Component description

Objective

This component aims at encapsulating some code written in C into a hardware component, called *virtual coprocessor wrapper*. The objective targetted is to have an estimation of the performances when a task is executed in hardware, without having to write a real coprocessor.

Modelisation

The coprocessor is modeled by a hardware component, the *wrapper*, which contains registers, a transition function and a Moore generation function. It can be interfaced with a `vci_mwmr_controller` that allows access to any number of MWMR channels in memory. The C function corresponding to the hardware task is associated to a posix thread, which is executed in parallel with the simulator for the architecture. This task thread is launched by the *wrapper* on reset.

The task thread communicates with the *wrapper* with two fifo channels:

- the `cmd` channel, which transmits to the *wrapper* the SRL commands. These commands take the form of function calls in the task code:
  - `srl_mwmr_read(channel, buffer, size)`
  - `srl_mwmr_write(channel, buffer, size)`
  - `srl_busy_cycles(n_cycles)`
- the `rsp` channel, which is used to transmit the response to the read requests

To avoid race conditions, a single lock is used for both fifos cmd and rsp.

The *wrapper* is implemented as a four-state automaton, whose role is to execute the SRL commands transmitted by the hardware task.

1. In IDLE state, the automaton tests if a command is available in the `cmd` fifo.
   - If yes, it gets the parameters and goes in a state corresponding to the type of the command to execute it.
   - If no, it waits until a command is available

2. In READ state, the automaton transfers data from the MWMR controller directly into the buffer of the task thread (1 word/cycle). When the last word has been transfered, the automatod signals the completion to wake up the task thread, and returns in IDLE state.

3. In WRITE state, the automaton makes the transfer from the task thread to the MWMR controller (1 word/cycle), and returns in idle when the transfer is complete.

4. In BUSY state, the automaton decreases a counter at each cycle until the counter reaches 0.

Synchronization

The *wrapper* behaves as if all the SRL commands were blocking for the task thread, which is well adapted to sequential coprocessors, which never execute several commands in parallel:

However, to avoid potential useless switches between the threads, only the read commands really are blocking. Thus, if a task thread only makes writes, it will continue executing until the `cmd` fifo is considered full (the defaut value is 50 commands, but it could be anything).

The two threads are synchronized with two `pthread_cond_t` conditions: `task_cond` and `wrapper_cond`. Here is the simplified execution scheme on one (host) processor, for a task thread making only reads:

- On creation, the task thread is executed
- The task thread makes a read: it sends a command and then try to read a response; as there is none, it waits on `task_cond`
- The wrapper thread is executed, receives the read request, and process it. When the response is ready, it signals `task_cond`
- The wrapper thread tries to read another command. As there is none, it waits on `wrapper_cond`
- The task thread is executed, and makes another read, etc.

Here is the simplified execution scheme on one (host) processor, for a task thread making only writes:

- On creation, the task thread is executed
- The task thread makes a write: it sends a command and then continue its execution
- After a certain number of writes, the command fifo reaches its maximum value. The task thread signals `wrapper_cond` and waits on `task_cond`
- The wrapper thread is executed, and start processing all the writes commands.
- When there is no more commands in the `cmd` fifo, the wrapper thread signals `task_cond` and waits on `wrapper_cond`
- The task thread is executed, and makes another write, etc.

With several cores, the difference is that after the creation of the task thread, the wrapper thread can continue its execution, and theoritically consume the writes as they are produced, thus none of the threads never waits (though in practice the task thread will always be faster).

Component usage

The virtual Coprocessor Wrapper is primarily intended to be used with Dsx-vm (to be released soon?), but it can also be used "by hand". In that case, the component cannot be used directly; instead, it is necessary to create a new component which derives from the *wrapper*.

Example

In the following is an example of a dummy *adder* task, which makes a vectorial 32-bit addition of size 8 (i.e. there are 16 words in input and 8 words in output). 3 files must be created:

- `my_adder_copro.sd`
- `my_adder_copro.h`
- `my_adder_copro.cpp`

The file `my_adder_copro.sd` must contain the following:

```
#-*- python -*-

Module('caba:my_adder_copro',
       classname = 'dsx::caba::MyAdderCopro',
       header_files = [
               "my_adder_copro.h",

       ],
       interface_files = [
       ],
       implementation_files = [
               "my_adder_copro.cpp",

       ],
       ports = [
       ],
       uses = [
           Uses('caba:fifo_virtual_copro_wrapper'),
       ],
       instance_parameters = [
       ],
       tmpl_parameters = [
       ],
       extensions = [
       ],
)
```

The file `my_adder_copro.h` must be as following:

```
#ifndef _ADDER_COPRO_H
#define _ADDER_COPRO_H

#include <systemc>

#include "fifo_virtual_copro_wrapper.h"

namespace dsx { namespace caba {

class MyAdderCopro
    : public dsx::caba::FifoVirtualCoprocessorWrapper
{

    public:
    ~MyAdderCopro();
    MyAdderCopro(sc_core::sc_module_name insname);


    private:
    void * task_func(); // Task code

};

}}
#endif /* _ADDER_COPRO_H */
```

Finally, the file `my_adder_copro.cpp` must contain the task code. We declare here two "input fifos" (`input0` and `input1`) and one "output fifo" (`output`) in the constructor.

```
#include "my_adder_copro.h"

namespace dsx { namespace caba {
```

```
#define tmpl(...) __VA_ARGS__ MyAdderCopro

tmpl(/**/)::~MyAdderCopro()
{
}

tmpl(/**/)::MyAdderCopro(sc_core::sc_module_name insname)
           :dsx::caba::FifoVirtualCoprocessorWrapper(insname, stringArray("output", NULL), intAr
{
}

tmpl(void *)::task_func() {
   srl_mwmr_t input = SRL_GET_MWMR(input);
   srl_mwmr_t output = SRL_GET_MWMR(output);

   uint32_t in0[8];
   uint32_t in1[8];
   uint32_t out[8];

   while (true) {
      srl_mwmr_read(input0, &in0, 1); // Read 8 words from input0, i.e. 1 item since the fifo is
      srl_mwmr_read(input1, &in1, 1); // Read 8 words from input1
      for (int32_t i = 0; i < 8; i++) {
         out[i] = in0[i] + in1[i];
      }
      srl_busy_cycles(2); // The computation takes 2 cycles
      srl_mwmr_write(output, &out, 1); // Write 8 words to output
   }
}

}}
```

The prototype of the parent class `FifoVirtualCoprocessorWrapper` is:

```
FifoVirtualCoprocessorWrapper(
        sc_core::sc_module_name insname,
        const vector<string>  &fifos_out,
        const vector<int32_t> &fifos_out_width,
        const vector<string>  &fifos_in,
        const vector<int32_t> &fifos_in_width)
: soclib::caba::BaseModule(insname)
```

with

fifos_out
: a vector containing the name of the output fifos

fifos_out_width
: a vector containing the width of the output fifos

fifos_in
: a vector containing the name of the input fifos

fifos_in_width
: a vector containing the width of the input fifos

The functions `stringArray` and `intArray` construct these vectors, the first argument of intArray being the number of fifos.

Component Instanciation

CABA Ports

- `sc_in<bool>` **p_resetn** : Global system reset
- `sc_in<bool>` **p_clk** : Global system clock
- `FifoOutput<uint32_t> *` **p_to_ctrl** : list of output ports to connect to a `soclib::caba::VciMwmrControllerCas` via a `soclib::caba::FifoSignals<uint32_t>`.
- `FifoInput<uint32_t> *` **p_from_ctrl** : list of input ports to connect to a `soclib::caba::VciMwmrControllerCas` via a `soclib::caba::FifoSignals<uint32_t>`.