RISC-V Wait-on-Reservation-Set (Zawrs) extension

Ved Shanbhogue

Version 1.0, 6/2022: This document is in development. Assume everything can change. See http://riscv.org/spec-state for details.

Table of Contents

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

Contributors

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

Aaron Durbin, Abel Bernabeu, David Weaver, Greg Favor, Ken Dockser, Paul Donahue, Philipp Tomsich, Tariq Kurd, Ved Shanbhogue

Chapter 1. Introduction

The Zawrs extension defines a pair of instructions to be used in polling loops that allows a core to enter a low-power state and wait on either a write to a memory location or other asynchronous system events. It addresses common use-cases in operating systems when waiting for contended locks or for completion events signaled by external I/O agents, accelerators, and other peripherals. An encoding under the SYSTEM opcode is proposed.

1.1. Motivation and use cases

Waiting for a memory location to be updated is a common pattern in many usages such as:

1. Contenders for a lock waiting for the lock variable to be updated.
2. Consumers waiting on the tail of an empty queue for the producer to queue work/data. The producer may be code executing on a RISC-V hart or an accelerator device.
3. Code waiting on a flag to be set in memory indicative of an event occurring. For example, software on a RISC-V hart may wait on a “done” flag to be set in memory by an accelerator device indicating completion of a job previously submitted to the device.

Such usages involve polling on memory locations, and such busy loops can be a wasteful expenditure of energy. To mitigate the wasteful looping in such usages, a `WRS.NTO` (WRS-with-no-timeout) instruction is provided. Instead of polling for a write at a specific memory location, software would add that memory location to the reservation set using the existing `LR` instruction - a subsequent `WRS.NTO` instruction would cause the hart to stall until a write occurs to the reservation set.

Sometimes the program waiting on a memory update may also need to carry out a task at a future time (e.g., generating a heartbeat, etc.) or otherwise place an upper bound on the wait. To support such usages a second instruction `WRS.STO` (WRS-with-a-short-timeout) is provided that works like `WRS.NTO` but bounds the stall duration to a short timeout such that the stall is removed on the timeout if no other conditions to remove the stall have occurred. The program using this instruction may then determine if its next deadline has has been reached.

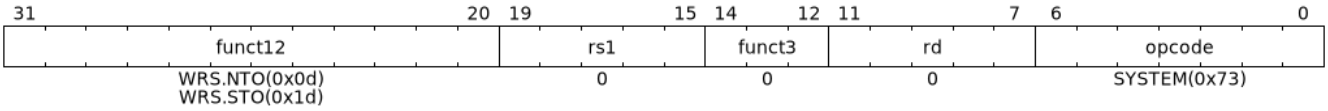
Such mechanisms have been demonstrated to help achieve increased efficiency both in terms of instruction count as well as cycle count and thereby lead to greater power efficiency compared to a busy-wait loop using yield/pause (Shamis et al., 2017), (Hu, 2019) and (Falsafi et al., 2016). In such applications, optionally bounding the wait for the event to occur with a timeout is commonly used (Scott & Scherer, 2001), (Franke et al., 2005), (Pthread, n.d.), and (Futex, n.d.).

Chapter 2. Zawrs

The WRS.NTO and WRS.STO instructions are available in all privilege modes and use the SYSTEM major opcode. When these instructions are invoked, the hart stalls until one of following events occur:

1. The reservation set is invalid
2. If WRS.STO, an implementation defined short time has passed since the hart was stalled.
3. An interrupt was observed - even if disabled

Encoding:



Operation:

```
if reservation-set is valid
   Stall hart execution until one of following events occur:
      a) reservation set is invalid
      b) if WRS.STO, a short time since start of stall has elapsed
      c) interrupt observed
```

While stalled, an implementation is permitted to remove the stall and complete execution occasionally for any reason. WRS.NTO and WRS.STO are allowed to complete in a bounded amount of time from when the condition to remove the stall occurs. These instructions are not supported in a constrained LR/SC loop.

Architecture Comment: WRS.STO and WRS.NTO are not defined as a hint but as having a defined behavior. Implementing as a hint that can be ignored (i.e., executed as the underlying nop) may lead to degradation in the system and/or application performance.

Architecture Comment: Since the WRS.STO and WRS.NTO instructions can complete execution for reasons other than writes to the reservation set, software will likely need a means of looping until the required writes have occurred.

Recommendation: An implementation should try to bound the short timeout to be long enough to allow meaningful power reduction but short enough to avoid a program using the timeout to meet a deadline from missing it significantly. Bounding the short timeout to not more than 10 microseconds is recommended.

Recommendation: An implementation should try to bound the latency to remove the stall to latency incurred on access to an on-chip cache furthest from the hart or in case of a cache-less system the access to main memory from the hart

WRS.NTO and WRS.STO instructions follows rules of the existing WFI instruction for resuming execution on a pending interrupt.

When the existing TW (Timeout Wait) bit in mstatus is set and WRS.NTO is executed in S or U mode,

and it does not complete within an implementation-specific bounded time limit, the `WRS.NTO` instruction will cause an illegal instruction exception.

When executing in VS or VU mode, if the existing `VTW` bit is set in `hstatus`, the `mstatus` `TW` bit is clear, and the `WRS.NTO` does not complete within an implementation-specific bounded time limit, the `WRS.NTO` instruction will cause a virtual instruction exception.

Bibliography

Falsafi, B., Guerraoui, R., Picorel, J., & Trigonakis, V. (2016). Unlocking Energy. *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 393–406. www.usenix.org/conference/atc16/technical-sessions/presentation/falsafi

Franke, Watson, T. J., & Kirkwood, M. (2005). *Fuss , Futexes and Furwocks : Fast Userlevel Locking in Linux Hubertus Franke IBM*. www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf

Futex. *Fast user-space locking*. linux.die.net/man/2/futex

Hu, G. (2019). *Armv8 WFE mechanism and usage in DPDK*. www.dpdk.org/wp-content/uploads/sites/35/2019/10/Armv8.pdf

Pthread. *Pthread timed wait on condition*. linux.die.net/man/3/pthread_cond_timedwait

Scott, M. L., & Scherer, W. N. (2001). Scalable Queue-Based Spin Locks with Timeout. *SIGPLAN Not.*, *36*(7), 44–52. doi.org/10.1145/568014.379566

Shamis, P., Lopez, M. G., & Shainer, G. (2017). *Enabling One-Sided Communication Semantics on ARM*. doi.org/10.1109/IPDPSW.2017.62