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 simulation engines, such as [SystemCass SystemCASS], which is optimized for such models.

Those 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 Moore generation function computes the values of those output port signals that depend only on the internal registers.
- The Mealy generation functions 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.

Components

A SoCLib CABA 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.cc` contains the code of the methods associated to this component.

Namespaces

SystemC is built upon C++. We can benefit from C++ constructs. Namespaces allows us to create unambiguous names while keeping them short and clean. SoCLib defines some namespaces:

- `soclib`
- `soclib::common`
- `soclib::caba`
- `soclib::tlmt`

Component indexation

In a VCI-based architecture, all initiators must be indexed. Targets are identified by their assigned address space segment. However, for simplification purposes, we'll also give an index to the targets. Index space for targets is different from index space for initiators.

- The target index is used by interconnects, that decode the VCI address MSB bits to get the target index.
- The initiator index is used by the interconnect components to route the response packets.

For the initiators, the index is the VCI SRCID value.

Indexes can be

- a simple scalar index, in case of a *flat* interconnect
- a composite index in case a hierarchical two level interconnect where each component is identified by two scalars: (cluster_index, local_index).

common/int_tab.h defines an utility class storing list of indexes. All indexes are `IntTab`s.

VCI Interface

In order to enforce interoperability between components, all SoCLib hardware components should respect the advanced VCI standard. Any component having a VCI interface must include one of the the two following files

- caba/interface/vci_target.h
- caba/interface/vci_initiator.h

The advanced VCI signals are defined in caba/interface/vci_signals.h.

As several VCI signals can have variable widths, all VCI components must be defined with templates. The caba/interface/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 "caba/util/base_module.h"
#include "caba/interface/vci_target.h"

namespace soclib { namespace caba {

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

};
```

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 addresses 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* must contain a *cacheability table* addressed by the 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 common/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 common/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 attribut
```

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

Naming conventions

The following conventions are not mandatory, but can help to read the code.

- All port names should be prefixed with `p_`
- All internal register names should be prefixed with `r_`
- All member variables should be prefixed with `m_`

Component ressources

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

All internal registers must be defined with the type *sc_signal*

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.

In order to improve the code readability, all internal registers should be prefixed with `r_`.

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_srcid;
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

All structural parameters must 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 a VCI target, assigned segment should be copied in order to check commands.

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

Constructor & destructor

Any hardware component must have an instance name, and most SoCLib component must have a VCI index. Moreover, generic simulation models can have structural parameters defined as arguments in the constructor, 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.

component behaviour

The component is described by simple sc_methods as member functions.

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() method

For each hardware component, there is zero, one or several `genMealy()` methods (one 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. This method can use automatic variables. It can be missing if there is no Mealy output.

Complete example

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 intrinsics
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 register holding the FSM state:
    sc_signal<int> r_vci_fsm;
```

```
    // Some registers used to save useful data between command & response
    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;

    // Pointer on the table of locks (allocated in the constructor)
    sc_signal<bool>             *r_contents;

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

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
}}
```

And the 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 [] m_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<r_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);
}

}}
```