Tracking updated registers [in sets of registers]

Background – consideration of co-routines

Co-routines are two or more participating processes that typically relinquish control to one another. Reserving sets of physical register for each process is a technique to allow efficient hand-off to one another.
Each would be compiled [using -ffixed-*reg*] excluding the registers that are reserved for the other process(es). [That is, they have a fixed purpose; that of being reserved for the other process(es)].

Thus on relinquishing and establishing control there are fewer registers to save and restore.
Of course there is the potential downside that performance in each routine is less because they have fewer registers to work with, but RV32 is register rich and many routines can still be efficient without all available. This technique turns the large register set to an asset rather than a (overhead) liability.

Each process has to be well behaved and thoroughly tested (or proved) correct.
Performance counters, that are specific to registers, could be used to validate that each process has not used the reserved registers. However, performance counters are not performant for application production use. A special csr that reset a bit corresponding the register that changed would be presumably be more performant to setup and check.

A plausible register write tracking csr:
Bits 30 through 0 correspond to registers x31 to x1.
If the corresponding x register is written, the bit is set to zero (or left at zero).

For RV32 bit 31 (in RV64 63) is set to zero if any of bits 30 through 0 transition from 1 to zero.
Thus the sign bit can be checked to determine if any of the “reserved” registers was written.

I will leave it as an exercise to the reader to consider an interface between cooperative processes that sets and checks this csr for transitions between co-processes.

So, what has this to do with interrupts?

Interrupts are effectively a [set of] co-processes the co-operated with the “normal” running processes.
They are co-processes because in most cases the “normal” running processes are recipients of information obtained from the interrupt; such as a process queued to handle data from a ready device.

However, these are not generally co-operating co-processes, and generally have not voluntarily relinquished control (although there are designs that do).

So, assuming the general case, what value does such a register-write-tracking process provide for fast-interrupt. Some. Mainly opportunistic and worse case scenario benefits.

In the worst case an interrupt handler enters the return-from-interrupt critical section (which could just be the mret [or sret]) just as an other interrupt that could be handled by this handler occurs. Without any useful progress in the interrupted routine [that has thus not written to any registers], the set of application registers will be resaved, overwritten with the save values as were just restored. An efficient check of used registers that bypasses the save saves those memory instructions, cycles and references. Which is a boon for fast-interrupt processing.

There is a marketing benefit.
RISCV can be configured to be very performant without this feature. My various iterations of the sample interrupt handling routines show we can reduce the critical-routine section down to a few instructions. (see issues 102 and 106). However, competitors claim they are best, because, if there is a pending interrupt their CICS-like-return-from-interrupt instruction will abort and restart the handler without resaving the registers. With this feature we definitely match them.

Indeed, with this feature we will surpass the competitors in opportunistic situations.
Specifically, where the registers that the handler needs have not been changed by the returned-to program since the last interrupt return.

When the return-to-program is co-operative [which can be readily managed in an embedded environment] this feature can provide up to 100% elimination of register saves.

Details on how to tailor the register-write-tracking for interrupt handling are in issue #106.