System Modeling and SystemC

Charles Wilson
www.xtreme-eda.com
info@xtreme-eda.com
Version 1.6
2008-Nov-04

© 2008 XtremeEDA USA Corporation - Version 080721.10

XtremeEDA USA - SystemC Specialists

• Founded 2003 under the name Eklectically Inc. (later DBA ESLX Inc.)
  – Broad Background (Hardware/Software/Methodology/Systems)
  – Active in SystemC Standardization working groups
  – Authors of book SystemC: From the Ground Up
  – Became XtremeEDA USA, a subsidiary of XtremeEDA in 2008

• Services
  – SystemC Adoption Planning
  – Methodology & Flow Definition & Development
    • General Modeling & Software Development Platforms
    • Architectural and Functional Verification
    • Behavioral Synthesis
  – Staffing
    • Mentoring
    • Peak staffing needs
  – Training & Quick Ramp Mentoring

• Clients include small “startups” to Fortune 500

Let our experts help your company be successful with SystemC
Objectives - System Modeling and SystemC

• Provide a quick overview of the topics
  – Several fast paced hours of lecture
  – What is system modeling
  – How does SystemC fit
  – Brief introduction to SystemC syntax
• NOT a complete tutorial
  – See books or call us for in-depth training
  – Use this as a guideline on what to learn

Topics

• System Design Context
  – General Methodology
  – Refinement
  – Benefits
• SystemC Overview
• Anatomy of an SC_MODULE
• SystemC Simulation Kernel
• An Example
• Some Homework
Languages Usage

Requirements
- Algorithm and Architectural
- Functional and Software Development
- Behavioral
- SoC Verification
- IP Verification
- RTL
- Gates
- Transistors

Matlab & C/C++

SystemC

Verilog

VHDL

SystemC

Vera

PSL

• Terms used to characterize models
  - UnTimed Functional (UTF)
  - Timed Functional (TF)
  - Bus Cycle Accurate (BCA)
  - Pin Cycle Accurate (PCA)
  - Register Transfer (RT) accurate

Modeling Characteristics and Models

• Model types
  - System Architectural Model (SAM)
  - System Performance Model (SPM)
  - Transaction Level Model (TLM)
  - Functional Level Model (FLM)
  - System Level Model (SLM)
  - Behavioral Level Model (BLM)
  - Register Transfer Level (RTL) model
Model Refinement

SystemC allows model refinement to proceed independently for functionality and interface.

TLM Based ESL Methodology

Requirements Definition → Requirements Document → System Architecture Model Development

Transaction Level Model Development → Hardware Refinement → RTL

Hardware Verification Environment Development → RTL to GDSII Flow
### ESL Impacts on Schedule - before

<table>
<thead>
<tr>
<th>Disparate Teams:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
</tr>
</tbody>
</table>

- **Architectural Verification**
- **Hardware Development**
- **Hardware Verification**
- **Software Development**
- **System Integration**

### ESL Impacts on Schedule - after

<table>
<thead>
<tr>
<th>Disparate Teams:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
</tr>
</tbody>
</table>

- **Architectural Verification**
- **Model Development**
- **Hardware Development**
- **Hardware Verification**
- **Software Development**
- **System Integration**
“Architectural Verification” vs. “Implementation Verification”

• Architectural Verification
  – “Have we defined ‘the right’ architecture?”
  – “Will it enable our customers to succeed?”
  – “Have we addressed specific use case requirements?”

• Block-Level Implementation Verification
  – “Have we implemented a given piece of the architecture correctly?”
  – “Does the implementation match the specification?”

• System-Level Implementation Verification
  – “Have we implemented the complete architecture (system) correctly?”
  – “Does the implementation match the specification?”

Considered by many to be the missing Link for ESL Flows

• Several Vendors now offering solutions
  – Forte Design Systems
  – Mentor
  – Cadence
  – Agility
  – Synfora

• Takes “behavioral code” and “synthesizes” to RTL code

• Results comparable to human generated RTL
# Modeling Abstraction Levels

<table>
<thead>
<tr>
<th>Model Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithmic level (AL)</td>
<td>Abstract Major events</td>
</tr>
<tr>
<td>Programmer’s view (PV)</td>
<td>TLM – minimal bus Instruction seq.</td>
</tr>
<tr>
<td>Loosely Timed (LT)</td>
<td>TLM – generic bus</td>
</tr>
<tr>
<td>Approximately Timed (AT)</td>
<td>TLM – arch. bus Performance Anal.</td>
</tr>
<tr>
<td>Cycle Approximate (CA)</td>
<td>TLM – arch. bus Cycle-accurate I/F</td>
</tr>
<tr>
<td>RT level (RT)</td>
<td>Signal/Bit Cycle-accurate</td>
</tr>
</tbody>
</table>

## TLM – Motivations

- **Speed**
  - Quick turn-around for architectural exploration
  - Appropriate for software development
  - Regression-style verification
- **Independently refinable**
  - Independently refine functionality and communication
  - Affords traceability from Architectural Specification to Hardware Specification and implementation
- **Use of Existing Techniques**
  - TLM is already widely used for verification (not just SystemC)
  - TLM Interface Spec v1.0 April 2005
  - TLM Specification v2.0 approved June 2008
TLM – Model Mix and Match

Relative Performance
Topics

• System Design Context
  – General Methodology
  – Refinement
  – Benefits
• SystemC Overview
• Anatomy of an SC_MODULE
• SystemC Simulation Kernel
• An Example
• Some Homework

SystemC Organizations

• IEEE Standards Group 1666
• OSCI - systemc.org
  – LWG (Language Working Group)
  – VWG (Verification Working Group)
  – SWG (Synthesis Working Group)
  – TWG (Transaction Level Modeling Working Group)
• GreenSOCs.org
  – Boost.org equivalent
• Users Groups
  – European SystemC User’s Group
  – North American SystemC User’s Group
  – Latin America SystemC User’s Group
  – India SystemC User’s Group
Websites

- IEEE Standards Association [standards.ieee.org/announcements/pr_p1666.html]
- OSCI [www.systemc.org]
- NASCUG [www.nascug.org]
- ESCUG [www-ti.informatik.uni-tuebingen.de/~systemc/systemc.html]
- GreenSOCs [www.greensocs.org]
- Boost [www.boost.org]
standards.ieee.org

SystemC Books
Can C++ be used as is?

No - C/C++ lacks

- **Notion of simulated time**
  Time sequenced operations

- **Concurrency**
  Hardware and systems are inherently concurrent, *i.e.* they operate in parallel

- **Hardware data types**
  Bit type, bit-vector type, multi-valued logic type, signed and unsigned specific width integer types and fixed-point types
**SystemC C++ Classes**

Enable C++ without extending the language (syntax) - use classes and templates

- Communication
- Notion of Time
- Concurrency
- Hardware Data Types

- Channels, events
- Clocks, sc_time
- Processes
- bit vectors, arbitrary precision signed and unsigned integers, fixed-point numbers

**SystemC Simulation & Testing Functionality**

Contains functionality for modular design, easy integration, testing and simulation management

- Hierarchy
- Interoperability
- Test Bench
- Running

- Modules
- TLM Standard
- Verification library
- Scheduler
Using SystemC With OSCI PoC Simulator

Standard C++ development environment

class library and simulation kernel

header files

libraries

compiler

linker

debugger

"make"

executable = simulator

source files for system and testbenches

DSP Interface

IP-Core

ASIC

Topics

• System Design Context
  – General Methodology
  – Refinement
  – Benefits
• SystemC Overview
• Anatomy of an SC_MODULE
• SystemC Simulation Kernel
• An Example
• Some Homework
## SystemC Language Architecture

<table>
<thead>
<tr>
<th>Layered Libraries</th>
<th>Primitive Channels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Verification Library, etc.</td>
</tr>
<tr>
<td></td>
<td>Signal, Mutex, Semaphore, FIFO, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core Language</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modules</td>
<td>4-valued Logic type</td>
</tr>
<tr>
<td>Ports</td>
<td>4-valued Logic Vectors</td>
</tr>
<tr>
<td>Processes</td>
<td>Bits and Bit Vectors</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Arbitrary Precision Integers</td>
</tr>
<tr>
<td>Channels</td>
<td>Fixed-point types</td>
</tr>
<tr>
<td>Events &amp; Time</td>
<td>C++ user-defined types</td>
</tr>
</tbody>
</table>

### C++ Language Standard

### A Simple Module – conceptual (not SystemC)

**Connectivity**

- **camera**
  - image
  - fifo
  - jpeg

**Hierarchy**

- **camera**
  - image
  - fifo
  - jpeg

**Verilog**

```verilog
module camera(ccd_p,out);
  // capture picture
endmodule

module jpeg(raw,jpg);
  // compress image
endmodule

module fifo(in,out);
  // buffer image
endmodule

module camera(ccd_p,img_p);
  image M1(...);
  fifo CH1(...);
  image M2(...);
endmodule camera;
```
SC_MODULE Anatomy - module

```cpp
SC_MODULE(module_name)
{
    // port declarations
    // channel declarations
    // variable declarations
    // event declarations
    // process declarations
    // helper method declarations
    // module instantiations
    SC_CTOR(module_name)
    : // initialization list
    {
        // connectivity
        // process registration
    }
};
```

Module declaration

```cpp
// Filename: Camera.h
#include <systemc>
// Sub-module declarations
struct Camera
: public sc_module
{
    // Ports
    // Local channels & instances
    // Local events
    // Processes
    // Constructor
    Camera(sc_module_name nm);
    private:
        // Helper member functions
        // Local data
};
```
SC_MODULE Anatomy - ports

SC_MODULE(module_name) {
    //port declarations
    //channel declarations
    //variable declarations
    //event declarations
    //process declarations
    //helper method declarations
    //module instantiations
    SC_CTOR(module_name) :
        //..init list...
    {
        //connectivity
        //process registration
    }
};

SC_MODULE Graphical View

Ch=channel  If=interface  M=module  P=port/pointer  Pr=process

© 2008 XtremeEDA USA Corporation - Version 080721.10
**SC_PORT**

- Interface (aka API)
  - `sc_fifo<int> c;`
  - `write();` → `read();`
- `modA mA`
- `modB mB`
- `sc_port< sc_fifo_out_if<int> > pB;`
- `A_thread`
  - `pA->write(v);`
- `B_thread`
  - `v=pB->read();`

"points to the channel via the interface"

**Port Declarations**

```
sc_port<interface_type> port_name;
```

```cpp
SC_MODULE(fir_arch) {
  // Port Declarations
  sc_port< sc_fifo_in_if<double> > data_i;
  sc_port< sc_fifo_out_if<double> > data_o;
  ..
}; // end fir_arch
```

`j = data_i->read();`
`data_i->read(j);`
`data_o->write(k);`

"points to the channel via the interface"
Ports added

//Filename: Camera.h
#include <systemc>
// Sub-module declarations
struct Camera : public sc_module {
  // Ports
  sc_port<ccd_p_if> ccd_p;
  sc_port<firewire_if> img_p;
  // Local channels & instances
  // Local events
  // Processes
  // Constructor
  Camera(sc_module_name nm);
  private:
    // Helper member functions
    // Local data
};

SC_EXPORT
"exports the interface of the channel"
modA mA
sc_export<sc_fifo_in_if<int>> > pA
sc_fifo<int> c;
write();
read();
A_thread
  c.write(v);

modB mB
sc_port<sc_fifo_in_if<int> > B_thread
  pB->read();

Pointer Access
Direction of subroutine call reversed.
SC_MODULE Anatomy - channels

```c
SC_MODULE(module_name) {
    // port declarations
    // channel declarations
    // variable declarations
    // event declarations
    // process declarations
    // helper method declarations
    // module instantiations
    SC_CTOR(module_name) :
    {
        // connectivity
        // process registration
    }
};
```

Channel Declarations

```c
channel_type channel_name;
```

```c
SC_MODULE(fir_arch) {
    // Channel Declarations
    sc_fifo<double> orig_in_fifo; // stimulus to results
    sc_fifo<double> data_in_fifo; // stimulus to filter
    sc_fifo<double> data_out_fifo; // filtered data
    ...
}; // end fir_arch
```

```c
j = orig_in_fifo.read();
data_in.read(j);
data_out_fifo.write(k);
```

// Example using channels
//Filename: Camera.h
#include <systemc>
struct Camera : public sc_module {
    // Ports
    sc_port<ccd_p_if> ccd_p;
    sc_port<firewire_if> img_p;
    // Local channels & instances
    sc_fifo<image_t> CH1;
    // Local events
    // Processes
    // Constructor
    Camera(sc_module_name nm);
    private:
    // Helper member functions
    // Local data
};

SC_MODULE(module_name) { 
    //port declarations
    //channel declarations
    //variable declarations
    //event declarations
    //process declarations
    //helper method declarations
    //module instantiations
    SC_CTOR(module_name) :
        //..init list...
        { 
            //connectivity
            //process registration
        } 
};

Variable Declarations

• Simply member data – Local to all methods in module
  – C++ data types
  – User Defined data types
  – SystemC data types

```c++
SC_MODULE(fir_arch) {
  ...
  sc_uint<16> m_taps;
  unsigned m_tap;
  unsigned m_results_cnt;
  char* m_cfg_filename;
}; //end fir_arch
```

SC_MODULE Anatomy - events

```c++
SC_MODULE(module_name) {
  //port declarations
  //channel declarations
  //variable declarations
  //event declarations
  //process declarations
  //helper method declarations
  //module instantiations
  SC_CTOR(module_name) :
    //..init list...
  {
    //connectivity
    //process registration
  }
};
```
Event Declarations

- Event object
  - Event is a basic synchronization object
  - Event is used to synchronize between processes
  - Channels use events to implement blocking
  - Event has no data type, only control
  - Declared inside of a module
    - Used for synchronization between the processes inside a module
    - Declare as many as wanted

```c
SC_MODULE(fir_sys) {…
  //Event Declarations
  sc_event fir_done_evt;
…}
```

Event Notify

The `sc_event` class has the following methods:

```c
void notify( )
void notify( const sc_time& )
void notify( double, sc_time_unit )
```

```c
//within a simulation process
sc_time time_out(10, SC_MS);
…
  event1.notify();
  event2.notify(time_out);
  event3.notify(1, SC_NS);
```

- Will discuss making simulation processes “sensitive” to events later
**notify() Behaviors**

- Three notify() behaviors
  - Immediate notification
    - Causes processes which are sensitive to the event to be made immediately ready to run
    - Run in the current evaluate phase
    - Useful for modeling software systems and operating systems, which lack the concept of delta cycles
  - Delayed
    - Causes processes which are sensitive to the event to be made ready to run in the next evaluate phase
  - Timed notification
    - Causes processes which are sensitive to the event to be made ready to run at a specified time in the future

---

**SC_MODULE Anatomy - processes**

```cpp
SC_MODULE(module_name) {
    //port declarations
    //channel declarations
    //variable declarations
    //event declarations
    //process declarations
    //helper method declarations
    //module instantiations
    SC_CTOR(module_name) :
        //..init list...
        {
            //connectivity
            //process registration
        }
};
```
Simulation Processes

- Functionality is described in simulation processes
- C++ Methods “registered” with the simulation kernel
  - Simulation kernel is the ONLY legal caller
  - Called based on the sensitivity (discussed later)
  - SC_METHOD processes execute when called and return control to the calling mechanism
    - behave like ordinary C++ method
    - Verilog always block or VHDL process
  - SC_THREAD and SC_CTHREAD processes are called once, and then can suspend themselves and resume execution later
    - behave like threads
    - Verilog initial block

Process Types

- Three different process types:
  - Methods (SC_METHOD)
  - Threads (SC_THREAD)
  - Clocked Threads (SC_CTHREAD) – will be deprecated
- May have many processes inside the same module

Process Usage:
- System architectural models tend to use Threads
- System Performance models tend to use primarily Threads
- Transaction Level Models tend to use primarily Threads
- Behavioral synthesis uses clocked Threads only
- RTL models use Methods
- Test benches may use all process types
SC_THREAD

- Runs only when invoked by the SystemC scheduler (part of SystemC kernel)
- Invoked based upon:
  - Start of simulation
  - Sensitivity
    - To event(s) in channels connected to ports
    - To event(s) in local channels
    - Local declared events (sc_event)
    - To time delays
- When SC_THREAD process is invoked:
  - Statements are executed until a wait statement is encountered
  - At the next wait() statement, the process execution is suspended
  - At the next reactivation, process execution starts from the statement following the wait

SC_THREAD

- Implemented as a method
  - Takes no arguments
  - Supplies no return value
  - Uses wait() to suspend
- Typically implemented with an infinite loop
  - Ensures that the process can be repeatedly reactivated
  - Allows for suspension and reactivation at different points
  - If no infinite loop then process is executed only once
    - May be desired - like in a test bench for example

```c
void main_thread(void)
{
  for(;;) {
    // Behavior
    wait(args...);
  }//end forever
  //Completely finished
  return;
}
```
SC_METHOD

- Runs only when invoked by the SystemC scheduler (part of SystemC kernel)
- Invoked based upon:
  - Start of simulation
  - Sensitivity
    - To event(s) in channels connected to ports
    - To event(s) in local channels
    - Local declared events (sc_event)
    - To time delays
- When SC_THREAD process is invoked:
  - Once invoked
    - Entire body of the process is executed
    - Must return
  - Upon completion returns execution control back to the simulation kernel

SC_METHOD

- Implemented as a method
  - Takes no arguments
  - Supplies no return value
  - Re-invoked as needed
  - May use next_trigger()
- May not use infinite loop
  - Execution would never terminate - hang
  - May not have wait()
  - Uses next_trigger()
- Local variables redefined each time invoked.
  - Need to save the state of the process in member variables

void my_method(void)
{
    //Behavior
    int local_i;
    next_trigger(args...);
    return;
    //until re-invoked
}
SC_MODULE Anatomy - subroutines

SC_MODULE(module_name)
{
    // port declarations
    // channel declarations
    // variable declarations
    // event declarations
    // process declarations
    // helper method declarations
    // module instantiations
    SC_CTOR(module_name)
    : // initialization list
    {
        // connectivity
        // process registration
    }
};

Subroutines - Helper Processes/Subroutines

- C++ Methods (Member functions)
  - Same C++ rules
- Called from Simulation Processes (or the Constructor)
- Adds readability and reusability
- NOTE: Can use ordinary C-functions too; however,
  - Cannot access module data directly
  - Pass explicit arguments
**Simulation Process - Declaration**

- A processes are C++ functions (usually within module)
- Declared functions that take and return **void**
- Need to “register” with the simulation kernel

```cpp
SC_MODULE(fir_sys) {
    // Simulation Processes
    void stimulus_thread(void);
    void fir_thread(void);
    void results_method(void);
    // Helper Processes
    void read_cfg(void);
    ... 
}; //end fir_sys
```

**Simulation Process - Implementation**

Recommended Style - define implementation in a separate file (**module_name**.cpp)

```cpp
void fir_sys::stimulus_thread(void) {
    for (int t=0; t!= STIM_PTS; ++t) {
        double data = 0.0;
        if (t==IMP_PT) data = 1.0; //impulse
        orig_in_fifo.write(data);
        data_in_fifo.write(data);
    }
}; //end fir_sys::stimulus_thread()
```

Implied wait () within sc_fifo
Simulation Process Implementation

```cpp
void fir_sys::results_method(void) {
while(data_out_fifo.num_available() > 0) {
    m_results_cnt++;
    cout
    "DATA: "
    m_results_cnt << ""]"
    << "orig_in_fifo.read()"
    << "end;
    } // endwhile
    next_trigger();
} // end fir_sys::results_method()
```

Sub-module instances added

```cpp
//Filename: Camera.h
#include <systemc>
// Sub-module declarations
struct Camera : public sc_module {
    // Ports
    sc_port<ccd_p_if> ccd_p;
    sc_port<firewire_if> img_p;
    // Local channels & instances
    sc_fifo<image_t> CH1;
    image M1;
    jpeg M2;
    // Processes
    // Constructor
    Camera(sc_module_name nm);
    private:
    // Helper member functions
    // Local data
};
```
SC_MODULE Anatomy - Constructor

SC_MODULE(module_name) {
    //port declarations
    //channel declarations
    //variable declarations
    //event declarations
    //process declarations
    //helper method declarations
    //module instantiations
    SC_CTOR(module_name)
        : //..init list...
    {
        //connectivity
        //process registration
    }
};

Constructor

• Normal initialization (as usual in C++)
• Create and initialize an instance of a module:
  – Instance name passed to the constructor at instantiation (creation) time
  – Simulation Processes are registered and modules instantiated inside

SC_MODULE(module_name) {
    //port,channels,variables,events,processes
    // Constructor - SystemC Macro
    SC_CTOR(module_name) /* : init list */ {
        //process registration
        //declarations of sensitivity lists
        //module instantiations
        //port connection declarations
    }
};
### Constructor implementation

```cpp
//Filename: Camera.cpp
#include "Camera.h"
// Sub-module includes
// Constructor
SC_HAS_PROCESS(Camera);
Camera::Camera(sc_module_name nm)
    : sc_module(nm),
      ccd_p("ccd_p"),
      img_p("img_p"),
      CH1("CH1",1024),
      M1("M1"),
      M2("M2")
{
    // Instance elaboration
    // Connectivity
    // Process registration
}
//Continue...
```
Simulation Process Dynamic Sensitivity

- **SC_THREAD**
  - `wait(args);`
  - `wait();` implies static sensitivity
  - Immediately suspends
- **SC_METHOD**
  - `next_trigger(args);`
  - `next_trigger();` implies static sensitivity
  - Still continue execution until the process is exited
  - Last trigger “wins”
- **args**
  - Specify one or more events to wait for
  - Specify a collection of events to wait for
  - Specify an amount of time to wait
  - Events on a port or channel are “legal”

```plaintext
wait(args)
for use with SC_THREAD

sc_event e1,e2,e3;       // events
sc_time t(200, SC_NS);   // variable t of type sc_time

// wait for an event in a list of events
wait(e1);
wait(e1 | e2 | e3);       // wait on e1, e2 or e3

// wait for all events in a list
wait( e1 & e2 & e3);     // wait on e1, e2 and e3

// wait for specific amount of time
wait(200, SC_NS);         // wait for 200 ns
wait(t);                  // wait for 200 ns

// wait for events with timeout
wait(200, SC_NS, e1 | e2 | e3);
wait(t, e1 | e2 | e3);
wait(200, SC_NS, e1 & e2 & e3);
wait(t, e1 & e2 & e3);

// wait for one delta cycle
wait( 0, SC_NS );         // wait one delta cycle
wait( SC_ZERO_TIME );     // wait one delta cycle
```
next_trigger(args)
for use with SC_METHOD

```
sc_event e1,e2,e3;        // event
sc_time t(200, SC_NS);    // variable t of type sc_time
// trigger on an event in a list of events
next_trigger(e1);
next_trigger(e1 | e2 | e3);  // any of e1, e2 or e3
// trigger on all events in a list
next_trigger( e1 & e2 & e3); // all of e1, e2 and e3
// trigger after a specific amount of time
next_trigger(200, SC_NS);  // trigger 200 ns later
next_trigger(t);           // trigger 200 ns later
// trigger on events with timeout
next_trigger(200, SC_NS, e1 | e2 | e3);
next_trigger(t, e1 | e2 | e3);
next_trigger(200, SC_NS, e1 & e2 & e3);
next_trigger(t, e1 & e2 & e3);
// trigger after one delta cycle
next_trigger(0, SC_NS);     // after 1 delta cycle
next_trigger( SC_ZERO_TIME ); // after 1 delta cycle
```

SC_MODULE Anatomy – register processes

```
SC_MODULE(module_name) {
    //port declarations
    //channel declarations
    //variable declarations
    //event declarations
    //process declarations
    //helper method declarations
    //module instantiations
    SC_CTOR(module_name) :
    //..init list...
    {
        //connectivity
        //process registration
    }
};
```
Static Sensitivity

SC_CTOR(fir_sys)
: sc_module(_name)
, m_cfg_filename("control.txt")
, ...
{
  SC_THREAD(stimulus_thread);
  SC_THREAD(fir_thread);
  sensitive << data_in_fifo.data_written_event();
  SC_METHOD(results_method);
  sensitive << data_out_fifo.data_written_event();
  sensitive << event1;
  dont_initialize();
  read_cfg(); //read coefficients
} //end constructor fir_sys

SC_MODULE Anatomy - connectivity

SC_MODULE(module_name) {
  //port declarations
  //channel declarations
  //variable declarations
  //event declarations
  //process declarations
  //helper method declarations
  //module instantiations
  SC_CTOR(module_name)
  : //..init list...
  {
    //connectivity
    //process registration
  }
};
Module Instantiation

Very top level is not a module – sc_main
- NOTE: main() is used by SystemC itself
- NOTE: some simulators do not use sc_main()
- File name is usually main.cpp
- Typically instantiate a single module inside sc_main() - top

Module Instantiation Example - 1

```
SC_MODULE(ex1) {
  sc_port<sc_fifo_in_if<int>> m;
  sc_port<sc_fifo_out_if<int>> n;
  ...
  SC_CTOR(ex1) {...
}

SC_MODULE(ex2) {
  sc_port<sc_fifo_in_if<int>> x;
  sc_port<sc_fifo_out_if<int>> y;
  ...
  SC_CTOR(ex2) {...
}
```

Module Instantiation Example - 1

```cpp
SC_MODULE(ex3) {
    // Ports
    sc_port<sc_fifo_in_if<int>> a;
    sc_port<sc_fifo_out_if<int>> b;
    // Internal channel
    sc_fifo<int> ch1;
    // Instances of ex1 and ex2
    ex1 ex1_instance;
    ex2 ex2_instance;
    // Module Constructor
    SC_CTOR(ex3):
        ex1_instance("ex1_instance"),
        ex2_instance("ex2_instance")
    {
        // Named connection for ex1
        ex1_instance.m(a);
        ex1_instance.n(ch1);
        // Positional connection for ex2
        ex2_instance(ch1, b);
    }
};
```
Constructor implementation

```cpp
//Filename: Camera.cpp
#include "Camera.h"
// Constructor
SC_HAS_PROCESS(Camera);
Camera::Camera(sc_module_name nm): sc_module(nm),
    ccd_p("ccd_p"),
    img_p("img_p"),
    CH1("CH1",1024)
    M1("M1")
    M2("M2")
{
    // Connectivity
    M1.ccd_p(ccd_p); M1.out(CH1);
    M2.raw(CH1); M2.jpeg(img_p);
    // Process registration
}
//Continue...
```

sc_main Example

```cpp
#include <systemc>
#include "top.h"

int sc_main(int argc, char *argv[])
{
    sc_set_time_resolution(1, SC_FS);
    sc_set_default_time_unit(1,SC_FS);
    top TOP("TOP");
    sc_start();
    return 0;
}
```
Alternate Syntax - 1

```cpp
SC_MODULE(MODULE_NAME) {
    // port declarations
    ...
    ...
    ...
    // module instantiations
    SC_CTOR(MODULE_NAME) :
        .init list...
    {
        // simulation directives
        }
};
```

```cpp
struct MODULE_NAME :public sc_module {
    // port declarations
    // module instantiations
    SC_HAS_PROCESS(MODULE_NAME);
    module_name(
        sc_module_name name,
        // additional args...
    ) :
        sc_module(name),
        // additional init list
    {
        // simulation directives
    }
};
```

Alternate Syntax - 2

```cpp
struct ex3 : sc_module {
    // Ports
    sc_port<sc_fifo_in_if<int>> a;
    sc_port<sc_fifo_out_if<int>> b;
    // Internal channel
    sc_fifo<int> ch1;
    // Instances of ex1 and ex2
    ex1 ex1_instance;
    ex2 ex2_instance;
    // Module Constructor
    SC_CTOR(ex3):...
    {
        ex1_ptr=new ex1("ex1_instance");
        ex2_ptr=new ex2("ex2_instance");
        // Named connection
        ex1_instance.m(a);
        ex1_instance.n(ch1);
        // Positional connection
        ex2_instance(ch1, b);//Bad
    } //end constructor
}; //end ex3
```

```cpp
struct ex3 : sc_module {
    // Ports
    sc_port<sc_fifo_in_if<int>> a;
    sc_port<sc_fifo_out_if<int>> b;
    // Internal channel
    sc_fifo<int> ch1;
    // Instances of ex1 and ex2
    ex1* ex1_ptr;
    ex2* ex2_ptr;
    // Module Constructor
    SC_CTOR(ex3):...
    {
        ex1_ptr=new ex1("ex1_instance");
        ex2_ptr=new ex2("ex2_instance");
        // Named connection
        ex1_ptr->m(a);
        ex1_ptr->n(ch1);
        // Positional connection
        (*ex2_ptr)(ch1, b);//Bad
    } //end constructor
}; //end ex3
```
SystemC Language Architecture

Layered Libraries
- Verification Library, etc.

Primitive Channels
- Signal, Mutex, Semaphore, FIFO, etc.

Core Language
- Modules
- Ports
- Processes
- Interfaces
- Channels
- Events & Time
- Event-driven simulation

Data Types
- 4-valued Logic type
- 4-valued Logic Vectors
- Bits and Bit Vectors
- Arbitrary Precision Integers
- Fixed-point types
- C++ user-defined types

C++ Language Standard

SystemC Data Types

Important to use right data types in right place for simulation performance
- Use native C++ types as much as possible
  - Use sc_int<W> or sc_uint<W>
    - Up to 64 bits wide
    - Two value logic
    - Boolean and arithmetic operations on integers
  - Use sc_logic, sc_lv<W>
    - tri-state ports ("0", '1', 'X', 'Z' or "01XZxx")
    - Convert to appropriate type for computation
  - Use sc_bigint<W> or sc_biguint<W>
    - More than 64 bits wide
- Use sc_fixed<> or sc_fix() or sc_ufixed<> or sc_ufix()
  - Fixed-point arithmetic
  - Convert to sc_uint if many boolean operations

Fastest

Slowest
### SystemC Language Architecture

- **Layered Libraries**
  - Verification Library, etc.

- **Primitive Channels**
  - Signal, Mutex, Semaphore, FIFO, etc.

- **Core Language**
  - Modules
  - Ports
  - Processes
  - Interfaces
  - Channels
  - Events & Time
  - Event-driven simulation

- **Data Types**
  - 4-valued Logic type
  - 4-valued Logic Vectors
  - Bits and Bit Vectors
  - Arbitrary Precision Integers
  - Fixed-point types
  - C++ user-defined types

- **C++ Language Standard**

### Channel types

- **Primitive channels**
  - No visible structure
  - No processes
  - Cannot directly access other primitive channels
  - Types provided in 2.0 – See LRM for details
    - sc_signal
    - sc_signal_rv
    - sc_fifo
    - sc_mutex
    - sc_semaphore
    - sc_buffer

- **Hierarchical channels**
  - Are modules
    - May contain processes, other modules etc
  - May directly access other hierarchical channels
Topics

- System Design Context
  - General Methodology
  - Refinement
  - Benefits
- SystemC Overview
- Anatomy of an SC_MODULE
- SystemC Simulation Kernel
- An Example
- Some Homework

SystemC Simulation Kernel

```
sc_start()
```

```
sc_main()
```

```
notify(0) or wait(0)
```

```
notify() immediate
```

```
notify(t) wait(t)
```

```
Evalulate
```

```
Due to Update Events
```

```
Advance Time
```

```
Update
```

```
Runnable
```

```
Running
```

```
Waiting
```

```
Scheduled
```

```
P1
P2
P3
```

```
P4
P5
P6
```

```
done
```

© 2008 XtremeEDA USA Corporation - Version 080721.10
Topics

• System Design Context
  – General Methodology
  – Refinement
  – Benefits
• SystemC Overview
• Anatomy of an SC_MODULE
• SystemC Simulation Kernel
• An Example
• Some Homework

Example – Block Diagram
Example – SC_MODULE

```c++
#ifndef FIR_SYS_H
#define FIR_SYS_H

//BEGIN fir_sys.h
#include <systemc>
SC_MODULE(fir_sys) {
    //Port Declarations - NONE
    SC_CTOR(fir_sys);
    //Channel Declarations
    sc_fifo<double> orig_in_fifo;
    sc_fifo<double> data_in_fifo;
    sc_fifo<double> data_out_fifo;
    //Processes
    void stimulus_thread(void);
    void fir_thread(void);
    void results_method(void);
    //Helper Processes
    void read_cfg(void);
    //Event Declarations - NONE
    private:
        //Data Declarations
        const unsigned STIM_PTS;
        const unsigned IMP_PT;
        double* m_pipe;
        double* m_coeff;
        unsigned m_taps;
        unsigned m_tap;
        unsigned m_results_cnt;
    };//end fir_sys module
#endif
```

code from fir_sys.h

Example - Constructor

```c++
//Constructor
SC_HAS_PROCESS(fir_sys);
fir_sys::fir_sys(sc_module_name nm) : sc_module(nm)
,STIM_PTS(20), IMP_PT(10)
,m_taps(0), m_tap(0)
,m_results_cnt(0)
,orig_in_fifo(32)
,data_in_fifo(32)
,data_out_fifo(32)
{
    SC_THREAD(stimulus_thread);
    SC_THREAD(fir_thread);
    SC_METHOD(results_method);
    dont_initialize();
    sensitive <<
    .data_out_fifo.data_written_event();
}//end fir_sys
```

code from fir_sys.cpp
Example – stimulus_thread

```c
void fir_sys::stimulus_thread(void) {
    //stimulus_thread - Create impulse function
    //STIM_PTS - number of stimulus points
    //IMP_PT   - location of impulse function
    for (int t=0; t<STIM_PTS; t++) {
        double data = 0.0;
        if (t==IMP_PT) data = 1.0; //impulse
        orig_in_fifo.write(data);
        data_in_fifo.write(data);
    } //endfor
} //end fir_sys::stimulus_thread()
```

code from fir_sys.cpp

Example – fir_thread

```c
void fir_sys::fir_thread(void) {
    double data = 0;  //used to hold intermediate data point
    double result = 0; //contains next filtered data point
    unsigned coeff = 0; //used to index coefficients
    for(;;) { //loop forever
        coeff = m_tap; //used to index coefficients
        //read next piece of data
        data = data_in_fifo.read();
        m_pipe[m_tap++] = data;
        if (m_tap == m_taps) m_tap = 0; //wrap data buffer
        result = 0; //contains next filtered data point
        for (unsigned tap=0; tap!=m_taps; tap++, coeff++) {
            if (coeff == m_taps) coeff = 0; //wrap coeff.
            result += m_coeff[coeff] * m_pipe[tap];
        } //endfor
        data_out_fifo.write(result);
    } //endforever
} //end fir_sys::fir_thread()
```

code from fir_sys.cpp
Example – results_method

```c
void fir_sys::results_method(void) {
  //results_method - Print results with orig data. 
  // Method was used as a coding guideline illustration. 
  while(data_out_fifo.num_available() > 0) {
    m_results_cnt++;
    cout << "DATA: "
        << "[" << setw(2) << m_results_cnt << "]
        << "= " << setw(9) << fixed
        << setprecision(5) << orig_in_fifo.read()
        << " " << setw(9) << fixed
        << setprecision(5) << data_out_fifo.read() << endl;
  } //endwhile
  next_trigger();
} //end fir_sys::results_method()
```

code from fir_sys.cpp

Topics

- System Design Context
  - General Methodology
  - Refinement
  - Benefits
- SystemC Overview
- Anatomy of an SC_MODULE
- SystemC Simulation Kernel
- An Example
- Some Homework
Homework

• Provided separately