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

Version 1.0-rc1, 6/2023: This document is in stable state. See http://riscv.org/spec-state for details.

Table of Contents

Preamble

This document is in the Stable state

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

Contributors

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

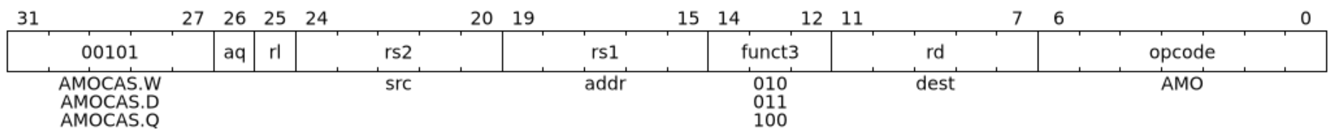
Chapter 1. Introduction

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.

The Zacas extension depends upon the A extension [1].

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



For RV32, AMOCAS.W atomically loads a 32-bit data value from address in rs1, compares the loaded value to the 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. The operation performed by AMOCAS.W for RV32 is as follows:

```
temp = mem[X(rs1)]
if ( temp == X(rd) )
    mem[X(rs1)] = X(rs2)
endif
X(rd) = temp
```

For RV64, AMOCAS.W atomically loads a 32-bit data value from address in rs1, compares the loaded value to the lower 32 bits of the value held in rd, and if the comparison is bitwise equal, then stores the lower 32 bits of the value held in rs2 to the original address in rs1. The 32-bit value loaded from memory is sign-extended and is placed into register rd. The operation performed by AMOCAS.W for RV64 is as follows:

```
temp[31:0] = mem[X(rs1)]
if ( temp[31:0] == X(rd)[31:0] )
    mem[X(rs1)] = X(rs2)[31:0]
endif
X(rd) = SignExtend(temp[31:0])
```

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-bit 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 the entire register result is discarded and neither destination register is written. The operation performed by AMOCAS.D for RV32 is as follows:

```
temp0 = mem[X(rs1)+0]
temp1 = mem[X(rs1)+4]
comp0 = (rd == x0)  ? 0 : X(rd)
```

```
    comp1 = (rd == x0)  ? 0 : X(rd+1)
    swap0 = (rs2 == x0) ? 0 : X(rs2)
    swap1 = (rs2 == x0) ? 0 : X(rs2+1)
    if ( temp0 == comp0 ) && ( temp1 == comp1 )
        mem[X(rs1)+0] = swap0
        mem[X(rs1)+4] = swap1
    endif
    if ( rd != x0 )
        X(rd)   = temp0
        X(rd+1) = temp1
    endif
```

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:

```
    temp = mem[X(rs1)]
    if ( temp == X(rd) )
        mem[X(rs1)] = X(rs2)
    endif
    X(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-bit 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 the entire register result is discarded and neither destination register is written. The operation performed by AMOCAS.Q is as follows:

```
    temp0 = mem[X(rs1)+0]
    temp1 = mem[X(rs1)+8]
    comp0 = (rd == x0)  ? 0 : X(rd)
    comp1 = (rd == x0)  ? 0 : X(rd+1)
    swap0 = (rs2 == x0) ? 0 : X(rs2)
    swap1 = (rs2 == x0) ? 0 : X(rs2+1)
    if ( temp0 == comp0 ) && ( temp1 == comp1 )
        mem[X(rs1)+0] = swap0
        mem[X(rs1)+8] = swap1
    endif
    if ( rd != x0 )
        X(rd)   = temp0
        X(rd+1) = temp1
    endif
```

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 *quadwords*, eight-byte aligned for *doublewords*, and four-byte aligned for *words*). And the same exception options apply if the address is not naturally aligned.

Just as for AMOs in the A extension, the AMOCAS.W/D/Q optionally provide release consistency semantics, using the aq and rl bits, to help implement multiprocessor synchronization.

Some algorithms may load the previous data value of a memory location into the register used as the compare data value source by a Zacas instruction. When using a Zacas instruction that uses a register pair to source the compare value, the two registers may be loaded using two individual loads. The two individual loads may read an inconsistent pair of values but that is not an issue since the AMOCAS operation itself uses an atomic load-pair from memory to obtain the data value for its comparison.

The following example code sequence illustrates the use of AMOCAS.D in a RV32 implementation to atomically increment a 64-bit counter.

```
# a0 - address of the counter.
increment:
  lw   a2, (a0)      # Load current counter value using
  lw   a3, 4(a0)     # two individual loads.
retry:
  mv   a6, a2        # Save the low 32 bits of the current value.
  mv   a7, a3        # Save the high 32 bits of the current value.
  addi a4, a2, 1     # Increment the low 32 bits.
  sltu a1, a4, a2    # Determine if there is a carry out.
  add  a5, a3, a1    # Add the carry if any to high 32 bits.
  amocas.d.aqrl a2, a4, (a0)
  bne  a2, a6, retry # If amocas.d failed then retry
  bne  a3, a7, retry # using current values loaded by amocas.d.
  ret
```

The following example code sequence illustrates the use of AMOCAS.Q to implement the *enqueue* operation for a non-blocking concurrent queue using the algorithm outlined in [2]. The algorithm atomically operates on a pointer and its associated modification counter using the AMOCAS.Q instruction to avoid the ABA problem.

```
# Enqueue operation of a non-blocking concurrent queue.
# Data structures used by the queue:
#   structure pointer_t {ptr:   node_t *, count: uint64_t}
#   structure node_t    {next: pointer_t, value: data type}
#   structure queue_t   {Head: pointer_t, Tail:  pointer_t}
# Inputs to the procedure:
```

```
#   a0 - address of Tail variable
#   a4 - address of a new node to insert at tail
enqueue:
  ld   a6, (a0)          # a6 = Tail.ptr
  ld   a7, 8(a0)         # a7 = Tail.count
  ld   a2, (a6)          # a2 = Tail.ptr.next.ptr
  ld   a3, 8(a6)         # a3 = Tail.ptr.next.count
  ld   t1, (a0)
  ld   t2, 8(a0)
  bne  a6, t1, enqueue   # Retry if Tail & next are not consistent
  bne  a7, t2, enqueue   # Retry if Tail & next are not consistent
  beq  a2, x0, move_tail # Was tail pointing to the last node?
  mv   t1, a2            # Save Tail.ptr.next.ptr
  mv   t2, a3            # Save Tail.ptr.next.count
  addi a5, a3, 1         # Link the node at the end of the list
  amocas.q.aqrl a2, a4, (a6)
  bne  a2, t1, enqueue   # Retry if CAS failed
  bne  a3, t2, enqueue   # Retry if CAS failed
  addi a5, a7, 1         # Update Tail to the inserted node
  amocas.q.aqrl a6, a4, (a0)
  ret                    # Enqueue done
move_tail:               # Tail was not pointing to the last node
  addi a3, a3, 1         # Try to swing Tail to the next node
  amocas.q.aqrl a6, a2, (a0)
  j    enqueue           # Retry
```

Chapter 3. Additional AMO PMAs

There are four levels of PMA support defined for AMOs in the A extension. Zacas defines three additional levels of support: `AMOCASW`, `AMOCASD`, and `AMOCASQ`.

`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.

`AMOCASW/D/Q` require `AMOArithmetic` level support as the `AMOCAS.W/D/Q` instructions require ability to perform an arithmetic comparison and a swap operation.

Bibliography

[1] “RISC-V Instruction Set Manual, Volume I: Unprivileged ISA .” [Online]. Available: github.com/riscv/riscv-isa-manual.

[2] M. M. Michael and M. L. Scott, “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms,” in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, New York, NY, USA, 1996, pp. 267–275, doi: 10.1145/248052.248106.