Restricted material

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
- System Verilog
- Vera
- PSL

* Modified from DVCon - Gabe Moretti EDN

---

### Modeling Characteristics and Models

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

- **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

© 2009 XtremeEDA USA Corporation - Version 090820.10

Restricted for use by registered University of Texas students only.
Model Refinement

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

TLM Based ESL Methodology

Restricted material
ESL Impacts on Schedule - before

Disparate Teams:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Architecture</th>
<th>Hardware</th>
<th>Verification</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware Development</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Development</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Integration</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ESL Impacts on Schedule - after

Disparate Teams:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Architecture</th>
<th>Model</th>
<th>Verification</th>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model Development</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware Development</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Development</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System Integration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Restricted material
“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?”

Behavioral Synthesis

Considered by many to be the missing Link for ESL Flows
• Several Vendors now offering solutions
  – Forte Design Systems
  – Mentor
  – Cadence
  – Agility
  – AutoESL
  – Synfora

• Takes “behavioral code” and “synthesizes” to RTL code
• Results comparable to human generated RTL
  – Less code ➔ faster design cycle
  – More microarchitectures considered
### Modeling Abstraction Levels

<table>
<thead>
<tr>
<th>Algorithmic level (AL)</th>
<th>Abstract Major events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmer’s view (PV)</td>
<td>TLM – minimal bus Instruction seq.</td>
</tr>
<tr>
<td></td>
<td>Programmed events</td>
</tr>
<tr>
<td></td>
<td>TLM – generic bus</td>
</tr>
<tr>
<td></td>
<td>Approximately Timed (AT) TLM – arch. bus Cycle-accurate I/F</td>
</tr>
<tr>
<td>Cycle Approximate (CA)</td>
<td>TLM – arch. bus Cycle-accurate</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

systemc.org

TLM-2.0 Reference Manual Released

OSCI has just announced the completion of a TLM-2.0 Reference Manual for public release. The manual is a formal description of the TLM-2.0 APIs and semantics, and is the result of intensive work by the TLM WG over the past year. It clarifies the original specification, incorporates minor changes for consistency and completeness, and is accompanied by an update of the TLM library “tcl Class Library”...
Coming in December

SystemC: From the Ground Up
Second Edition

David C. Black
Jack Donovan
Bill Binnon
Anna Keist

Springer

Restricted material

SystemC Books: Details

- Advanced Verification Techniques: A SystemC Based Approach for Successful Tapeout by Leena Singh, Leonard Drucker and Neyaz Khan ©2004
- ESL Design and Verification by Brian Bailey, Grant Martin and Andrew Piziali ©2007
- SystemC: From the Ground Up by David Black and Jack Donovan ©2004 (now in paperback!)
- System Design with SystemC by Thorsten Groetker, Stan Liao, Grant Martin and Stuart Swan ©2002
- SystemC: Methodologies and Applications by Wolfgang Muller, Wolfgang Rosenstiel and Jurgen Ruf
- SystemC Primer by Jayram Bhasker ©2004
- Transaction-Level Modeling with SystemC - TLM Concepts and Applications for Embedded Systems by Frank Ghenassia ©2005
**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
  - Channels, events
- Notion of Time
  - Clocks, sc_time
- Concurrency
  - Processes
- Hardware Data Types
  - 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 ➔ Modules
Interoperability ➔ TLM Standard
Test Bench ➔ Verification library
Running ➔ Scheduler

Using SystemC With OSCI PoC Simulator

Standard C++ development environment

header files
libraries

class library and simulation kernel

Compiler
Linker
Debugger

"make"

executable = simulator

source files for system and testbenches

ASIC
IP-Core
DSP

Restricted for use by registered University of Texas students only.
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>Verification Library, etc.</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>Channels</td>
<td>C++ user-defined types</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Events &amp; Time</th>
<th>Event-driven simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-driven simulation</td>
<td></td>
</tr>
</tbody>
</table>

| C++ Language Standard | |
|-----------------------||
A Simple Module – conceptual (not SystemC)

**Connectivity**

```
module camera(ccd_p, img_p);
image M1(…);
fifo CH1(…);
image M2(…);
endmodule
```

**Hierarchy**

```
camera
  image M1
  fifo CH1
  jpeg M2
```

**Verilog**

```
module image(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
```

SC_MODULE Anatomy - module

```
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

//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

Module (rectangle)
Port (square)
Export (diamond)
Process (oval)
Channel (hex)

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

SC_PORT

Interface (aka API)

modA mA
sc_port<
sc_fifo_out_if<int>>
A_thread
pA->write(v);

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

"points to the channel via the interface"

Restricted material
Port Declarations

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

```c
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);`

Ports added

```c
#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
};
```

Ports bound to interfaces
SC_EXPORT

```cpp
modA mA

sc_port<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>> pB;

B_thread

pB->read();
```

“exports the interface of the channel”

Direction of subroutine call reversed.

Restricted material

SC_MODULE Anatomy - channels

```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
  }
};
```

Restricted for use by registered University of Texas students only.
Channel Declarations

```cpp
channel_type channel_name;

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

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

Channels added

```cpp
// Filename: Camera.h
#include <systemc>
struct Camera : public sc_module {

  // Ports
  scport<ccd_p_if> ccd_p;
  scport<firewire_if> img_p;
  // Local channels & instances
  scfifo<image_t> CH1;
  // Local events
  // Processes
  // Constructor
  Camera(sc_module_name nm);

private:
  // Helper member functions
  // Local data
};
```

Restricted material
SC_MODULE Anatomy - variables

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

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

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

sc_event event_name, event_name,...;

- 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

SC_MODULE(fir_sys) {...
    //Event Declarations
    sc_event fir_doneEvt;
    ...
}; //end fir_sys
Event Notify

The sc_event class has the following methods:

```cpp
void notify();
void notify( const sc_time& );
void notify( double, sc_time_unit );
```

//within a simulation process
```cpp
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 process 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**

```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
      }
};
```

**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)
{
    // 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

```c
void my_method(void) {
    //Behavior
    int local_i;
    next_trigger(args...);
    return;
    //until re-invoked
}
```

**SC_MODULE Anatomy - subroutines**

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

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: "
        << "[" << setw(2) << m_results_cnt << "]"
        << "= " << setw(9) << fixed
        << setprecision(5) << orig_in_fifo.read()
        << " = " << setw(9) << fixed
        << setprecision(5) << data_out_fifo.read()
        << endl;
    }
    //next_trigger();
}
```

assumes static sensitivity - TBD
//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

```c
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**

```c
//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...
```
Constructor – Simulation Process Registration

```cpp
SC_CTOR(fir_sys)
  : sc_module(\_name)
  , ...  
  , orig_in_fifo(32)
  , data_in_fifo(32)
  , data_out_fifo(32)
{
  SC_THREAD(stimulus_thread);
  SC_THREAD(fir_thread);
  SC_METHOD(results_method);
  ...  
}
//end constructor fir_sys
```

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”
wait(args)
for use with SC_THREAD

```c
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

```c
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) {...
    }
};

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")
        {...
    }
    ...
};
Module Instantiation Example - 1

```c
SC_MODULE(ex3)
{
    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);
        ...
    }//end SC_CTOR
};//end ex3
```

Constructor implementation

```c
//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

```c
#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

```c
SC_MODULE(MODULE_NAME) {
    //port declarations
    ...
    ...
    //module instantiations
    SC_CTOR(MODULE_NAME)
    : ..init list...
    {
        //simulation directives
    }
};
```
### Alternate Syntax - 2

```c
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_instance("ex1_instance"),
        ex2_instance("ex2_instance")
    { // Named connection
        ex1_instance.m(a);
        ex1_instance.n(ch1);
        // Positional connection
        ex2_instance(ch1, b);//Bad
    }//end constructor
};//end ex3
```

```c
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_inst");
        ex2_ptr=new ex2("ex2_inst");
    { // Named connection
        ex1_ptr->m(a);
        ex1_ptr->n(ch1);
        // Positional connection
        (*ex2_ptr)(ch1, b); //Bad
    }//end constructor
};//end ex3
```
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 “01XZxz”)
    - Convert to appropriate type for computation
- Use `sc_bigint<W>` or `sc_biguint<W>`
  - More than 64 bits wide
- Use `sc_fixed<>`, `sc_fix()` or `sc_ufixed<>`, `sc_ufix()`
  - Fixed-point arithmetic
  - Convert to `sc_uint` if many boolean operations

SystemC Language Architecture
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()**
  - **Elaborate**
  - **Initialize**
  - **Update**
  - **Advance Time**
  - **Cleanup**

- **sc_main()**
  - **Evaluate**
  - **notify()**
  - **notify(t)**
  - **wait()**

- **Runnable Waiting Running**
  - P1
  - P2
  - P3
  - P4
  - P5
  - P6
  - done

### Topics

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

---

Restricted for use by registered University of Texas students only.

Copyright (C) 2009 by XtremeEDA Corporation. All rights reserved.
Example – Block Diagram

```
orig_in_fifo  data_out_fifo

stimulus_thread    fir_thread    results_method

data_in_fifo

fir_sys
```

Example – SC_MODULE

```
#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;
};
#endif
```

code from fir_sys.h

```
© 2009 XtremeEDA USA Corporation - Version 090820.10

Example – Block Diagram

```
orig_in_fifo  data_out_fifo

stimulus_thread    fir_thread    results_method

data_in_fifo

fir_sys
```

Example – SC_MODULE

```
#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;
};
#endif
```
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

Restricted material
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(;;) {
        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++) {
            // wrap coeff.
            result += m_coeff[coeff] * m_pipe[tap];
        }
        data_out_fifo.write(result);
    }
}
```

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;
    }
    next_trigger();
}
```

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

1. Get Hello running (next slide)
2. Create a for-loop in the process to output the "Hello" message 10 times in bursts with a random delay between messages evenly distributed from 50 to 90 ns
3. Create two sub-modules, Generate & Monitor connected by an sc_signal<string> channel. You will need an output port and an input port on each. Instantiate them inside Hello. Move the loop into the Generate module, but have it write to the output port. Have the Monitor display values that show up on the input port.
Hello

Hello.h

```c
#ifndef Hello_h
#define Hello_h
#include <systemc>
SC_MODULE(Hello) {
    SC_CTOR(Hello);
    void end_of_elaboration(void);
    void main_thread(void);
    ~Hello(void);
};
#endif
```

Hello.cpp

```c
#include "Hello.h"
#include <iostream>
using namespace std;
void Hello::Hello(sc_module_name nm) {
    cout << "Constructing " << name() << endl;
    SC_HAS_PROCESS(Hello);
    SC_THREAD(main_thread);
    void Hello::end_of_elaboration(void) {
        cout << "End of elaboration" << endl;
    }
    void Hello::main_thread(void) {
        cout << "Hello World!" << endl;
    }
    Hello::~Hello(void) {
        cout << "Destroy " << name() << endl;
    }
};
```