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 Specific 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 debug of the multi-tasks application software 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 bits 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 allows the various wrappers to access those registers. The main access function is the **step()** function, that 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.

- **inline void reset()**

This function reset 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 larger 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 , enum InsAccessType & type, uint32_t & address)**

This function is used by the wrapper 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. The **type** parameter can have the values defined below:

```
enum InsAccessType {
    RC ,  // Read Instruction Cached
    RU ,  // Read Instruction Uncached
}
```

- **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 {
    RW ,   // Read Word Cached
    RH ,  // Read Half Cached
    RB  ,  // Read Byte Cached
    RZ ,   // Cache Line Invalidate
    RWU ,  // Read Word Uncached
    RHU ,  // Read Half Uncached
    RBU ,  // Read Byte Uncached
```

```
        WW ,  // Write Word
        WH ,  // Write Half
        WB ,  // Write Byte
        SC ,  // Store Conditional Word
        LL ,  // 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 setRdata (bool error, uint32_t rdata)**

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

- **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 variable **n_irq**.

C) ISS internal organisation

As an example, we present the general structure of the MIPS R3000 ISS, as depicted in the 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** et **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. If, at the end of cycle (i) 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 (i+1).

From an implementation point of view, a specific ISS is implemented by a class **processorIss**. This class inherits the class **genericIss**, that defines the characteristics common to all ISS, including the prototypes of the access function presented in section B, that are defined as virtual functions.

D) Generic cache controler

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

contains two 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 mapping, 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<2>  >  type ;  // instruction access type
    sc_in<sc_dt::sc_uint<2> > mode;  // processor mode
    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<2> > mode;  // processor mode
    sc_in<sc_dt::sc_uint<32> >  wdata;  // data to be written
    sc_in<sc_dt::sc_uint<32> >  adr;  // data address
    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 processor - is specified by the template parameter **iss_t**.

To communicate with the **VciXcache**, the **IssWrapper** class contains two member variables **p_icache**, of type **IcacheProcessorPort** and **p_dcache**, of type **DcacheProcessorPort**. It contains also N member variables **p_irq[i]**, of type **sc_in<bool>**, representing the interrupt ports. The number N of interrupt ports depends on the wrapped ISS, an is defined by the **n_irq** member variable of the **iss** object.

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[iss_t ::n_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),
    p_icache("icache"),
    p_dcache("dcache"),
    p_resetn("resetn"),
    p_clk("clk"),
    m_iss(ident)
    {
    SC_METHOD(transition);
    dont_initialize();
    sensitive << p_clk.pos();
    SC_METHOD(genMoore);
    dont_initialize();
    sensitive << p_clk.neg();
    }

private :

///////// Variables /////////
iss_t  m_iss ;
bool  m_ins_asked ;
enum InsAccessType  m_ins_type ;
uint32_t  m_ins_address ;
bool  m_mem_asked ;
enum DataAccessType  m_mem_type ;
uint32_t  m_mem_address ;
uint32_t  m_mem_wdata ;

////////////////////////
void transition()
    {
    if ( ! p_resetn.read() ) {
        m_iss.reset();
        return;
    }
    bool frozen = false;
    m_iss.getDataRequest(m_mem_asked,
                m_mem_type,
                m_mem_address,
                m_mem_wdata );
    m_iss.getInstructionRequest(m_ins_asked,
                m_ins_type,
                m_ins_address );
    if ( m_ins_asked ) {
        if ( p_icache.frz.read() ) frozen = true;
        else m_iss.setInstruction(p_icache.berr, p_icache.ins.read())
    }
    if ( m_mem_asked ) {
        if ( p_dcache.frz.read()) frozen = true;
        else  m_iss.setRdata(false, p_dcache.rdata.read());
        }
    if ( frozen || m_iss.isBusy() ) {   //  Processor frozen or busy
        m_iss.nullStep();
```

```
    } else {                    // Execute one instruction:
        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);
        m_iss.step();
        } // end transition()

////////////////////////////
void genMoore()
{
p_icache.req = m_ins_asked;
p_icache.type = m_ins_type ;
p_icache.mode = m_ins_mode ;
p_icache.adr = m_ins_addr;
p_dcache_req = m_mem_asked ;
p_dcache_type = m_mem_type ;
p_dcache_mode = m_mem_mode ;
p_dcache.adr = m_mem_addr;
p_dcache.wdata = m_mem_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 controler itself, and it is implemented as the class **VciXcache**. This class contains the SC_THREAD **execLoop()** implementing the PDES process, and the **m_time** member variable implementing the associated local clock. The class **VciXcache** inherit the class **!Tlmt::ModuleBase**, that is the basis for all TLM-T modules. This class contains the member variable ???iss??? representing the processor ISS. The type of the ???iss??? variable is defined by the template parameter ???iss_t???.

FIGURE 4

The class ??? VciXcache??? contain a member variable p_vci, of type ???VciInitPort???, to send VCI command packets, and receive VCI response packets. This class contains also N member variables ???p_irq[i]???, of type ??? IrqInPort???, representing the interrupt inputs. The number N of interrupt ports depends on the wrapped ISS, an is defined by the ???n_irq??? member variable of the ???iss??? object.

The ???execLoop()??? function contains an infinite loop. One iteration in this loop corresponds to one cycle for the local clock, or more in case of MISS, as the thread is suspended in case of MISS.

The cache behavior is specifically described by the ???cacheAccess()??? method, that is a member variable of the class ??? VciXcache???. This function is called in the main execution loop (i.e. at each cycle). This function has the following prototype :

```
void cacheAccess(icache_request_t  *ireq,
                 dcache_request_t  *dreq,
                 xcache_response_t  *rsp)
```

The ???icache_request_t???, ???dcache_request_t???, and ???xcache_response_t??? classes represent the instruction and data requests, and the cache response respectively :

```
class icache_request_t {
bool  valid ;
enum ins_access_type  type ;
enum access_mode  mode ;
uint32_t  address ;
}
class dcache_request_t {
bool  valid ;
```

```
enum data_access_type  type ;
enum access_mode  mode ;
uint32_t  address ;
uint32_t  wdata ;
}
class xcache_response_t {
bool  iber ;
uint32_t  instruction ;
bool  dber ;
uint32_t  rdata ;
}
```

The ???cacheAccess()??? function détermines the actions to be done. In case of data or instruction MISS MISS, the ???cacheAccess()??? function send the proper VCI command packet on the ???p_vci??? port., and the ???exedcLoop??? thread is suspended. In case of data write, the the ???cacheAccess()??? function send the proper VCI command packet on the ???p_vci??? port., but the ???exedcLoop??? thread is not suspended.

At each iteration in the execution loop, the ???cacheAccess()??? method updates the local clock (variable ???m_time???) :

- 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 containeded in the VCI response packet in case of MISS

The TLM-T model for the VciXcache module is presented below :

```
template<typename iss_t, typename vci_param>
class VciXcache<iss_t> : tlmt ::BaseModule {

public :
/////// ports ///////
VciInitiatorPort<vci_param>  p_vci ;
IrqInPort  p_irq[iss_t ::n_irq] ;

/////// constructor /////
VciXcache (sc_module_name  name,
          uint32_t  initiatorIndex,
          uint32_t  processorIdent,
          uint32_t  lookahead,
          uint32_t  dcache_nlines,
          uint32_t  dcache_nwords,
          uint32_t  icache_nlines,
          uint32_t  icache_nwords)
    p_vci(« vci », this, &VciXcache::rspReceived, &m_time) ,
    for (uint32_t i = 0 ; i < iss_t ::n_irq ; i++) { p_irq[i] (« irq », i, this, &VciXcache::irq
    BaseModule(name),
    m_iss(processorIdent),
    m_time(0)
    {
    m_initiator_index = initiatorIndex ;
    m_counter = 0 ;
    m_lookahead = lookahead ;
    m_icache_nlines = icache_nlines ;
    mi_icache_nwords = icache_nwords ;
    m_dcache_nlines = dcache_nlines ;
    m_dcache_nwords = dcache_nwords ;
    SC_THREAD(execLoop) ;
    } // end constructor

private :
///////  member variables
tlmt_time  m_time ;
```

```
iss_t  m_iss ;
uint32_t  m_dcache_nlines ;
uint32_t  m_dcache_nwords ;
uint32_t  m_icache_nlines ;
uint32_t  m_icache_nwords ;
uint32_t  m_initiator_index ;
uint32_t         m_lookahead ;
uint32_t  m_counter ;
bool  m_irqpending[iss_t ;;n_irq];
uint32_t        m_irqtime[iss_t ::n_irq] ;
vci_cmd_t m_cmd ;
////////////////// thread
void execLoop()
    {
    icache_request_t  icache_req ;              // The Icache request
    dcache_request_t  dcache_req ;              // The Dcache request
    xcache_response_t  xcache_rsp ;     // The Xcache response
    uint32_t  irqword ;
    while(1) {
        // execute one cycle
        if (m_iss.isBusy() {
            m_iss.nullStep() ;
        } else {
            m_iss.getInstructionRequest(icache_req.valid,
                        icache_req.type,
                        icache_req.mode,
                        icache_req.address) ;
            m_iss.getDataRequest(dcache_req.valid,
                        dcache_req.type,
                        dcache_req.mode,
                        dcache_req.address,
                        dcache_req.wdata)
            xcacheAccess(&icache_req, &dcache_req, &xcache_rsp) ;
            m_iss.setInstruction(xcache_rsp.iber, xcache_rsp.instruction) ;
            if(dcache_req.isRead()) m_iss.setRdata(xcache_rsp.dber, xcache_rsp.rdata) ;
            irqword = 0 ;
            for ( size_t i = 0 ; i < iss_t ::n_irq ; i++) {
                if( m_irqpending[i] && m_irqtime[i] <= get_time()) irqword  |= (1<<i);
               }
            m_iss.setIrq(irqword) ;
            m_iss.step() ;
        } // end cycle
        // lookahead management
        m_counter++ ;
        if (m_counter >= m_lookahead) {
            m_counter = 0 ;
             wait(SC_ZERO_TIME) ;
        } // end if lookahead
        } // end while(1)
    } // end execLoop()

////////////////////////////////////////////////////
void cacheAccess(icache_request_t ireq,
                dcache_request_t dreq,
                xcache_response_t rsp)
    {
    } // end cacheAccess()


////////////////////////////
void  rspReceived(vci_rsp_t rsp,
              uint32_t  time)
    {
    } // end rspReceived()

////////////////////////////
```

```
void  irqReceived(bool val,
              uint32_t  time
              size_t index)
    {
    m_irqpending[index] = val ;
    m_irqtime[p_irq[index] = time ;
    } // end irqReceived()
```