



RISC-V Integer Conditional (Zicond) operations extension

Dr. Philipp Tomsich (VRULL GmbH)

Version 1.0, 2023-01-25: This document is in development. Assume everything can change. See
<http://riscv.org/spec-state> for details.

Table of Contents

Preamble.....	1
Copyright and license information.....	2
Contributors.....	3
1. Introduction.....	4
1.1. Suitability for Fast Track Extension Process	4
1.2. Motivation and use cases	4
2. Zicond proposed specification.....	5
3. Instructions (in alphabetical order).....	6
3.1. czero.eqz.....	7
3.2. czero.nez.....	9
4. Usage examples	11
4.1. Instruction sequences	11
4.2. Alternative sequences with data-invariant timing	11
Bibliography.....	?

Preamble



This document is in the [Development state](#)

Assume everything can change. This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Copyright and license information

This specification is licensed under the Creative Commons Attribution 4.0 International License (CC-BY 4.0). The full license text is available at creativecommons.org/licenses/by/4.0/.

Copyright 2022 by RISC-V International.

Contributors

This RISC-V specification has been contributed to directly or indirectly by:

- Dr. Philipp Tomsich <philipp.tomsich@vrull.eu>

Chapter 1. Introduction

The Zicnd extension defines 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 destination operand). Using these instructions, branchless sequences can be implemented (typically in two-instruction sequences) without the need for instruction fusion, special provisions during the decoding of architectural instructions, or other microarchitectural provisions.

1.1. Suitability for Fast Track Extension Process

This proposed extension meets the Fast Track criteria: it consists of two, simple R-form instructions, it addresses a wide range of use-cases for branchless sequences, it composes with the existing RISC-V instruction set, and is not expected to be contentious.

1.2. Motivation and use cases





One of the shortcomings 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 of 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).

Chapter 2. Zicond proposed specification

The "Conditional" operations 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 destination operand). Using these instructions, branchless sequences can be implemented (typically in two-instruction sequences) without the need for instruction fusion, special provisions during the decoding of architectural instructions, or other microarchitectural provisions.

The following instructions comprise the Zicond extension:

RV32	RV64	Mnemonic	Instruction
		<code>czero.eqz rd, rs1, rs2</code>	Conditional zero, if condition is equal to zero
		<code>czero.nez rd, rs1, rs2</code>	Conditional zero, if condition is nonzero



Architecture Comment: defining additional comparisons, in addition to equal-to-zero and not-equal-to-zero, does not offer a benefit due to the lack of immediates or an additional register operand that the comparison takes place against.

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** instructions are supported:

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

More complex conditions, such as comparisons against immediates, registers, single-bit tests, comparisons against ranges, etc. can be realized by composing these new instructions with existing instructions.

Chapter 3. Instructions (in alphabetical order)

3.1. czero.eqz

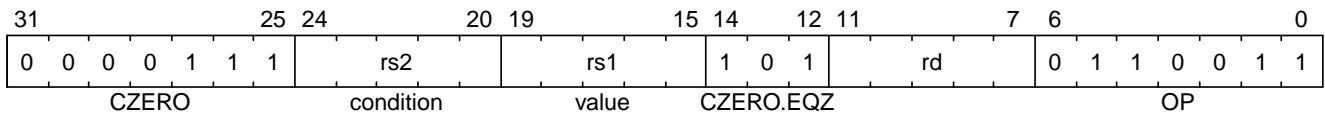
Synopsis

Moves zero to a register *rd*, if the condition *rs2* is equal to zero, otherwise moves *rs1* to *rd*.

Mnemonic

`czero.eqz rd, rs1, rs2`

Encoding



Description

This instruction behaves as if there is a conditional branch dependent on *rs2* being equal to zero, wherein it branches to code that writes a 0 into *rd* when the equivalence is true, and otherwise falls through to code that moves *rs1* into *rd*. Accordingly, the syntactic dependency on *rs1* is only propagated when the condition is false.

In effect, if the value of register *rs2* is zero, place 0 (zero) into the register *rd*; otherwise, place the value of register *rs1* into *rd*.



These branch-based semantics do not prevent implementing this instruction as a simple select (e.g., "(rs2==0) ? 0 : rs1"). Instead, they allow for more sophisticated implementations where a zero-result can be returned when the condition (rs2==0) is true without waiting for *rs1* to be available. Furthermore, implementations can predict the condition just as they might for branches.

As a consequence of this instruction's equivalence to `mv rd, rs1` when *rs2* is nonzero, this instruction's timing is independent of the data value of *rs1* if the Zkt extension is implemented.

SAIL code

```
let value = X(rs1);
let condition = X(rs2);
result : xlenbits = if (condition == zeros()) then zeros()
                    else value;
X(rd) = result;
```

Pseudocode

```
beqz rs2, 1f
mv    rd, rs1
j     2f
1:
mv    rd, zero
```

2:

3.2. czero.nez

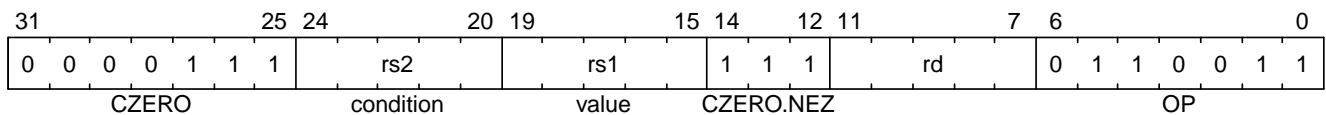
Synopsis

Moves zero to a register *rd*, if the condition *rs2* is nonzero, otherwise moves *rs1* to *rd*.

Mnemonic

`czero.nez rd, rs1, rs2`

Encoding



Description

This instruction behaves as if there is a conditional branch dependent on *rs2* being not equal to zero, wherein it branches to code that writes a 0 into *rd* when the equivalence is true, and otherwise falls through to code that moves *rs1* into *rd*. Accordingly, the syntactic dependency on *rs1* is only propagated when the condition is false.

In effect, if the value of register *rs2* is non-zero, place 0 (zero) into the register *rd*; otherwise, place the value of register *rs1* into *rd*.



These branch-based semantics do not prevent implementing this instruction as a simple select (e.g., "`rs2!=0` ? 0 : `rs1`"). Instead, they allow for more sophisticated implementations where a zero-result can be returned when the condition (`rs2!=0`) is true without waiting for *rs1* to be available. Furthermore, implementations can predict the condition just as they might for branches.

As a consequence of this instruction's equivalence to `mv rd, rs1` when *rs2* is zero, this instruction's timing is independent of the data value of *rs1* if the Zkt extension is implemented.

SAIL code

```
let value = X(rs1);
let condition = X(rs2);
result : xlenbits = if (condition != zeros()) then zeros()
                    else value;
X(rd) = result;
```

Pseudocode

```
bnez rs2, 1f
mv   rd, rs1
j    2f
1:
mv   rd, zero
```

2:

Chapter 4. Usage examples

The instructions from this extension can be used to construct sequences that perform conditional-arithmetic, conditional-bitwise-logical, and conditional-select operations.

4.1. Instruction sequences

Operation	Instruction sequence	Length
Conditional add, if zero <code>rd = (rc == 0) ? (rs1 + rs2) : rs1</code>	<code>czero.nez rd, rs2, rc</code> <code>add rd, rs1, rd</code>	2 insns
Conditional add, if non-zero <code>rd = (rc != 0) ? (rs1 + rs2) : rs1</code>	<code>czero.eqz rd, rs2, rc</code> <code>add rd, rs1, rd</code>	
Conditional subtract, if zero <code>rd = (rc == 0) ? (rs1 - rs2) : rs1</code>	<code>czero.nez rd, rs2, rc</code> <code>sub rd, rs1, rd</code>	
Conditional subtract, if non-zero <code>rd = (rc != 0) ? (rs1 - rs2) : rs1</code>	<code>czero.eqz rd, rs2, rc</code> <code>sub rd, rs1, rd</code>	
Conditional bitwise-or, if zero <code>rd = (rc == 0) ? (rs1 rs2) : rs1</code>	<code>czero.nez rd, rs2, rc</code> <code>or rd, rs1, rd</code>	
Conditional bitwise-or, if non-zero <code>rd = (rc != 0) ? (rs1 rs2) : rs1</code>	<code>czero.eqz rd, rs2, rc</code> <code>or rd, rs1, rd</code>	
Conditional bitwise-xor, if zero <code>rd = (rc == 0) ? (rs1 ^ rs2) : rs1</code>	<code>czero.nez rd, rs2, rc</code> <code>xor rd, rs1, rd</code>	
Conditional bitwise-xor, if non-zero <code>rd = (rc != 0) ? (rs1 ^ rs2) : rs1</code>	<code>czero.eqz rd, rs2, rc</code> <code>xor rd, rs1, rd</code>	
Conditional bitwise-and, if zero <code>rd = (rc == 0) ? (rs1 & rs2) : rs1</code>	<code>and rd, rs1, rs2</code> <code>czero.eqz rtmp, rs1, rc</code> <code>or rd, rd, rtmp</code>	3 insns (requires 1 temporary)
Conditional bitwise-and, if non-zero <code>rd = (rc != 0) ? (rs1 & rs2) : rs1</code>	<code>and rd, rs1, rs2</code> <code>czero.nez rtmp, rs1, rc</code> <code>or rd, rd, rtmp</code>	
Conditional select, if zero <code>rd = (rc == 0) ? rs1 : rs2</code>	<code>czero.nez rd, rs1, rc</code> <code>czero.eqz rtmp, rs2, rc</code> <code>or rd, rd, rtmp</code>	
Conditional select, if non-zero <code>rd = (rc != 0) ? rs1 : rs2</code>	<code>czero.eqz rd, rs1, rc</code> <code>czero.nez rtmp, rs2, rc</code> <code>or rd, rd, rtmp</code>	

4.2. Alternative sequences with data-invariant timing

The definition of `czero.eqz` and `czero.nez` does not generally guarantee data-invariant timing (although it guarantees independence of the value of one of its arguments, if the Zkt extension is implemented).

However, sequences using instructions covered by Zkt are available to express the same semantics as for the `czero.eqz` and `czero.nez` instructions:

Zicond instruction	Alternative sequence with data-invariant timing
<code>czero.eqz rd, rs1, rs2</code>	<code>snez rtmp, rs2</code> <code>neg rtmp, rtmp</code> <code>and rd, rtmp, rs1</code>
<code>czero.nez rd, rs1, rs2</code>	<code>seqz rtmp, rs2</code> <code>neg rtmp, rtmp</code> <code>and rd, rtmp, rs1</code>