Atomic compare-and-Swap (CAS) instructions (Zacas)

Ved Shanbhogue

Version 0.1, 4/2023: This document is in development state. See http://riscv.org/spec-state for details.

Table of Contents

Preamble . . . 1
Copyright and license information . . . 2
    Contributors . . . 2
1. Introduction . . . 3
2. Word/Doubleword/Quadword CAS (AMOCAS.W/D/Q) . . . 4
3. AMO PMA . . . 7

Preamble

This document is in the Stable state

Assume anything could still change, but limited change should be expected.

Copyright and license information

Contributors

This RISC-V specification has been contributed to directly or indirectly by: Greg Favor, Ved Shanbhogue

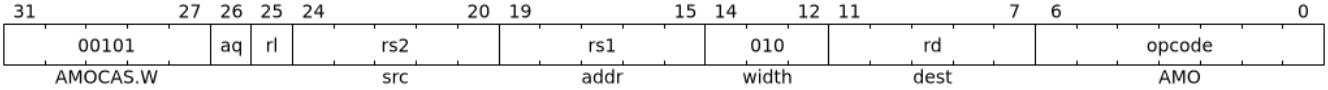
Chapter 1. Introduction

Compare-and-swap (CAS) provides an easy and typically faster way to perform thread synchronization operations when supported as a hardware instruction. CAS is typically used by lock-free and wait-free algorithms. This extension proposes CAS instructions to operate on 32-bit, 64-bit, and 128-bit (RV64 only) data values. The CAS instruction supports the C++11 atomic compare and exchange operation.

While compare-and-swap for XLEN wide data may be accomplished using LR/SC, the CAS atomic instructions scale better to highly parallel systems than LR/SC. Many lock-free algorithms, such as a lock-free queue, require manipulation of pointer variables. A simple CAS operation may not be sufficient to guard against what is commonly referred to as the ABA problem in such algorithms that manipulate pointer variables. To avoid the ABA problem, the algorithms associate a reference counter with the pointer variable and perform updates using a quadword compare and swap (of both the pointer and the counter). The double and quadword CAS instructions support implementation of algorithms for ABA problem avoidance.

Chapter 2. Word/Doubleword/Quadword CAS (AMOCAS.W/D/Q)

`AMOCAS.W` atomically loads 32-bits of a data value from address in `rs1`, compares the loaded value to a 32-bit value held in `rd` and if the comparison is bitwise equal, then stores the 32-bit value held in `rs2` to the original address in `rs1`. The value loaded from memory is placed into register `rd`. For RV64, `AMOCAS.W` always sign-extends the value placed in `rd`, and ignores the upper 32 bits of the original value in `rd` and `rs2`.

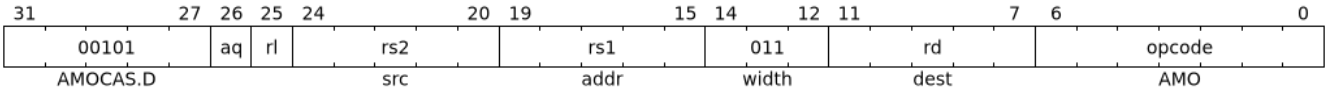


The operation performed by `AMOCAS.W` is as follows:

Listing 1. `AMOCAS.W` operation

```
temp = *[rs1]
if temp == [rd]
    *[rs1] = [rs2]
endif
[rd] = temp
```

`AMOCAS.D` is similar to `AMOCAS.W` but operates on 64-bit data values.



For RV32, `AMOCAS.D` atomically loads 64-bits of a data value from address in `rs1`, compares the loaded value to a 64-bits value held in a register pair consisting of `rd` and `rd+1` and if the comparison is bitwise equal, then stores the 64-bit value held in the register pair `rs2` and `rs2+1` to the original address in `rs1`. The value loaded from memory is placed into the register pair `rd` and `rd+1`. The instruction requires the first register in the pair to be even numbered; encodings with odd numbered registers specified in `rs2` and `rd` are reserved. When the first register of a source register pair is `x0`, then both halves of the pair read as zero. When the first register of a destination register pair is `x0`, then writes discard both halves of the pair result.

The operation performed by `AMOCAS.D` for RV32 is as follows:

Listing 2. `AMOCAS.D` for RV32 operation

```
temp  = *([rs1]+0)
temp1 = *([rs1]+4)
rs2_plus_1 = rs2 | ((rs2 == x0) ? 0 : 1)
rd_plus_1  = rd  | ((rd == x0) ? 0 : 1)
If (temp == [rd]) && (temp1 == [rd_plus_1])
    *([rs1]+0) = [rs2]
    *([rs1]+4) = [rs2_plus_1]
endif
```

```
    [rd] = temp
    [rd_plus_1] = temp1
```

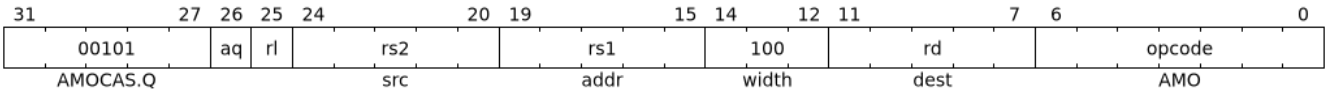
For RV64, `AMOCAS.D` atomically loads 64-bits of a data value from address in `rs1`, compares the loaded value to a 64-bit value held in `rd` and if the comparison is bitwise equal, then stores the 64-bit value held in `rs2` to the original address in `rs1`. The value loaded from memory is placed into register `rd`.

The operation performed by `AMOCAS.D` for RV64 is as follows:

Listing 3. `AMOCAS.D` for RV64 operation

```
    temp = *[rs1]
    if temp == [rd]
        *[rs1] = [rs2]
    endif
    [rd] = temp
```

`AMOCAS.Q` (RV64 only) atomically loads 128-bits of a data value from address in `rs1`, compares the loaded value to a 128-bits value held in a register pair consisting of `rd` and `rd+1` and if the comparison is bitwise equal, then stores the 128-bit value held in the register pair `rs2` and `rs2+1` to the original address in `rs1`. The value loaded from memory is placed into the register pair `rd` and `rd+1`. The instruction requires the first register in the pair to be even numbered; encodings with odd numbered registers specified in `rs2` and `rd` are reserved. When the first register of a source register pair is `x0`, then both halves of the pair read as zero. When the first register of a destination register pair is `x0`, then writes discard both halves of the pair result.



The operation performed by `AMOCAS.Q` is as follows:

Listing 4. `AMOCAS.Q` operation

```
    temp  = *([rs1]+0)
    temp1 = *([rs1]+8)
    rs2_plus_1 = rs2 | ((rs2 == x0) ? 0 : 1)
    rd_plus_1  = rd  | ((rd == x0) ? 0 : 1)
    If (temp == [rd]) && (temp1 == [rd_plus_1])
        *([rs1]+0) = [rs2]
        *([rs1]+8) = [rs2_plus_1]
    endif
    [rd] = temp
    [rd_plus_1] = temp1
```

For a future RV128 extension, `AMOCAS.Q` would encode a single XLEN=128 register in `rs2` and `rd`.

Just as for AMOs in the A extension, `AMOCAS.W/D/Q` requires that the address held in `rs1` be naturally

aligned to the size of the operand (i.e., 16-byte aligned for 128-bit words, eight-byte aligned for 64-bit words, and four-byte aligned for 32-bit words). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated. The draft “Zam” extension, described in Chapter 23, relaxes this requirement and specifies the semantics of misaligned AMOs.

Just as for AMOs in the A extension, the AMOCAS optionally provides release consistency semantics to help implement multiprocessor synchronization. If the `aq` bit is set, then no later memory operations in this RISC-V hart can be observed to take place before the AMOCAS Conversely, if the `rl` bit is set, then other RISC-V harts will not observe the AMOCAS before memory accesses preceding the AMOCAS in this RISC-V hart. Setting both the `aq` and the `rl` bit on an AMO makes the sequence sequentially consistent, meaning that it cannot be reordered with earlier or later memory operations from the same hart.

Chapter 3. AMO PMA

Within AMOs, there are seven levels of support: AMONone, AMOSwap, AMOLogical, AMOArithmetic, AMOCasW, AMOCasD, and AMOCasQ. AMONone indicates that no AMO operations are supported. AMOSwap indicates that only amoswap instructions are supported in this address range. AMOLogical indicates that swap instructions plus all the logical AMOs (amoand, amoor, amoxor) are supported. AMOArithmetic indicates that in addition to instructions supported by AMOLogical, the arithmetic AMOs (amoadd, amomin, amomax, amominu, amomaxu) are supported. AMOCasW indicates that in addition to instructions indicated by AMOArithmetic level support, the amocas.w instruction is supported. AMOCasD indicates that in addition to instructions indicated by AMOCasW level support, the amocas.d instruction is supported. AMOCasQ indicates that all RISC-V AMOs are supported. For each level of support, naturally aligned AMOs of a given width are supported if the underlying memory region supports reads and writes of that width. The draft "Zam" extension, described in Chapter 23, relaxes this requirement and specifies the semantics of misaligned AMOs. Main memory and I/O regions may only support a subset or none of the processor-supported atomic operations.

AMO Class	Supported Operations
AMONone	None
AMOSwap	amoswap
AMOLogical	above + amoand, amoor, amoxor
AMOArithmetic	above + amoadd, amomin, amomax, amominu, amomaxu
AMOCasW	above + amocas.w
AMOCasD	above + amocas.d
AMOCasQ	above + amocas.q

We recommend providing at least AMOLogical support for I/O regions where possible.

AMOCasW/D/Q PMA requires AMOArithmetic level support as the amocas instructions requires ability to perform an arithmetic comparison and a swap operation.