A general method for SystemC modeling of RISC processors

Authors : Alain Greiner, François Pécheux, Nicolas Pouillon

The goal of the method presented here is to simplify the SystemC modeling of a specific class of embedded processors : The method is well suited to 32 bits RISC processors, with one single instruction issue per cycle, and blocking instruction and data caches.

A) General principles

The method relies on three basic principles :

- The processor core is modeled as a generic ISS (Instruction Set Simulator).
- This ISS is wrapped in apropriate wrappers for several types of simulation models : CABA, TLM-T and PV.
- All processors types use the same generic cache controler.

On one hand, the same ISS is encapsulated in different wrappers to generate several simulation models, corresponding to several abstraction levels: CABA (Cycle-Accurate Bit-Accurate), TLM-T (Transaction Level Models with Time), and PV (Programmer View, untimed). On the other hand, it is possible to use the same wrapper for different types of processor architectures. As illustrated below, all simulation models can be obtained as the cartesian product of the ISS set, by the wrappers set.

	CABA Wrapper	TLM-T Wrapper	PV Wrapper
ISS MIPSR3000	CABA Model MIPS	TLM-T Model MIPS	PV Model MIPS
ISS PPC405	CABA Model PPC	TLM-T Model PPC	PV Model PPC
ISS OpenRISC	CABA Model OpenRISC	TLM-T Model OpenRISC	PV Model OpenRISC

The method has been demonstrated for the MIPSR3000 and PPC 405 processors, and can be simply extended to the OpenRISC, Sparc, Nios, and MicroBLAZE processors.

This modeling approach supposes that all ISS implement the same generic API (Application Programming Interface), as this API must be independant from both the procesor architecture, and the wrapper type.

The proposed method makes the assumption that the processors use the **VcIXcache** cache controler available in the SoCLib library to interface the VCI interconnect. Such modular approach allows to share the modeling effort of the L1 cache controler. The functionnal validation and debug of this component has been a tedious task, and such reuse is probably a good policy. Nevertheless, a clean procedural interface has been defined between the processor core, and the cache controler, and the cache behaviour can be easily modified if required.

Finally this generic approach has been exploited to develop the GdbServer module that is mandatory to help the debugging of multi-task applications running on the MP-SoC architectures modeled with SoCLib. This tool can be used for all simulation models compliant with the method described below.

B) Generic ISS API

As explained in the introduction, the modeling method relies on a generic ISS API, usable by any 32-bit RISC processor, and by the three wrappers CABA, TLM-T & PV. The Instruction Set Simulator corresponding to a given processor handles a set of registers definning the processor internal state. The API described below defines a procedural interface to allow the various wrappers to access those registers.

Function **step()** is the main entry point, it executes one ISS step :

- For an untimed model (PV wrapper) one step corresponds to one instruction.
- For a timed model (CABA wrapper or TLM-T wrapper), one step corresponds to one cycle.

API:

- **inline void reset()**

This function resets all registers defining the processor internal state.

- **inline bool isBusy()**

This function is only used by timed wrappers (CABA & TLM-T). In RISC processors, most instructions have a visible latency of one cycle. But some instructions (such as multiplication or division) can have a visible latency longer than one cycle. This function is called by the CABA and TLM-T wrappers before executing one step : If the processor is busy, the wrapper calls the **nullStep()** function. If the processor is available, the wrapper may call the **step()** function to execute one instruction.

- **inline void step()**

This function executes one instruction. All processor internal registers can be modified.

- **inline void nullStep()**

This function performs one internal step of a long instruction.

- **inline void getInstructionRequest (bool & req , uint32_t & address)**

This function is used by the wrappers to obtain from the ISS the instruction request parameters. The **req** parameter is true when there is a valid request. The **address** parameter is the instruction address.

- **inline void getDataRequest (bool &req , enum DataAccessType & type, uint32_t & address, uint32_t & wdata)**

This function is used by the wrapper to obtain from the ISS the data request parameters. The **req** parameter is true when there is a valid request. The **address** parameter is the data address, and the **wdata** parameter is the data value to be written. The **type** parameter is defined below :

```
enum DataAccessType {
    READ_WORD,   // Read Word
    READ_HALF,   // Read Half
    READ_BYTE,   // Read Byte
    LINE_INVAL,  // Cache Line Invalidate
    WRITE_WORD,  // Write Word
    WRITE_HALF,  // Write Half
    WRITE_BYTE,  // Write Byte
    STORE_COND,  // Store Conditional Word
```

```
    READ_LINKED, // Load Linked Word
}
```

- **inline void setInstruction (bool error, uint32_t ins)**

This function is used by the wrapper to transmit to the ISS, the instruction to be executed (**ins** parameter). In case of exception (bus error), the **error** parameter is set.

- **inline void setDataResponse (bool error, uint32_t rdata)**

This function is used by the wrapper to transmit to the ISS, the response to the data request. In case of a read request, the **rdata** parameter contains the read value. In case of exception (bus error), the **error** parameter is set.

In any case, this function must reset the ISS data request.

- **inline void setWriteBerr ()**

This function is used by the wrapper to signal asynchronous bus errors, in case of a write acces, that is non blocking for the processor.

- **inline void setIrq (uint32_t irq)**

This function is used by the wrapper to signal the current value of the interrupt lines. For each processor, the number of interrupt lines must be defined by the ISS static variable **n_irq**.

C) ISS internal organisation

As an example, we present the general structure of the MIPS-R3000 ISS (chronogram of figure 1). The instruction fetch, instruction decode, and instruction execution are done in one cycle.

A specific register **r_npc** is introduced to model the delayed branch mechanism : the instruction following a branch instruction is always executed.

The load instructions are executed in two cycles, as those instructions require two cache access (one for the instruction, one for the data). The ISS can issue two simultaneous request for the instruction cache, and the data cache, but those requests are done for different instructions.

The **r_pc** and **r_npc** registers contain respectively the current instruction address, and the next instruction address. The wrapper can obtain the PC content using the **getInstructionRequest()** function, fetch the instruction in the cache (or in memory in case of MISS), and propagate the requested intruction to the ISS using the **setInstruction()** function.

The wrapper starts the instruction execution using the **step()** function. The general registers **r_gp**, as well as the **r_mem** registers defining the possible data access, are modified.

At the end of cycle (i), if the **r-mem** registers contain a valid data access, this access will be performed during the next cycle, in parallel with the execution of instruction executed at cycle (i+1).

From an implementation point of view, a specific ISS is implemented by a class **processorIss**. This class inherits the class **soclib::common::Iss**, that defines the prototypes of the access function presented in section B (defined as pure virtual methods).

D) Generic cache controler

The hardware component **VciXcache** is a generic cache controler that can be used by various processor cores.

It contains separated instruction and data caches, but has a single VCI port to acces the VCI interconnect.

The cache line width, and the cache size are defined as independant parameters for the data cache and the instruction cache.

On the processor side, the cache controler can receive two requests at each cycle : one instruction request (read only), and one data request (read or write). Those requests, and the corresponding responses are transmited through a normalised interface described below.

Both instruction and data caches are blocking : the processor is supposed to be frozen in case of MISS (uncached read acces are handled as MISS).

Both caches are direct mapped, and the write policy for the data cache is WRITE-THROUGH. The cache controler contains a write buffer supporting up to 8 fposted write requests. In case of successive write requests to contiguous addresses, the cache controler will build a single VCI burst. Therefore, the procesor can be blocked in case of MISS on a read request, but is generally not blocked in case of write request.

Finally, in order to garanty a strong ordering memory consistency, the ???VciXcache??? controler sequencialize the memory accesses, strictly respecting the access ordering defined by the processor on the **VciXcache** interface. As the VCI interconnect does not garanty the in order delivery property, the cache controler waits the VCI response packet corresponding to transaction (n) before sending the VCI command packet corresponding to transaction (n+1).

To communicate with the processor, the CABA model of the **VciXcache** component contains two ports defined below :

```
class IcacheCachePort {
    sc_in<bool>  req;  // valid request
    sc_in<sc_dt::sc_uint<32> >  adr;  // instruction address
    sc_out<bool>  frz ;  // frozen processor
    sc_out<sc_dt::sc_uint<32> >  ins;  // instruction
    sc_out<bool>  berr;  //  bus error
}

class DcacheCachePort {
    sc_in<bool>  req;  // valid request
    sc_in<sc_dt::sc_uint<4>  >  type ;  // data access type
    sc_in<sc_dt::sc_uint<32> >  adr;  // data address
    sc_in<sc_dt::sc_uint<32> >  wdata;  // data to be written
    sc_out<bool>  frz ;  // frozen processor
    sc_out<sc_dt::sc_uint<32> >  rdata;  // read data
    sc_out<bool>  berr;  // bus error
}
```

E) CABA modeling

The CABA modeling for a complete CPU (processor + cache) is presented in figure 2.

The processor ISS is wrapped in the generic CABA wrapper, implemented by the class **IssWrapper**.

The class **IssWrapper** contains the member variable **m_iss** representing the processor ISS. The type of the **m_iss** variable - defining the type of the wrapped processor - is specified by the template parameter **iss_t**. The class **IssWrapper** inherits the class **caba::BaseModule**, that is the basis for all CABA modules.

To communicate with the **VciXcache**, the **IssWrapper** class contains two member variables **p_icache**, of type **IcacheProcessorPort** and **p_dcache**, of type **DcacheProcessorPort**. It also contains the member variable **p_irq**, that is a pointer to an array of ports of type **sc_in<bool>**. This array represents the interrupt ports. The number N of interrupt ports depends on the wrapped processor, an is defined by the **n_irq** static member variable of the **iss_t** class.

The SystemC code for the generic CABA wrapper is presented below :

```
template<typename iss_t>
class IssWrapper : Caba::BaseModule
{

public :

//////// ports ////////
sc_in<bool>  *p_irq ;
IcacheProcessorPort  p_icache ;
DcacheProcessorPort  p_dcache ;
sc_in<bool>  p_resetn ;
sc_in<bool>  p_clk ;

///////// constructor ///////////
IssWrapper(sc_module_name insname,
           int  ident ) :
    BaseModule(insname),
    m_iss(ident)
    {
    p_icache("icache") ;
    p_dcache("dcache") ;
    p_resetn("resetn") ;
    p_clk("clk") ;
    for (uint32_t i = 0 ; i < iss_t::n_irq ; i++) {
        new(&p_irq[i]) sc_in<bool> ("irq", i) ;
        }
    m_ins_asked = false ;
    m_data_asked = false ;
    SC_METHOD(transition);
    dont_initialize();
    sensitive << p_clk.pos();
    SC_METHOD(genMoore);
    dont_initialize();
    sensitive << p_clk.neg();
    }

private :

///////////
iss_t  m_iss ;
bool m_iss_asked;
bool m_data_asked;

////////////////////////
void transition()
    {
    bool ifrz = p_icache.frz.read() ;
    bool iberr = p_icache.berr.read() ;
```

```
    bool dfrz = p_dcache.frz.read() ;
    bool dberr = p_dcache.berr.read() ;

    if ( ! p_resetn.read() ) {
        m_iss.reset();
        return;
    }

    if ( m_iss_asked ) m_iss.setInstruction( iberr, p_icache.ins.read() ) ;

    if ( dberr && ( !m_data_asked || dfrz ) ) {
        m_iss.setWriteBerr() ;
    } else if ( m_data_asked ) {
        m_iss.setDataResponse( dberr, p_dcache.rdata.read() ) ;
    }
    if ( m_iss.isBusy() || ifrz || dfrz ) {     //  Processor frozen or busy
        m_iss.nullStep();
    } else {                        // Execute one instruction:
        m_iss.step();
    }
    // report interrupts
    uint32_t irqword = 0;
    for ( size_t i=0; i<(size_t)iss_t::n_irq; i++ )  { if (p_irq[i].read()) irqword |= (1<<i); }
    m_iss.setIrq(irqword);

    } // end transition()

//////////////////////////////
void genMoore()
    {
    bool ins_req ;
    uint32_t ins_address ;
    bool data_req ;
    enum DataAccessType data_type ;
    uint32_t data_address ;
    uint32_t  data_wdata ;

    m_iss.getDataRequest( data_req, data_type, data_address, data_wdata ) ;
    m_iss.getInstructionRequest( ins_req, ins_address ) ;

    m_ins_asked = ins_req ;
    m_data_asked = data_req ;

    p_icache.req = ins_req ;
    p_icache.adr = ins_address;
    p_dcache_req = data_req ;
    p_dcache_type = data_type ;
    p_dcache.adr = data_address;
    p_dcache.wdata = data_wdata;
} // end genMoore
```

F) TLM-T modeling

The TLM-T modeling for a complete CPU (processor + cache) is presented in figure 3.

To increase the simulation speed, the TLM-T wrapper is the cache controller itself, and it is implemented as the class **VciXcache**. This class contains the SC_THREAD **execLoop()** implementing the PDES process, and the **c0** member variable of type **tlmt_core::tlmt_thread_context** that mainly contains the associated local clock.

The class **VciXcache** inherits from the class **tlmt::ModuleBase**, that is the root class for all TLM-T modules.

This class contains the member variable **m_iss** representing the processor ISS. The type of the **m_iss** variable is defined by the template parameter **iss_t**.

The class **VciXcache** contains a member variable **p_vci**, of type **VciInitiator**, to send VCI command packets, and receive VCI response packets.

This class also contains the member variable **p_irq**, which is a pointer to an array of ports of type **tlmt_core::tlmt_in<bool>**. This array represents the interrupt ports. The number N of interrupt ports depends on the wrapped processor, and is defined by the **n_irq** member variable of the **iss_t** class.

The **execLoop()** function contains an infinite loop. One iteration in this loop corresponds to one cycle for the local clock (or more, if the thread is suspended in case of a cache miss).

The cache behavior is specifically described by the **xcacheAccess()** method, which is a member function of the class **VciXcache**. This function is called by **execLoop()** at each cycle. This function has the following prototype :

```
void xcacheAccess(
        bool &ins_asked,                        // The iss requests a new instruction
        uint32_t &ins_addr,                     // at the address ins_addr

        bool &mem_asked,                        // In parallel, the iss can ask for a memory acc
        enum iss_t::DataAccessType &mem_type    // of type mem_type,
        uint32_t &mem_addr,                     // and at address mem_addr
        uint32_t &mem_wdata,                    // Eventually, the iss also provides the write d

        uint32_t &mem_rdata,                    // In return, the xcacheAccess function returns
        bool &mem_dber,                         // and a potential data bus error
        uint32_t &ins_rdata,                    // as well as the next instruction to be execute
        bool &ins_iber                          // and a potential instruction bu error
        );
```

The **xcacheAccess()** function determines the actions to be done by the cache:

- In case of data or instruction MISS, the **xcacheAccess()** function sends the appropriate VCI command packet on the **p_vci** port, and the **exedcLoop()** thread is suspended.
- In case of data write, the the **xcacheAccess()** function sends the appropriate VCI command packet on the **p_vci** port, but the **exedcLoop()** thread is not suspended. Thus, the processor is not stalled

At each iteration in the execution loop, the **xcacheAccess()** method updates the local clock (through the variable **c0**) :

- The local time is simply incremented by one cycle, if the cache controller is able to answer immediately to the processor requests.
- The local time is updated using the date contained in the VCI response packet in case of MISS.

The SystemC TLM-T model for the **VciXcache** module is presented below in the following files

!Add the links to the actual source code here !!'''