VTx-family custom instructions

Custom ISA extensions for Ventana Micro Systems RISC-V cores

Version 1.0.1, 2022-08-07: Public release

Table of Contents

Front matter

Attributions

This document is based on the Asciidoctor documentation template designed at VRULL GmbH.
Typeset with Asciidoctor. Diagrams generated by Wavedrom.

The fonts used in this document are licensed under the Open Font License:

- M PLUS 1p,designed by the M+ Outline Fonts Project;
- Fira Sans, designed for Mozilla FirefoxOS by Carrois in Berlin, Germany;
- Unna, designed by Omnibus-Type in Buenos Aires, Argentina;
- SourceCodePro, designed by Paul D. Hunt for use in user interfaces.

Contributors

Written by Greg Favor (Ventana Micro Systems) and Philipp Tomsich (VRULL GmbH).
Production by VRULL GmbH.

Revision history

Version	Description
1.0.0	January 2022 Initial revision.
1.0.1	August 2022 Updated guidance for `conditional bitwise-and` sequences to expand to an `and` instruction followed by a `conditional bitwise-or` sequence, no longer depending on the Zbb extension (for the `andn` instruction).

Chapter 1. Vendor-prefix for mnemonics

To avoid conflicts between mnemonics defined by different implementors of custom RISC-V extensions, each vendor-defined mnemonic is—in consequence—prefixed by a vendor-specific prefix: i.e., each vendor-defined mnemonic is structured as "<*vendor prefix*> . <*mnemonic base name*>".

The vendor-prefix to all mnemonics defined by Ventana Micro Systems is "vt" (e.g., the “maskc” instruction becomes “vt.maskc”).

Chapter 2. Custom extensions

This document describes the custom ISA extensions defined by Ventana Micro Systems.

2.1. XVentanaCondOps ("Conditional Ops") Extension

One of the shortcoming of RISC-V, compared to competing instruction set architectures, is the absence of conditional operations to support branchless code-generation: this includes conditional arithmetic, conditional select and conditional move operations. The design principles or RISC-V (e.g. the absence of an instruction-format that supports 3 source registers and an output register) make it unlikely that direct equivalents of the competing instructions will be introduced.

Yet, low-cost conditional instructions are a desirable feature as they allow the replacement of branches in a broad range of suitable situations (whether the branch turns out to be unpredictable or predictable) so as to reduce the capacity and aliasing pressures on BTBs and branch predictors, and to allow for longer basic blocks (for both the hardware and the compiler to work with).

The "Conditional Ops" extension provides a simple solution that provides most of the benefit and all of the flexibility one would desire to support conditional arithmetic and conditional-select/move operations, while remaining true to the RISC-V design philosophy. The instructions follow the format for R-type instructions with 3 operands (i.e., 2 source operands and 1 destinantion operand). Using these instructions, branchless sequences can be implemented (typically in two-instruction sequenes) without the need for instruction fusion, special provisions during the decoding of architectural instrucitons, or other microarchitectural provisions.

The following instructions comprise the XVentanaCondOps extension:

RV32	RV64	Mnemonic	Instruction
n/a	✓	vt.maskc *rd*, *rs1*, *rs2*	Mask register value on condition
n/a	✓	vt.maskcn *rd*, *rs1*, *rs2*	Mask register value on negated condition

All current cores by Ventana Micro implement RV64 and are designed as 64-bit only, the RV32-column is marked "n/a". The instructions in the XVentanaCondOps extension are defined to operate on *XLEN* and would thus be directly applicable to RV32.

2.1.1. Applicability

Based on these two instructions, synthetic instructions (i.e., short instruction sequences) for the following **conditional arithmetic** operations are supported:

- conditional add, if zero
- conditional add, if non-zero
- conditional subtract, if zero
- conditional subtract, if non-zero
- conditional bitwise-and, if zero
- conditional bitwise-and, if non-zero
- conditional bitwise-or, if zero
- conditional bitwise-or, if non-zero
- conditional bitwise-xor, if zero
- conditional bitwise-xor, if non-zero

Additionally, the following **conditional select** intructions are supported:

- conditional-select, if zero
- conditional-select, if non-zero

Note that a **conditional move** is a degenerate version of the **conditional select** and can be built from these sequences.

2.1.2. Instruction sequences

Operation	Instruction sequence	Length
Conditional add, if zero `rd = (rc == 0) ? (rs1 + rs2) : rs1`	`vt.maskcn rd, rs2, rc` `add rd, rs1, rd`	2 insns
Conditional add, if non-zero `rd = (rc != 0) ? (rs1 + rs2) : rs1`	`vt.maskc rd, rs2, rc` `add rd, rs1, rd`	
Conditional subtract, if zero `rd = (rc == 0) ? (rs1 - rs2) : rs1`	`vt.maskcn rd, rs2, rc` `sub rd, rs1, rd`	
Conditional subtract, if non-zero `rd = (rc != 0) ? (rs1 - rs2) : rs1`	`vt.maskc rd, rs2, rc` `sub rd, rs1, rd`	
Conditional bitwise-or, if zero `rd = (rc == 0) ? (rs1 | rs2) : rs1`	`vt.maskcn rd, rs2, rc` `or rd, rs1, rd`	
Conditional bitwise-or, if non-zero `rd = (rc != 0) ? (rs1 | rs2) : rs1`	`vt.maskc rd, rs2, rc` `or rd, rs1, rd`	
Conditional bitwise-xor, if zero `rd = (rc == 0) ? (rs1 ^ rs2) : rs1`	`vt.maskcn rd, rs2, rc` `xor rd, rs1, rd`	
Conditional bitwise-xor, if non-zero `rd = (rc != 0) ? (rs1 ^ rs2) : rs1`	`vt.maskc rd, rs2, rc` `xor rd, rs1, rd`	

Conditional bitwise-and, if zero
`rd = (rc == 0) ? (rs1 & rs2) : rs1`
	`and rd, rs1, rs2`
	`vt.maskc rtmp, rs1, rc`
	`or rd, rd, rtmp`
		3 insns
		(requires 1 temporary)

Conditional bitwise-and, if non-zero
`rd = (rc != 0) ? (rs1 & rs2) : rs1`
	`and rd, rs1, rs2`
	`vt.maskcn rtmp, rs1, rc`
	`or rd, rd, rtmp`	

Conditional select, if zero
`rd = (rc == 0) ? rs1 : rs2`
	`vt.maskcn rd, rs1, rc`
	`vt.maskc rtmp, rs2, rc`
	`or rd, rd, rtmp`	

Conditional select, if non-zero
`rd = (rc != 0) ? rs1 : rs2`
	`vt.maskc rd, rs1, rc`
	`vt.maskcn rtmp, rs2, rc`
	`or rd, rd, rtmp`	

Note that a common benchmark benefiting from the conditional-xor sequence is EEMBC Coremark, where the CRC computation requires one conditional-xor per bit (i.e. a total of 16 conditional-xor operations for a CRC-16).

Listing 1. Conditional bitwise-xor use-case from EEMBC Coremark's core_util.c

```
if (crc & 1)
  crc ^= 0xa001;
```


Chapter 3. Supported extensions (by core)

This chapter lists the custom extensions supported by each core/core-family.

3.1. VT1 core

The VT1 core implements the following custom extensions:

- XVentanaCondOps: conditional operations

Chapter 4. Instructions (in alphabetical order)

This chapter lists all custom instructions defined by Ventana Micro Systems in alphabetical order.

4.1. vt.maskc

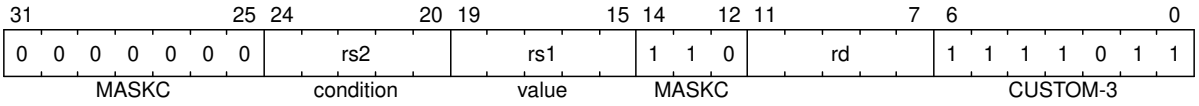
Synopsis

Masks a value (*rs1*) on condition of a truth value (*rs2*)

Mnemonic

vt.maskc *rd*, *rs1*, *rs2*

Encoding



Description

If the value of register *rs2* is non-zero, place the value of register *rs1* into the register *rd*.
Otherwise, put the value 0 (zero) into *rd*.

Operation

```
let value = X(rs1);
let condition = X(rs2);

X(rd) = if (condition != 0) then value
                            else 0;
```

Implemented in

Extension	Minimum version	Supported cores
XVentanaCondOps	1.0	VT1

4.2. vt.maskcn

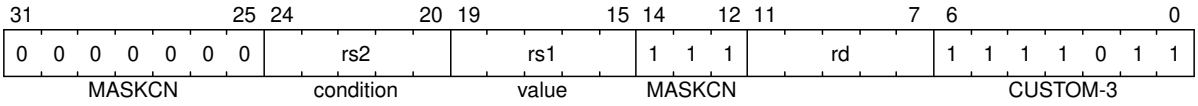
Synopsis

Masks a value (*rs1*) on the negated condition of a truth value (*rs2*)

Mnemonic

vt.maskc *rd*, *rs1*, *rs2*

Encoding



Description

If the value of register *rs2* is zero, place the value of register *rs1* into the register *rd*.
Otherwise, put the value 0 (zero) into *rd*.

Operation

```
let value = X(rs1);
let condition = X(rs2);

X(rd) = if (condition == 0) then value
                            else 0;
```

Implemented in

Extension	Minimum version	Supported cores
XVentanaCondOps	1.0	VT1

