```
# New comparand is in a1:a0.
li t0, -1
la t1, mtimecmp
sw t0, 0(t1)     # No smaller than old value.
sw a1, 4(t1)     # No smaller than new value.
sw a0, 0(t1)     # New value.
```

Figure 3.29: Sample code for setting the 64-bit time comparand in RV32, assuming a little-endian memory system and that the registers live in a strongly ordered I/O region. Storing -1 to the low-order bits of `mtimecmp` prevents `mtimecmp` from temporarily becoming smaller than the lesser of the old and new values.

3.3 Machine-Mode Privileged Instructions

3.3.1 Environment Call and Breakpoint

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

The ECALL instruction is used to make a request to the supporting execution environment. When executed in U-mode, S-mode, or M-mode, it generates an environment-call-from-U-mode exception, environment-call-from-S-mode exception, or environment-call-from-M-mode exception, respectively, and performs no other operation.

ECALL generates a different exception for each originating privilege mode so that environment call exceptions can be selectively delegated. A typical use case for Unix-like operating systems is to delegate to S-mode the environment-call-from-U-mode exception but not the others.

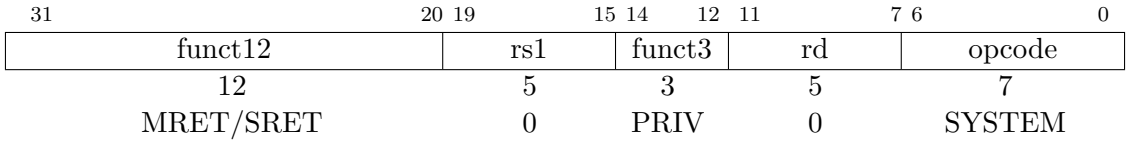
The EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment. It generates a breakpoint exception and performs no other operation.

As described in the "C" Standard Extension for Compressed Instructions in Volume I of this manual, the C.EBREAK instruction performs the same operation as the EBREAK instruction.

ECALL and EBREAK cause the receiving privilege mode's `epc` register to be set to the address of the ECALL or EBREAK instruction itself, *not* the address of the following instruction. As ECALL and EBREAK cause synchronous exceptions, they are not considered to retire, and should not increment the `minstret` CSR.

3.3.2 Trap-Return Instructions

Instructions to return from trap are encoded under the PRIV minor opcode.



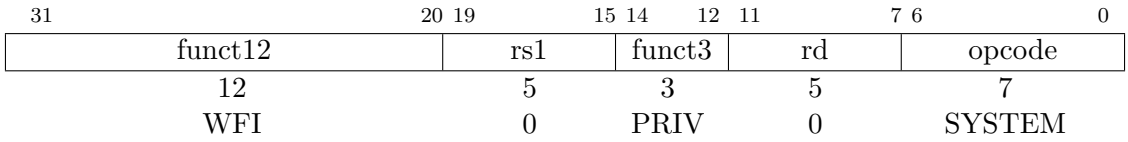
To return after handling a trap, there are separate trap return instructions per privilege level, MRET and SRET. MRET is always provided. SRET must be provided if supervisor mode is supported, and should raise an illegal instruction exception otherwise. SRET should also raise an illegal instruction exception when TSR=1 in `mstatus`, as described in Section 3.1.6.5. An *x*RET instruction can be executed in privilege mode *x* or higher, where executing a lower-privilege *x*RET instruction will pop the relevant lower-privilege interrupt enable and privilege mode stack. In addition to manipulating the privilege stack as described in Section 3.1.6.1, *x*RET sets the `pc` to the value stored in the *x*`epc` register.

If the A extension is supported, the *x*RET instruction is allowed to clear any outstanding LR address reservation but is not required to. Trap handlers should explicitly clear the reservation if required (e.g., by using a dummy SC) before executing the *x*RET.

If x*RET instructions always cleared LR reservations, it would be impossible to single-step through LR/SC sequences using a debugger.*

3.3.3 Wait for Interrupt

The Wait for Interrupt instruction (WFI) provides a hint to the implementation that the current hart can be stalled until an interrupt might need servicing. Execution of the WFI instruction can also be used to inform the hardware platform that suitable interrupts should preferentially be routed to this hart. WFI is available in all privileged modes, and optionally available to U-mode. This instruction may raise an illegal instruction exception when TW=1 in `mstatus`, as described in Section 3.1.6.5.



If an enabled interrupt is present or later becomes present while the hart is stalled, the interrupt trap will be taken on the following instruction, i.e., execution resumes in the trap handler and `mepc` = `pc` + 4.

The following instruction takes the interrupt trap so that a simple return from the trap handler will execute code after the WFI instruction.

The purpose of the WFI instruction is to provide a hint to the implementation, and so a legal implementation is to simply implement WFI as a NOP.