MUTEX (MUTual EXclusion) Mechanism in Hardware

Klaus Schleisiek - kschleisiek at freenet.de

Thanks to µCore's PAUSE signal, mutual exclusion processing can be completely realised in hardware. This gets rid of a very popular source of hard to track errors in complex control systems.

What is mutual exclusion and why do we have to bother?

In complex control systems, there are usually several control loops, each of which is running in its own task. For input signal acquisition, the same A/D converter (ADC) may be shared by several tasks. A/D conversion usually takes several processor cycles and therefore, a MUTEX mechnisam is needed for two things:

a) To prevent any task to touch the ADC while it is converting another task's input signal.

b) To guarantee that a conversion result is actually read by the task that started the conversion previously.

For the sake of discussion I assume that we have an ADC with an integrated eight channel multiplexer. It is connected to a hardware/software interface that allows to store a channel number into its memory mapped ADC register (**ADC_reg**) initiating signal acquisition. When the A/D conversion has finished, **ADC_reg** can be read returning the conversion result of the selected channel. In order to detect when the conversion has finished, we also have flag bit **#adc_ready** in the FLAG register (**FLAG_reg**). In addition we can define **Semaphor**s with methods **lock** and **unlock**.

In the first part of the paper I will show how mutual exclusion is usually implemented in software.

In the second part I will show how the entire MUTEX mechanism can be realised in hardware using µCore's PAUSE signal.

Conventional MUTEX in software

At first some informal word definitions of a typical co-operative multi tasking system:

pause (--) puts the current task to sleep and calls the scheduler in order to give another task the chance to run. Please note that µCore's PAUSE signal input and the Forth word **pause** are completely different things!

Semaphor is a defining word that creates a semaphor. In priciple it is a variable that allows to store true and false.

lock (addr --) as long as the semaphor at addr is true, **pause** is executed. When it is false, it is set to true and execution of the task continues.

unlock (addr --) sets the semaphor at addr to false.

```
Semaphor sema_ADC
: sample  ( channel -- sample )
   sema_ADC lock   ADC_reg !
   BEGIN  pause   FLAG_reg @ #adc_ready and UNTIL
   ADC_reg @   sema_ADC unlock
;
Variable Result
```

And now a task may acquire the analog data of channel 4 in a safe way using the following phrase:

```
... 4 sample Result ! ...
```

MUTEX (MUTual EXclusion) Mechanism in Hardware

MUTEX in hardware

As the Janus-faced side of interrupts, µCore has an additional hardware input signal **pause**, which, when raised, aborts the current instruction, pushes the instruction's program memory address on the return stack, and branches to the pause trap.

interrupt:	An event did happen that was **not** expected by the software.
pause:	An event did **not** happen that was expected by the software.

In a single task environment, the pause trap just holds an `exit` instruction. As a result, the processor would continuously try to execute the instruction that raised the pause signal until some external event makes the instruction executable.

In a multi tasking environment, the pause trap holds a branch to Forth's `pause` routine defined above and therefore, the processor can do other things while waiting for the external event to happen.

As before, we need the flag bit **#adc_ready**. But this time, it is not only a flag that can be read by the processor, it becomes the semaphor itself.

Here is some hardware pseudo code (simplified VHDL) that is needed for the MUTEX mechanism:

`read_ADC_reg` is true when the Forth phrase **`ADC_reg @`** is executed.

`write_ADC_reg` is true when the Forth phrase **`ADC_reg !`** is executed.

`ADC_busy` is true while the ADC is converting. Very often this is an output signal of the ADC chip.

The hardware implementation consists of a combinatorial part that feeds the pause input of µCore

```
pause <= true
         WHEN (read_ADC_reg = true AND ADC_busy = true) OR
              (write_ADC_reg = true AND FLAG_reg(#adc_ready) = false)
         ELSE false;
```

and a sequential part that controls the **#adc_ready** bit storing its result on the rising edge of a clock.

```
IF  rising_edge(clk)  THEN
   IF  write_ADC_reg = true  THEN
      FLAG_reg(#adc_ready) <= false;
   END IF;
   IF  read_ADC_reg = true AND pause = false  THEN
      FLAG_reg(#adc_ready) <= true;
   END IF;
   IF  reset = true  THEN
      FLAG_reg(#adc_ready) <= true;
   END IF;
END IF;
```

And now a task may acquire the analog data of channel 4 in a safe way using the following phrase:

```
... 4 ADC_reg !   ADC_reg @ Result ! ...
```

In essence, you can treat the **ADC_reg** similar to a variable without bothering about conversion time needed or mutual exclusion on the software level any more.