Introduction

This manual describes the modeling rules for writing "cycle-accurate / bit-accurate" SystemC simulation models. Those models can be used with the "standard" OSCI simulation engine (SystemC 2.0), but can be used also with others simulation engines, such as SystemCASS, which is optimized for models complying with those rules.

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 instanciation 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.

How to write simulation models

Component's module

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:

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

Component indexation

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

The target index is used in the routing table implemented by the interconnect components in order to choose destination port for command packet. The initiator index is used by the interconnect components to route the response packets.

For the initiators, the index corresponds to the VCI SRCID value.

This index can be a simple scalar index, in case of a *flat* interconnect, or it can be a composite index in case a hierarchical two level interconnect where each component is identified by two indexes : (cluster_index,

local_index).

VCI Interface

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

- trunk/soclib/systemc/include/caba/interface/vci_target.h
- trunk/soclib/systemc/include/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.

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. This routing table is implemented as a ROM.
- 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.

Constructor arguments

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, that must be defined as arguments in the constructor. A typical VCI component will have the following constructor arguments:

```
VciLocks(
```

```
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.

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. A typical SoCLib component will contain an instance name, and a reference to the segment allocated to this component :

```
sc_module_name                  m_name;
```

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

Constructor & descructor

The first constructor argument must be the instance name. Other arguments can be identifiers (such as a target VCI index or initiator VCI index). The constructor must configure some hardware ressources, such as the address decoding ROM that exists in any VCI interconnect. An argument frequently used is a reference on the Soclib segment table, that defines the segmentation of the system address space.

Moreover, the constructor must define the sensitivity list of the Transition,(), genMoore() and genMealy() methods, that are described below. The sensitivity list of the Transition () method contains only the clock rising edge. The sensitivity list of the genMoore()n method contains only the clock falling edge. The 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 initialisation must be an hardware mechanism explicitely described in the Transition function on reset condition.

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.