FUJITSU FR-V KERNEL ATOMIC OPERATIONS

On the FR-V CPUs, there is only one atomic Read-Modify-Write operation: the SWAP/SWAPI
instruction. Unfortunately, this alone can't be used to implement the following operations:

 (*) Atomic add to memory

 (*) Atomic subtract from memory

 (*) Atomic bit modification (set, clear or invert)

 (*) Atomic compare and exchange

On such CPUs, the standard way of emulating such operations in uniprocessor mode is to disable
interrupts, but on the FR-V CPUs, modifying the PSR takes a lot of clock cycles, and it has to be
done twice. This means the CPU runs for a relatively long time with interrupts disabled,
potentially having a great effect on interrupt latency.

NEW ALGORITHM

To get around this, the following algorithm has been implemented. It operates in a way similar to
the LL/SC instruction pairs supported on a number of platforms.

 (*) The CCCR.CC3 register is reserved within the kernel to act as an atomic modify abort flag.

 (*) In the exception prologues run on kernel->kernel entry, CCCR.CC3 is set to 0 (Undefined
     state).

 (*) All atomic operations can then be broken down into the following algorithm:

     (1) Set ICC3.Z to true and set CC3 to True (ORCC/CKEQ/ORCR).

     (2) Load the value currently in the memory to be modified into a register.

     (3) Make changes to the value.

     (4) If CC3 is still True, simultaneously and atomically (by VLIW packing):

         (a) Store the modified value back to memory.

         (b) Set ICC3.Z to false (CORCC on GR29 is sufficient for this - GR29 holds the current
             task pointer in the kernel, and so is guaranteed to be non-zero).

(5) If ICC3.Z is still true, go back to step (1).

This works in a non-SMP environment because any interrupt or other exception that happens between
steps (1) and (4) will set CC3 to the Undefined, thus aborting the store in (4a), and causing the
condition in ICC3 to remain with the Z flag set, thus causing step (5) to loop back to step (1).

This algorithm suffers from two problems:

(1) The condition CCCR.CC3 is cleared unconditionally by an exception, irrespective of whether or
not any changes were made to the target memory location during that exception.

(2) The branch from step (5) back to step (1) may have to happen more than once until the store
manages to take place. In theory, this loop could cycle forever because there are too many
interrupts coming in, but it's unlikely.

=======
EXAMPLE
=======

Taking an example from include/asm-frv/atomic.h:

```
        static inline int atomic_add_return(int i, atomic_t *v)
        {
                unsigned long val;

                asm("0:                                                 \n"
```

It starts by setting ICC3.Z to true for later use, and also transforming that into CC3 being in the
True state.

```
                    "   orcc            gr0,gr0,gr0,icc3        \n"     <-- (1)
                    "   ckeq            icc3,cc7                \n"     <-- (1)
```

Then it does the load. Note that the final phase of step (1) is done at the same time as the
load. The VLIW packing ensures they are done simultaneously. The ".p" on the load must not be
removed without swapping the order of these two instructions.

```
                    "   ld.p            %M0,%1                  \n"     <-- (2)
                    "   orcr            cc7,cc7,cc3             \n"     <-- (1)
```

Then the proposed modification is generated. Note that the old value can be retained if required
(such as in test_and_set_bit()).

```
                        "   add%I2          %1,%2,%1                \n"     <-- (3)
```

Then it attempts to store the value back, contingent on no exception having cleared CC3 since it
was set to True.

```
                        "   cst.p           %1,%M0          ,cc3,#1 \n"     <-- (4a)
```

It simultaneously records the success or failure of the store in ICC3.Z.

```
                        "   corcc           gr29,gr29,gr0   ,cc3,#1 \n"     <-- (4b)
```

Such that the branch can then be taken if the operation was aborted.

```
                        "   beq             icc3,#0,0b              \n"     <-- (5)
                        : "+U"(v->counter), "=&r"(val)
                        : "NPr"(i)
                        : "memory", "cc7", "cc3", "icc3"
                        );

                return val;
        }
```

==========
CONFIGURATION
==========

The atomic ops implementation can be made inline or out-of-line by changing the CONFIG_FRV_OUTOFLINE_ATOMIC_OPS configuration variable. Making it out-of-line has a number of
advantages:

 - The resulting kernel image may be smaller
 - Debugging is easier as atomic ops can just be stepped over and they can be breakpointed

Keeping it inline also has a number of advantages:

 - The resulting kernel may be Faster
   - no out-of-line function calls need to be made
   - the compiler doesn't have half its registers clobbered by making a call

The out-of-line implementations live in arch/frv/lib/atomic-ops.S.