`–`v31`
  - Fixed vector length ($\texttt{VLEN} \leq 2^{16}$) bits of state (must be power of 2)
- Application configures vector type (`vtype`):
  - Selected element width ($\texttt{SEW} \geq 8$), how to interpret data
    - Must be smaller than hardware supported element length ($\texttt{ELEN} \geq 8$)

"V" Standard Extension for Vector Operations (RVV) – Registers

- 32 vector registers, `v0-v31`
  - Fixed vector length ($\texttt{VLEN} \leq 2^{16}$) bits of state (must be power of 2)

- Application configures vector type (`vtype`):
  - Selected element width ($\texttt{SEW} \geq 8$), how to interpret data
    - Must be smaller than hardware supported element length ($\texttt{ELEN} \geq 8$)
  - Vector register group multiplier ($\texttt{LMUL} \leq 8$)
    - For example, $\texttt{LMUL} = 8$ creates four groups of eight registers

"V" Standard Extension for Vector Operations (RVV) – Registers

- 32 vector registers, `v0-v31`
  - Fixed vector length ($\texttt{VLEN} \leq 2^{16}$) bits of state (must be power of 2)
- Application configures vector type (`vtype`):
  - Selected element width ($\texttt{SEW} \geq 8$), how to interpret data
    - Must be smaller than hardware supported element length ($\texttt{ELEN} \geq 8$)
  - Vector register group multiplier ($\texttt{LMUL} \leq 8$)
    - For example, $\texttt{LMUL} = 8$ creates four groups of eight registers
- Maximum number of elements: $\texttt{VLMAX} = \texttt{LMUL} * \texttt{VLEN}/\texttt{SEW}$
  - Indices guaranteed to fit in 16 bits, maximum $\texttt{VLMAX} = 8 * 2^{16}/8 = 65,536$
  - Selected vector length $\texttt{vl} \leq \texttt{VLMAX}$

RVV Configuration-Setting Instructions

- Three instructions to set `vl` and `vtype`: `vset{i}vl{i}`
- Focus on `vsetvli rd, rs1, vtypei`
  - `rs1`: application vector length (`AVL`), ie total number of elements to process
  - `vtypei`: immediate with new `vtype` setting (with assembler names)
  - `rd`: instruction returns new `vl` that was set

RVV Configuration-Setting Instructions

- Three instructions to set `vl` and `vtype`: `vset{i}vl{i}`
- Focus on `vsetvli rd, rs1, vtypei`
  - `rs1`: application vector length (`AVL`), ie total number of elements to process
  - `vtypei`: immediate with new `vtype` setting (with assembler names)
  - `rd`: instruction returns new `vl` that was set

- An example: `vsetvli t0, a0, e32, m8, ta, ma`
  - `a0` elements of `SEW` $= 32$ (for example single-precision FP)
  - `LMUL` $= 8$ to group eight registers each
  - (`ta, ma` sets vector mask and tail agnostic)
  - `t0` is assigned `vl` $\leq$ `VLMAX`, `vl` $\leq$ `AVL` (and some more constraints)

RVV Instructions

- Configuration-setting instructions
- Vector load and store instructions
- Vector integer arithmetic instructions
- Vector fixed-point arithmetic instructions
- Vector floating-point instructions
- Vector reduction operations
- Vector mask instructions
- Vector permutation instructions

Code using RVV (from the standard)

```
# register arguments:
#      a0       n
#      fa0      a
#      a1       x
#      a2       y

saxpy:
    vsetvli a4, a0, e32, m8, ta, ma     # ask for a0 elements, get a4
    vle32.v v0, (a1)                    # load x
    sub a0, a0, a4                      # subtract processed elements
    slli a4, a4, 2                      # shift by 2 = multiply by 4
    add a1, a1, a4                      # compute next pointer for x
    vle32.v v8, (a2)                    # load y
    vfmacc.vf v8, fa0, v0               # compute ax + y
    vse32.v v8, (a2)                    # store y
    add a2, a2, a4                      # compute next pointer for y
    bnez a0, saxpy                      # if elements left, jump back
    ret
```

Compiler Vectorization

- Manual vectorization is tedious, time-consuming, and error-prone

→ if possible, should rely on compiler to optimize the code

Compiler Vectorization

- Manual vectorization is tedious, time-consuming, and error-prone

→ if possible, should rely on compiler to optimize the code

- Autovectorization improved a lot in the last years
  - Works well for simple examples
  - Might need some help for more complex cases
  - Still not possible for many advanced cases

Compiler Vectorization

- Manual vectorization is tedious, time-consuming, and error-prone

→ if possible, should rely on compiler to optimize the code

- Autovectorization improved a lot in the last years
  - Works well for simple examples
  - Might need some help for more complex cases
  - Still not possible for many advanced cases

- RVV 1.0 currently only supported by upstream Clang
  - Old fork of GCC not updated anymore

saxpy for RV64GC with Clang

```
saxpy(int, float, float const*, float*):
        blez    a0, .LBB0_2
.LBB0_1:
        flw     ft0, 0(a1)
        flw     ft1, 0(a2)
        fmadd.s ft0, fa0, ft0, ft1
        fsw     ft0, 0(a2)
        addi    a2, a2, 4
        addi    a0, a0, -1
        addi    a1, a1, 4
        bnez    a0, .LBB0_1
.LBB0_2:
        ret
```

saxpy for RV64GCV with Clang (1/3)

```
saxpy(int, float, float const*, float*):
        blez    a0, done
        csrr    t1, vlenb           # get Vector Byte Length (VLEN/8)
        srli    t0, t1, 2           # divide by 4 = number of floats
        bgeu    a0, t0, vectorized_setup
        li      a7, 0
        j       scalar_setup
vectorized_setup:
        addi    a3, t0, -1                      # subtract 1 -> mask
        and     a6, a0, a3                      # remainder elements
        sub     a7, a0, a6                      # vectorizable elements
        vsetvli a3, zero, e32, m1, ta, ma       # request zero = max elements
        vfmv.v.f        v8, fa0                 # "splat" scalar value of a
        mv      a3, a7                          # vectorized elements left
        mv      a4, a2                          # address of y
        mv      a5, a1                          # address of x
vectorized:
        # see next slide
```

saxpy for RV64GCV with Clang (2/3)

```
vectorized:
        vl1re32.v       v9, (a5)        # load x
        vl1re32.v       v10, (a4)       # load y
        vfmacc.vv       v10, v8, v9     # compute ax + y
        vs1r.v  v10, (a4)               # store y
        add     a5, a5, t1              # compute next pointer for x
        sub     a3, a3, t0              # subtract processed elements
        add     a4, a4, t1              # compute next pointer for x
        bnez    a3, vectorized          # if elements left, jump back
        beqz    a6, done                # done if no remainder
scalar_setup:
        slli    a3, a7, 2               # offset after vectorized loop
        add     a2, a2, a3              # address of y
        add     a1, a1, a3              # address of x
        sub     a0, a0, a7              # subtract processed elements
scalar:
        # see next slide
```

saxpy for RV64GCV with Clang (3/3)

```
scalar:
        flw     ft0, 0(a1)              # load x
        flw     ft1, 0(a2)              # load y
        fmadd.s ft0, fa0, ft0, ft1      # compute ax + y
        fsw     ft0, 0(a2)              # store y
        addi    a2, a2, 4               # compute next pointer for y
        addi    a0, a0, -1              # subtract processed elements
        addi    a1, a1, 4               # compute next pointer for x
        bnez    a0, scalar              # if elements left, jump back
done:
        ret
```

Comments on `saxpy` for RV64GCV with Clang

- *Big* caveat: RVV 1.0 is still quite new!
  - Encouraging to see compiler support, even if suboptimal for now

Comments on `saxpy` for RV64GCV with Clang

- *Big* caveat: RVV 1.0 is still quite new!
  - Encouraging to see compiler support, even if suboptimal for now
- Generated code uses RVV with fixed-size vectors
  - Queries implementation-defined size of vector register (`VLEN`/8)
  - Likely reason: matches Arm Scalable Vector Extensions

Comments on `saxpy` for RV64GCV with Clang

- *Big* caveat: RVV 1.0 is still quite new!
  - Encouraging to see compiler support, even if suboptimal for now
- Generated code uses RVV with fixed-size vectors
  - Queries implementation-defined size of vector register (`VLEN`/8)
  - Likely reason: matches Arm Scalable Vector Extensions
- ... and scalar remainder loop
  - Probably does not yield optimal performance
  - Especially more remainder elements for bigger vector registers

Conclusions

Conclusions

- RISC-V is an Open Standard ISA

Conclusions

- RISC-V is an Open Standard ISA
- Modular design with base ISA and extensions

Conclusions

- RISC-V is an Open Standard ISA
- Modular design with base ISA and extensions
- RISC-V Vector Extensions with interesting design decisions
  - Portable code to future hardware with larger vectors

Conclusions

- RISC-V is an Open Standard ISA
- Modular design with base ISA and extensions
- RISC-V Vector Extensions with interesting design decisions
  - Portable code to future hardware with larger vectors
- First single board computers are there (e.g. VisionFive v2)
  - No hardware with RVV 1.0 *yet* (some with RVV 0.7.1)