Writing efficient Cycle-Accurate, Bit-Accurate SystemC simulation models for SoCLib

Author: Alain Greiner

A) Introduction

This manual describes the modeling rules for writing "cycle-accurate / bit-accurate" SystemC simulation models for SoCLib. Models complying with those rules can be used with the "standard" OSCI simulation engine (SystemC 2.x), but can be used also with others simulations engines, such as SystemCASS, which is optimized for such models.

Besides you may also want to follow the general SoCLib rules.

Those CABA modeling rules are based on the "Synchronous Communicating Finite State Machines" theory. The idea is to force the "event driven" SystemC simulation engine to run as a cycle based simulator.

A given hardware architecture is obtained by direct instantiation of hardware modules, connected by signals. A given architecture can contain several instances of the same module. Each module is described as one (or several) synchronous FSM(s).

If all internal register are clearly identified, any FSM can be described by three types of functions:

- The **transition function** computes the next values of the register, depending on the current values of the register and the values of the input ports signals.
- The **genMoore function** computes the values of those output port signals that depend only on the internal registers.
- The **genMealy function(s)** computes values of those output port signals that depend both on the internal registers AND the values of the input port signals.

In this figure we represented a single FSM, but a SoCLib component contains generally several small FSMs running in parallel inside a single module. This internal parallelism should be properly described.

B) CABA rules

B1) Components

A SoCLib CABA simulation model for an hardware component `XXX` is generally described as a class derived from the soclib::caba::BaseModule class.

At least two files are associated to each hardware component:

- `XXX.h` describes the component interface, internal registers, and structural parameters.
- `XXX.cpp` contains the code of the methods associated to this component.

B2) VCI Interface

In order to enforce interoperability, all hardware architectures build with the SoCLib component library will communicate through a "flat", shared address space, and all system interconnect will respect the VCI advanced standard. Therefore, most SoCLib components have VCI interfaces :

Any SoCLib hardware components having a VCI interface is called a *VCI component*, and must include at least one of the the two following files, defining the VCI advanced ports :

- trunk/soclib/soclib/communication/vci/caba/source/include/vci_target.h?
- trunk/soclib/soclib/communication/vci/caba/source/include/vci_initiator.h?

As VCI signals can have variable widths, all VCI components must be defined with templates. The trunk/soclib/soclib/communication/vci/caba/source/include/vci_param.h? file contains the definition of the VCI parameters object. This object must be passed as a template parameter to the component.

A typical VCI component declaration is:

```
#include "base_module.h"
#include "vci_target.h"

namespace soclib { namespace caba {

template<typename vci_params>
class VciExampleModule
    : soclib::caba::BaseModule
{

};

}}
```

The SystemC top cell defining the system architecture must include the following file, defining the advanced VCI signals :

- trunk/soclib/soclib/communication/vci/caba/source/include/vci_signals.h?.

A SoCLib hardware component that has no VCI interface should use a dedicated VCI wrapper in order to be connected to the VCI interconnect.

B3) Address space segmentation

In a shared memory architecture, the address space segmentation (or memory map) is a global characteristic of the system. This memory map must be defined by the system designer, and is used by both software, and hardware

components.

Most hardware components use this memory map:

- VCI interconnect components contain a *routing table* used to decode the VCI address MSB bits and route VCI commands to the proper targets.
- VCI target components must be able to check for segmentation violation when receiving a command packet. Therefore, the base address and size of the segment allocated to a given VCI target must be *known* by this target.
- A cache controller supporting uncached segments can contain a *cacheability table* addressed by the VCI address MSB bits.

In order to simplify the memory map definition, and the hardware component configuration, a generic object, called *mapping table* has been defined in mapping_table.h?. This is an associative table of memory segments. Any segment must be allocated to one single VCI target. The segment object is defined in segment.h?, and contains five attributes:

```
const std::string   m_name;           // segment's name
addr_t              m_base_address;   // base address
size_t              m_size;           // size (bytes)
IntTab              m_target_index;   // VCI target index
bool                m_cacheability;   // cacheable
```

Any hardware component using the memory map should have a constant reference to the mapping table as constructor argument.

B4) Component definition

The component *XXX.h* file contains the following informations

Interface definition A typical VCI target component will contain the following ports:

```
sc_in<bool>                          p_resetn;
sc_in<bool>                          p_clk;
soclib::caba::VciTarget<vci_param>   p_vci;
```

Internal registers definition

All internal registers should be defined with the *sc_signal* type.

This point is a bit tricky: It allows the model designer to benefit from the delayed update mechanism associated by SystemC to the sc_signal type. When a single module contains several interacting FSMs, it helps to write the `Transition()`, as the registers values are not updated until the exit from the transition function. Conversely, any member variable declared with the `sc_signal` type is considered as a register.

A typical VCI target will contain the following registers :

```
sc_signal<int>                           r_vci_fsm;
sc_signal<typename vci_param::trdid_t>   r_buf_trdid;
sc_signal<typename vci_param::pktid_t>   r_buf_pktid;
sc_signal<typename vci_param::srcid_t>   r_buf_srcid;
```

`typename vci_param::trdid_t` and others are generically-defined VCI field types

Structural parameters definition

All structural parameters should be be defined as member variables. The values are generally defined by a constructor argument. Instance name is stored in soclib::common::BaseModule?, inherited by soclib::caba::BaseModule?. For example, a VCI target will contain a reference to the the assigned segment, in order to check possible segmentation errors during execution.

```
const soclib::common::Segment   m_segment;
```

B5) Constructor & destructor

Any hardware component must have an instance name, and most SoCLib component must have a VCI target or VCI initiator index. Moreover, generic simulation models can have structural parameters. The parameter values must be defined as constructor arguments, and used by the constructor to configure the hardware ressources. A constructor argument frequently used is a reference on the soclib::common::MappingTable?, that defines the segmentation of the system address space. A typical VCI component will have the following constructor arguments:

```
VciExampleModule(
    sc_module_name                      insname,
    const soclib::common::IntTab        &index,
    const soclib::common::MappingTable &mt);
```

In this example, the first argument is the instance name, the second argument is the VCI target index, and the third argument is the mapping table.

Moreover, the constructor must define the sensitivity list of the Transition(), genMoore() and genMealy() methods, that are described below.

- sensitivity list of the transition() method contains only the clock rising edge.
- sensitivity list of the genMoore() method contains only the clock falling edge.
- sensitivity list of the genMealy() method contains the clock falling edge, as well as all input ports thare in the support of the Mealy generation function.

Be careful: the constructor should NOT initialize the registers. The register initialization must be an hardware mechanism explicitly described in the Transition function on reset condition.

B6) member functions

The component behaviour is described by simple member functions. The type ot those methods (Transition, genMoore, or genMealy) is defined by the sensitivity lists, as specified in B5.

transition() method

For each hardware component, there is only one `Transition()` method. It is called once per cycle, as the sensitivity list contains only the clock rising edge. This method computes the next values of the registers (variables that have the `sc_signal` type). No output port can be assigned in this method. Each register should be assigned only once.

genMoore() method

For each hardware component, there is only one `genMoore()` method. It is called once per cycle, as the sensitivity list contains only the clock falling edge. This method computes the values of the Moore output ports. (variables that have the `sc_out` type). No register can be assigned in this method. Each output port can be assigned only once. No input port can be read in this method

genMealy() methods

For each hardware component, there is zero, one or several `genMealy()` methods (it can be useful to have one separated `genMealy()` method for each output port). These methods can be called several times per cycle. The sensitivity list can contain several input ports. This method computes the Mealy values of the ouput ports, using only the register values and the input ports values. No register can be assigned in this method. Each output port can be assigned only once.

C) Complete example

C1) Component definition

Let's take the soclib::caba::VciLocks? component definition and comment it.

```
#include <systemc.h>
#include "caba/util/base_module.h"
#include "caba/interface/vci_target.h"
#include "common/mapping_table.h"

// Have this component in the soclib::caba namespace
namespace soclib { namespace caba {

// Here we pass the VCI parameters as a template argument. This is intended because VCI paramete
// change data type widths, therefore change some compile-time charcteristics.
template<typename vci_param>
class VciLocks
    : public soclib::caba::BaseModule
{

    // We have only one FSM in this component. It handles the
    // VCI target port. The states are:
    enum vci_target_fsm_state_e {
        IDLE,
        WRITE_RSP,
        READ_RSP,
        ERROR_RSP,
    };

    // The registers must have the sc_signal type
    sc_signal<int>                          r_vci_fsm;  // FSM state register
    sc_signal<typename vci_param::srcid_t> r_buf_srcid;
    sc_signal<typename vci_param::trdid_t> r_buf_trdid;
    sc_signal<typename vci_param::pktid_t> r_buf_pktid;
    sc_signal<typename vci_param::eop_t>   r_buf_eop;
    sc_signal<bool>                         r_buf_value;
    sc_signal<bool>                        *r_contents; // the locks array is allocated in constr

    // The structural constants
    const soclib::common::Segment m_segment; // segment assigned to this peripheral

protected:
    // Mandatory SystemC construct
    SC_HAS_PROCESS(VciLocks);

public:
    // The ports
    sc_in<bool> p_resetn;
    sc_in<bool> p_clk;
    soclib::caba::VciTarget<vci_param> p_vci;

    // Constructor & descructor, explained above
```

```
    VciLocks(
        sc_module_name insname,
        const soclib::common::IntTab &index,
        const soclib::common::MappingTable &mt);
    ~VciLocks();

private:
    // The FSM functions
    void transition();
    void genMoore();
};

// Namespace closing
}}
```

C2) Component implementation

Here is the soclib::caba::VciLocks component implementation:

```
#include "caba/target/vci_locks.h"

// Namespace, again
namespace soclib { namespace caba {

// This macro is an helper function to factor out the template parameters
// This is useful in two ways:
// * It makes the syntax clearer
// * It makes template parameters changes easier (only one line to change them all)
// x is the method's return value
#define tmpl(x) template<typename vci_param> x VciLocks<vci_param>

// The /**/ is a hack to pass no argument to a macro taking one. (constructor has no
// return value in C++)
tmpl(/**/)::VciLocks(
    sc_module_name insname,
    const soclib::common::IntTab &index,
    const soclib::common::MappingTable &mt)
// This is the C++ construct for parent construction and
// member variables initialization.
// We initialize the BaseModule with the component's name
    : soclib::caba::BaseModule(insname),
// and get the segment from the mapping table and our index
      m_segment(mt.getSegment(index))
{

    // There is one lock every 32-bit word in memory. We
    // allocate an array of bool for the locks
    r_contents = new sc_signal<bool>[m_segment.size()/4];

    // Sensitivity list for transition() and genMoore(), no genMealy()
    // in this component
    SC_METHOD(transition);
    dont_initialize();
    sensitive << p_clk.pos();

    SC_METHOD(genMoore);
    dont_initialize();
    sensitive << p_clk.neg();
}

tmpl(/**/)::~VciLocks()
{
    // Here we must delete dynamically-allocated data...
    delete [] r_contents;
```

```
}

tmpl(void)::transition()
{
    // On reset condition, we initialize the component,
    // from FSMs to internal data.
    if (!p_resetn) {
        for (size_t i=0; i<m_segment.size()/4; ++i)
            r_contents[i] = false;
        r_vci_fsm = IDLE;
        return;
    }

    // We are not on reset case.

    // Take the address, transform it into an index in our locks table.
    typename vci_param::addr_t address = p_vci.address.read();
    uint32_t cell = (address-m_segment.baseAddress())/4;

    // Implement the VCI target FSM
    switch (r_vci_fsm) {
    case IDLE:
        if ( ! p_vci.cmdval.read() )
            break;
        /*
         * We only accept 1-word request packets
         * and we check for segmentation violations
         */
        if ( ! p_vci.eop.read() ||
             ! m_segment.contains(address) )
            r_vci_fsm = ERROR_RSP;
        else {
            switch (p_vci.cmd.read()) {
            case VCI_CMD_READ:
                r_buf_value = r_contents[cell];
                r_contents[cell] = true;
                r_vci_fsm = READ_RSP;
                break;
            case VCI_CMD_WRITE:
                r_contents[cell] = false;
                r_vci_fsm = WRITE_RSP;
                break;
            default:
                r_vci_fsm = ERROR_RSP;
                break;
            }
        }
        r_buf_srcid = p_vci.srcid.read();
        r_buf_trdid = p_vci.trdid.read();
        r_buf_pktid = p_vci.pktid.read();
        r_buf_eop = p_vci.eop.read();
        break;

    // In those states, we only wait for response to be accepted.
    case WRITE_RSP:
    case READ_RSP:
    case ERROR_RSP:
        if ( p_vci.rspack.read() )
            r_vci_fsm = IDLE;
        break;
    }
}

tmpl(void)::genMoore()
{
    // This is an helper function defined in the VciTarget port definition
```

```
    p_vci.rspSetIds( r_buf_srcid.read(), r_buf_trdid.read(), r_buf_pktid.read() );

    // Depending on the state, we give back the expected response.
    switch (r_vci_fsm) {
    case IDLE:
        p_vci.rspNop();
        break;
    case WRITE_RSP:
        p_vci.rspWrite( r_buf_eop.read() );
        break;
    case READ_RSP:
        p_vci.rspRead( r_buf_eop.read(), r_buf_value.read() );
        break;
    case ERROR_RSP:
        p_vci.rspError( r_buf_eop.read() );
        break;
    }

    // We only accept commands in Idle state
    p_vci.cmdack = (r_vci_fsm == IDLE);
}
}}
```

Component instanciation could be (`template_inst.cc`):

```
#include "caba/target/vci_locks.cc"
template class soclib::caba::VciLocks<soclib::caba::VciParams<4,1,32,1,1,1,8,1,1,1> >;
```

Command line:

```
g++ -c -o obj.o -I/path/to/soclib/systemc/src -I/path/to/soclib/systemc/include template_inst.cc
```