• Introduction to SystemC
  • Overview and background
  • Central concepts
  • The SystemC World
  • Use cases and benefits

• Core Concepts and Syntax
• Bus Modeling
• Odds and Ends
What is SystemC?

System-level modeling language
• Network of communicating processes (c.f. HDL)
• Supports heterogeneous models-of-computation
• Models hardware and software

C++ class library
• Open source proof-of-concept simulator
• Owned by Accellera Systems Initiative
Features of SystemC

- Modules (structure)
- Ports (structure)
- Processes (computation, concurrency)
- Channels (communication)
- Interfaces (communication refinement)
- Events (time, scheduling, synchronization)
- Data types (hardware, fixed point)
Modules and Channels

Module

Process

Channel

Module

Process

Functionality / Computation

Separated!

Communication / Protocol
Architecture of SystemC

- User Applications
- SystemC Verification Library SCV
- TLM Library
- Other Model Libraries
- Primitive Channels (signal, buffer, fifo, mutex, semaphore)
- Core Language (module, port, process, channel, interface, event)
- Data Types
- C++ Language
Accellera Systems Initiative

Formed from Open-SystemC Initiative (OSCI) and Accellera
Website  http://www.accellera.org/community/systemc

National Users Groups

- http://www-ti.informatik.uni-tuebingen.de/~systemc
- http://www.nascug.org
- http://www.iscug.in
- Others...

Language Working Group LWG

- IEEE 1666™ -2005
  - SystemC 2.2 released Mar 2006  (IEEE 1666 compliant)
- IEEE 1666™ -2011
  - SystemC 2.3.1 released Apr 2014  (IEEE 1666 compliant)
  - Includes TLM-2
Other Working Groups

Verification Working Group VWG
  • SystemC Verification (SCV) library released 2003

Synthesis Working Group SWG
  • Synthesisable subset of SystemC

Transaction-Level Modeling Working Group TLMWG
  • TLM-1.0 released May 2005
  • TLM-2.0 released June 2008
  • TLM-2.0 part of IEEE 1666-2011

Analog and Mixed Signal Working Group AMSWG

Control, Configuration & Inspection Working Group CCIWG
What can you do with SystemC?

Discrete Event Simulation (events, time)
- Register Transfer Level (delta delays, bus resolution)
- Transaction Level (communication using function calls)
- Kahn Process Networks (infinite fifos, reads block when empty)
- Dataflow (input-execution-output stages)
- CSP (rendezvous, blocking reads & writes)

Continuous Time or Frequency Domain (Analog) SystemC AMS

NOT gate level

NOT abstract RTOS modeling (process scheduling, priorities, pre-emption) (originally planned as version 3)
Typical Use Case: Virtual Platform

- Multiple software stacks
- Multiple buses and bridges
- Digital and analog hardware IP blocks

Software

- CPU
- ROM
- RAM
- DMA

中断（Interrupt）
- Timer
- I/O
- Bridge

DSP
- ROM
- RAM

中断（Interrupt）
- Timer
- A/D

I/O
- Memory interface
- RAM
- DMA
- Custom peripheral
- D/A

TLM-2.0
What is SystemC Used For?

- Virtual Platform
  - Architectural exploration, performance modeling
  - Software development
  - Reference model for functional verification

- Available before RTL - **early!**
- Simulates much faster than RTL - **fast!**

- High-level synthesis

- Used by
  - Architects, software, hardware, and verification engineers
SystemC is Glue!

- Transaction-level modeling is communication-centric
Why SystemC in Particular?

- Industry standard IEEE 1666™
- Open-source C++ simulator
  - Available across multiple platforms (Linux, Windows, Mac)
  - No vendor lock-in, no licensing issues
- Easy integration with the C/C++ world
- Easy to put SystemC wrappers around existing HDL or C models
- Wide availability of tools, models and know-how
- Mature tool vendor support for co-simulation and debug
• **Introduction to SystemC**

• Core Concepts and Syntax
  • Data
  • Modules and connectivity
  • Processes & Events
  • Channels and Interfaces
  • Ports

• **Bus Modeling**

• **Odds and Ends**
### SystemC Data Types

In namespace `sc_dt::`

<table>
<thead>
<tr>
<th>Template</th>
<th>Base class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sc_int&lt;W&gt;</code></td>
<td><code>sc_int_base</code></td>
<td>Signed integer, $W &lt; 65$</td>
</tr>
<tr>
<td><code>sc_uint&lt;W&gt;</code></td>
<td><code>sc_uint_base</code></td>
<td>Unsigned integer, $W &lt; 65$</td>
</tr>
<tr>
<td><code>sc_bigint&lt;W&gt;</code></td>
<td><code>sc_signed</code></td>
<td>Arbitrary precision signed integer</td>
</tr>
<tr>
<td><code>sc_biguint&lt;W&gt;</code></td>
<td><code>sc_unsigned</code></td>
<td>Arbitrary precision unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(intermediate results unbounded)</td>
</tr>
<tr>
<td><code>sc_logic</code></td>
<td></td>
<td>4-valued logic: '0' '1' 'X' 'Z'</td>
</tr>
<tr>
<td><code>sc_bv&lt;W&gt;</code></td>
<td><code>sc_bv_base</code></td>
<td>Bool vector</td>
</tr>
<tr>
<td><code>sc_lv&lt;W&gt;</code></td>
<td><code>sc_lv_base</code></td>
<td>Logic vector</td>
</tr>
<tr>
<td><code>sc_fixed&lt;&gt;</code></td>
<td><code>sc_fix</code></td>
<td>Signed fixed point number</td>
</tr>
<tr>
<td><code>sc_ufixed&lt;&gt;</code></td>
<td><code>sc_ufix</code></td>
<td>Unsigned fixed point number</td>
</tr>
</tbody>
</table>
Limited Precision Integer sc_int

```cpp
int    i;
sc_int<8>  j;

i = 0x123;
sc_assert( i == 0x123 );

j = 0x123;
sc_assert( j == 0x23 );

sc_assert( j[0] == 1 );
sc_assert( j.range(7,4) == 0x2 );
sc_assert( concat(j,j) == 0x2323 );
```

- Other useful operators: arithmetic, relational, bitwise, reduction, assignment

- `length()`, `to_int()`, `to_string()`, `implicit-conversion-to-64-bit-int`
Logic and Vector Types

sc_logic and sc_lv<W>

- Values SC_LOGIC_0, SC_LOGIC_1, SC_LOGIC_X, SC_LOGIC_Z
- Initial value is SC_LOGIC_X

No arithmetic operators

Can write values as chars and strings, i.e. '0' '1' 'X' 'Z'

```cpp
sc_logic R, S;
R = '1';
S = 'Z';
S = S & R;

sc_int<4> n = "0b1010";
bool boo = n[3];
sc_lv<4> lv = "01XZ";
sc_assert(lv[0] == 'Z');
n += lv.to_int();
cout << n.to_string(SC_HEX);
```
Fixed Point Types

**sc_fixed** <\(wl, \text{iwl}, \text{q\_mode}, \text{o\_mode}, \text{n\_bits}\)> a;
**sc_ufixed** <\(wl, \text{iwl}, \text{q\_mode}, \text{o\_mode}, \text{n\_bits}\)> b;
**sc\_fix** c(\(wl, \text{iwl}, \text{q\_mode}, \text{o\_mode}, \text{n\_bits}\));
**sc\_ufix** d(\(wl, \text{iwl}, \text{q\_mode}, \text{o\_mode}, \text{n\_bits}\));

**Word length** - number of stored bits - no limit
**Integer word length** - number of bits before binary point
**Quantization mode** - behavior when insufficient precision
**Overflow mode** - behavior when result too big
**Number of saturated bits** - used with wrap overflow modes

Compiler flag **-DSC_INCLUDE_FX**
# Data Summary

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>SystemC</th>
<th>SystemC</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsigned</strong></td>
<td>unsigned int</td>
<td>sc_bv, sc_lv</td>
<td>sc_uint</td>
<td>sc_biguint</td>
</tr>
<tr>
<td><strong>Signed</strong></td>
<td>int</td>
<td>sc_int</td>
<td>sc_int</td>
<td>sc_bigint</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>Host-dependent</td>
<td>Limited precision</td>
<td>Limited precision</td>
<td>Unlimited precision</td>
</tr>
<tr>
<td><strong>Operators</strong></td>
<td>C++ operators</td>
<td>No arithmetic operators</td>
<td>Full set of operators</td>
<td>Full set of operators</td>
</tr>
<tr>
<td><strong>Speed</strong></td>
<td>Fastest</td>
<td>Faster</td>
<td>Slower</td>
<td><strong>Slowest</strong></td>
</tr>
</tbody>
</table>
Modules

- **Module**
- **Channel**
- **Process**

- **Instances of other modules - hierarchy**

Copyright © 2014-2015 by Doulos Ltd
```c++
#include "systemc.h"

SC_MODULE(Mult) {
    sc_in<int> a;
    sc_in<int> b;
    sc_out<int> f;

    void action() { f = a * b; }

    SC_CTOR(Mult) {
        SC_METHOD(action);
        sensitive << a << b;
    }
};
```
SC_MODULE or sc_module?

- Equivalent:

```cpp
SC_MODULE(Name)
{
    ...
};
```

```cpp
struct Name: sc_module
{
    ...
};
```

```cpp
class Name: public sc_module
{
    public:
    ...
};
```
Separate Header File

// mult.h
#include "systemc.h"

SC_MODULE(Mult)
{
    sc_in<int> a;
    sc_in<int> b;
    sc_out<int> f;

    void action();

    SC_CTOR(Mult)
    {
        SC_METHOD(action);
        sensitive << a << b;
    }
};

// mult.cpp
#include "mult.h"

void Mult::action()
{
    f = a * b;
}

• Define constructor in .cpp?
  Yes - explained later
Implicit read/write and Delta delays

```cpp
sc_in<int> a, b;

SC_METHOD(action);
    sensitive << a << b;
```

- **Implicit method calls**
  - For convenience
    ```cpp
    void action() { f = a * b; }
    ```

- **Explicit method calls**
  - write() schedules event with delta delay
    ```cpp
    f->write( a->read() * b->read() );
    cout << f->read() << endl;
    ```
    - Old value!
The Test Bench

Module: Stim
Port: Mult
Channel: Mon

Multiplier formula: f = a * b

Signals:
- asig
- bsig
- testclk

Copyright © 2014-2015 by Doulos Ltd
#include "systemc.h"
#include "stim.h"
#include "mult.h"
#include "mon.h"

SC_MODULE(Top)
{
    sc_signal<int> asig, bsig, fsig;
    sc_clock testclk;

    Stim stim1;
    Mult uut;
    Mon mon1;

    ...
}
SC_MODULE(Top)
{
    sc_signal<int> asig, bsig, fsig;

    sc_clock testclk;

    Stim stim1;
    Mult uut;
    Mon mon1;

    SC_CTOR(Top):
    testclk("testclk", 10, SC_NS),
    stim1("stim1"),
    uut ("uut"),
    mon1 ("mon1")
    {
    ...
    }
}
Port Binding

SC_CTOR(Top)
: testclk("testclk", 10, SC_NS),
  stim1("stim1"),
  uut("uut"),
  mon1("mon1")
{
  stim1.a(asig);
  stim1.b(bsig);
  stim1.clk(testclk);
  
  uut.a(asig);
  uut.b(bsig);
  uut.f(fsig);

  mon1.a.bind(asig);
  mon1.b.bind(bsig);
  mon1.f.bind(fsig);
  mon1.clk.bind(testclk);
}
sc_main

- **sc_main** is the entry point to a SystemC application

```c
#include "systemc.h"
#include "top.h"

int sc_main(int argc, char* argv[]) {
    Top top("top");
    sc_start();
    return 0;
}
```

- **Called from main()**
- **Instantiate one top-level module**
- **End elaboration, run simulation**
An Alternative to sc_main

- A SystemC implementation is not obliged to support sc_main

```c
#include "top.h"

SC_MODULE_EXPORT(Top);

// NCSC_MODULE_EXPORT(Top);
```

Tool-specific macro

- QuestaSim
- Incisive
#include "systemc.h"

SC_MODULE(Mod) {
    sc_in<bool> clk;
    sc_out<int> out;

    ... cout << endl;
}

#include "systemc"

using namespace sc_core;
using namespace sc_dt;
using std::cout;
using std::endl;

SC_MODULE(Mod) {
    sc_in<bool> clk;
    sc_out<int> out;

    ... std::cout << std::endl;
}
Summary of Files

- stim.h
- mon.h
- mult.h
- top.h
- main.cpp
- stim.cpp
- mult.cpp
- mon.cpp

#include <systemc.h>
Compilation and Simulation

- SystemC class libraries
- Pre-compiled headers?
- Pre-compiled library

C++ Development Environment
- Text Editor
- Compiler
- Linker

Executable
- make
- Debugger

User's source files
- .h
- .cpp

prompt> make
prompt> run.x
prompt> ddd run.x
• Processes
  • Must be within a module (not in a function)
  • A module may contain many processes

• Three different kinds of process
  • Methods \text{SC\_METHOD}
  • Threads \text{SC\_THREAD}
  • Clocked threads \text{SC\_CTHREAD} (for synthesis)

• Processes can be \textit{static} or \textit{dynamic}
SC_METHOD Example

#include <systemc.h>

template<class T>
SC_MODULE(Register)
{
    sc_in<bool> clk, reset;
    sc_in<T>    d;
    sc_out<T>   q;

    void entry();

    SC_CTOR(Register)
    {
        SC_METHOD(entry);
        sensitive << reset;
        sensitive << clk.pos();
    }
};

template<class T>
void Register<T>:::entry()
{
    if (reset)
        q = 0; // promotion
    else if (clk.posedge())
        q = d;
}

• SC_METHODs execute in zero time
• SC_METHODs cannot be suspended
• SC_METHODs should not contain infinite loops
SC_THREAD Example

```cpp
#include "systemc.h"

SC_MODULE(Stim)
{
    sc_in<bool> Clk;
    sc_out<int> A;
    sc_out<int> B;

    void stimulus();

    SC_CTOR(Stim)
    {
        SC_THREAD(stimulus);
        sensitive << Clk.pos();
    }
};
```

```cpp
#include "stim.h"

void Stim::stimulus()
{
    wait();
    A = 100;
    B = 200;
    wait();
    A = -10;
    B = 23;
    wait();
    A = 25;
    B = -3;
    wait();
    sc_stop();
}
```

- More general and powerful than an `SC_METHOD`
- Simulation may be slightly slower than an `SC_METHOD`
- Called once only: hence often contains an infinite loop
#include "systemc.h"
class Counter: public sc_module
{
public:
    sc_in<bool> clock, reset;
    sc_out<int> q;

    Counter(sc_module_name _nm, int _mod) : sc_module(_nm), count(0), modulus(_mod)
    {
        SC_HAS_PROCESS(Counter);
        SC_METHOD(do_count);
        sensitive << clock.pos();
    }

private:
    void do_count();
    int count;
    int const modulus;
};
Dynamic Sensitivity

```c
SC_CTOR(Module)
{
    SC_THREAD(thread);
    sensitive << a << b;
}

void thread()
{
    for (;;)
    {
        wait();
        ...
        wait(10, SC_NS);
        ...
        wait(e);
        ...
    }
}
```

- **Static sensitivity list**
- Wait for event on `a` or `b`
- Wait for 10ns
- Ignore `a` or `b`
- Wait for event `e`
```cpp
SC_MODULE(Test) {
    int data;
    sc_event e;
    SC_CTOR(Test) {
        SC_THREAD(producer);
        SC_THREAD(consumer);
    }
    void producer() {
        wait(1, SC_NS);
        for (data = 0; data < 10; data++) {
            e.notify();
            wait(1, SC_NS);
        }
    }
    void consumer() {
        for (;;) {
            wait(e);
            cout << "Received " << data << endl;
        }
    }
};
```

**sc_event and Synchronization**

- **Shared variable**
- **Primitive synchronization object**
- **Schedule event immediately**
- **Resume when event occurs**
sc_time

- Simulation time is a 64-bit unsigned integer
- Time resolution is programmable - must be power of 10 x fs
- Resolution can be set once only, before use and before simulation
- Default time resolution is 1 ps

```c
enum sc_time_unit {SC_FS, SC_PS, SC_NS, SC_US, SC_MS, SC_SEC};

sc_time(double, sc_time_unit); Constructor

void sc_set_time_resolution(double, sc_time_unit);
sc_time sc_get_time_resolution();

const sc_time& sc_time_stamp(); Get current simulation time
```
**sc_clock**

- `sc_clock clk("clk", 10, SC_NS, 0.4, 12, SC_NS, true);`
- `sc_clock clk("clk", "clock_N", 1, SC_NS, 0.5, 0, SC_NS, true);`

Duty cycle: 1st edge rising

Defaults

Avoid clocks altogether for fast models!
```c
#define SC_INCLUDE_DYNAMIC_PROCESSES

void global_func()
{
    for (;;) {
        wait(10, SC_NS);
        ...
    }
}

void static_thread()
{
    wait(20, SC_NS);
    ...
    sc_spawn( &global_func );
    ...
}

SC_THREAD(static_thread);
```
Spawning Member Functions

```cpp
struct Mod: sc_module
{
    SC_CTOR(Mod)
    {
        SC_THREAD(T);
        sc_spawn( sc_bind( &Mod::proc, this) );
    }

    void T()
    {
        sc_spawn( sc_bind( &Mod::proc, this ) );
    }

    void proc ()
    {
        ...
    }
};
```

- Spawn a member function as a static process
- Spawn a member function as a dynamic process
Elaboration / Simulation Callbacks

Elaboration

- sc_start()
- before_end_of_elaboration()
- end_of_elaboration()

Simulation

- start_of_simulation()
- sc_stop()
- done
- end_of_simulation()

Every process without don't_initialize() runs once
Overriding the Callbacks

- Called for each
  - Module
  - Primitive channel
  - Port
  - Export

- Do nothing by default
- `before_end_of_elaboration()` may perform instantiation and port binding
- In PoC simulator, `end_of_simulation()` only called after `sc_stop()`
The Scheduler in Detail

Runnable processes \( \{R\} \)

Evaluation

Update requests \( \{U\} \)

Initialization

Update

Delta notifications \( \{D\} \)

Timed notifications \( \{T\} \)

Notify

Notify(0)

Notify(>0)

Sensitive

Advance time

All at current time from \( \{T\} \) into \( \{R\} \)

Done

\( \text{sc}_\text{start}() \)

\( \text{request}_\text{update}() \)

\( \text{sc}_\text{stop}() \)
Kinds of Channel

• Primitive channels
  • Implement one or more interfaces
  • Derived from sc_prim_channel
  • Have access to the update phase of the scheduler
  • Examples - sc_signal, sc_signal_resolved, sc_fifo

• Hierarchical channels
  • Implement one or more interfaces
  • Derived from sc_module
  • Can instantiate ports, processes and modules

• Minimal channels - implement one or more interfaces
## Built-in Primitive Channels

<table>
<thead>
<tr>
<th>Channel</th>
<th>Interfaces</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sc_signal&lt;T&gt;</code></td>
<td><code>sc_signal_in_if&lt;T&gt;</code></td>
<td><code>value_changed_event()</code></td>
</tr>
<tr>
<td></td>
<td><code>sc_signal_inout_if&lt;T&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>sc_buffer&lt;T&gt;</code></td>
<td>Same as <code>sc_signal</code></td>
<td>On every write()</td>
</tr>
<tr>
<td><code>sc_signal_resolved</code></td>
<td>Same as <code>sc_signal</code></td>
<td>Same as <code>sc_signal</code></td>
</tr>
<tr>
<td><code>sc_signal_rv&lt;W&gt;</code></td>
<td>Same as <code>sc_signal&lt;sc_logic&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>sc_clock</code></td>
<td>Same as <code>sc_signal&lt;bool&gt;</code></td>
<td><code>posedge &amp; negedge</code></td>
</tr>
<tr>
<td><code>sc_fifo&lt;T&gt;</code></td>
<td><code>sc_fifo_in_if&lt;T&gt;</code></td>
<td><code>data_written_event()</code></td>
</tr>
<tr>
<td></td>
<td><code>sc_fifo_out_if&lt;T&gt;</code></td>
<td><code>data_read_event()</code></td>
</tr>
<tr>
<td><code>sc_mutex</code></td>
<td><code>sc_mutex_if</code></td>
<td><code>n/a</code></td>
</tr>
<tr>
<td><code>sc_semaphore</code></td>
<td><code>sc_semaphore_if</code></td>
<td><code>n/a</code></td>
</tr>
<tr>
<td><code>sc_event_queue</code></td>
<td>n/a</td>
<td>Every notify() invocation</td>
</tr>
</tbody>
</table>
An interface declares a set of methods (pure virtual functions)
An interface is an abstract base class of the channel
A channel *implements* one or more interfaces (c.f. Java)
A module calls interface methods via a port
#include "systemc"

class queue_if : virtual public sc_core::sc_interface
{
public:
    virtual void write(char c) = 0;
    virtual char read() = 0;
};
#include "queue_if.h"

class Queue : public queue_if, public sc_core::sc_object
{
    public:
        Queue(char* nm, int _sz)
            : sc_core::sc_object(nm), sz(_sz)
        { data = new char[sz]; w = r = n = 0; }
    void write(char c);
    char read();

    private:
        char* data;
        int sz, w, r, n;
};

Implements interface methods
Understanding Ports

Module

Process

Channel

sc_port<i_f> p;

p->method();

struct i_f: virtual sc_interface
{
    virtual void method() = 0;
};

struct Chan: i_f,
    sc_module
{
    void method() {...}
};

Port

Required interface

Provided interface

Provided interface

Required interface

Port
class Producer : public sc_core::sc_module
{
public:
    sc_core::sc_port<queue_write_if> out;

    void do_writes();

    SC_CTOR(Producer)
    {
        SC_THREAD(do_writes);
    }
};
#include <systemc>
#include "producer.h"
using namespace sc_core;

void Producer::do_writes()
{
    std::string txt = "Hallo World.";
    for (int i = 0; i < txt.size(); i++)
    {
        wait(SC_ZERO_TIME);

        out->write(txt[i]);
    }
}
Why Ports?

• Ports allow modules to be independent of their environment

• Ports support elaboration-time checks (register_port, end_of_elaboration)

• Ports can have data members and member functions
• **Introduction to SystemC**
• **Core Concepts and Syntax**
• **Bus Modeling**
  • Master and slave interfaces
  • Blocking versus non-blocking
  • Multiports
• **Odds and Ends**
Example Bus Model

Multiple bus masters (modules), shared bus (channel), multiple slaves (channels)
Bus arbitration and memory mapping built into the bus

Bus master interface

Bus slave interface

Interfaces

- Required
- Provided
- Required
- Provided

Source0 Master

Proc1 Master

Ram0 Slave

Ram1 Slave
class master_if : virtual public sc_interface
{
public:
    virtual void write(sc_uint<8> address, sc_uint<12> data, int id) = 0;
    virtual void read (sc_uint<8> address, sc_uint<12> &data, int id) = 0;
};
Slave Interface Definition

class slave_if : virtual public sc_interface
{
public:
    virtual void slave_write(sc_uint<8> address, sc_uint<12> data) = 0;
    virtual void slave_read (sc_uint<8> address, sc_uint<12> &data) = 0;
    virtual void get_map(unsigned int &start, unsigned int &size) = 0;
};

Memory map managed within bus channel
void Bus::write(sc_uint<8> address, sc_uint<12> data, int id)
{
    request[id] = true;           // request access to the bus
    wait(proceed[id]);           // wait for permission
    request[id] = false;          // clear the flag
    slave_port[find_port(address)]->slave_write(address, data);
}
Blocking and Non-blocking Calls

An Interface Method Call runs in the context of the caller.

ASI terminology:

- A *blocking* method may call wait
- A *blocking* method must be called from a thread process
- A *non-blocking* method must not call wait
- A *non-blocking* method may be called from a thread or method process
- Naming convention `nb_*`

**Important!**
void Bus::control_bus()
{
    int highest;
    for (;;)
    {
        wait(clock->posedge_event());

        // Pick out a master that's made a request
        highest = -1;
        for (int i = 0; i < n_masters; i++)
            if (request[i])
                highest = i;

        // Notify the master with the highest id
        if (highest > -1)
            proceed[highest].notify();
    }
}
• Introduction to SystemC
• Modules, Processes and Ports
• Bus Modeling
• Odds and Ends
In either case the value is indeterminate ('0' or '1')

Indeed, writing a signal from > 1 process gives a run-time error
Bus Resolution

SC_CTOR(Test)
{
    SC_THREAD(p1);
    SC_THREAD(p2);
}

sc_signal_resolved S;

void p1()
{
    wait(10, SC_NS);
    S = sc_logic('0');
}
void p2()
{
    wait(20, SC_NS);
    S = SC_LOGIC_1;
    wait(10, SC_NS);
    S = SC_LOGIC_Z;
}

<table>
<thead>
<tr>
<th>Resolved</th>
<th>X</th>
<th>0</th>
<th>1</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>Z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>p1</th>
<th>p2</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ns</td>
<td>-</td>
<td>-</td>
<td>'X'</td>
</tr>
<tr>
<td>10 ns</td>
<td>'0'</td>
<td>-</td>
<td>'0'</td>
</tr>
<tr>
<td>20 ns</td>
<td>'0'</td>
<td>'1'</td>
<td>'X'</td>
</tr>
<tr>
<td>30 ns</td>
<td>'0'</td>
<td>'Z'</td>
<td>'0'</td>
</tr>
</tbody>
</table>
Multiple Bindings

Module → Channel
Module → Channel

Unsynchronised access to shared memory

sc_signal_resolved

Channel

Module

Channel

Channel

Array ports

sc_port<i_f,2> p;

p[0]->method();

p[1]->method();
Resolved Channels

class sc_signal_resolved
: public sc_signal<sc_logic>
{...

template <int W>
class sc_signal_rv
: public sc_signal<sc_lv<W>>
{...

Port:

sc_in<>
sc_inout<>
sc_out<>
sc_in_resolved
sc_inout_resolved
sc_out_resolved

Can only bind to resolved signals
**Event Methods**

```cpp
sc_in<bool> clock;

sensitive << clock.pos();

if (clock.posedge())...

wait(clock.posedge_event());
```

### Event Finder Needed - Port Not Yet Bound

```cpp
sc_event_finder& pos() const;

bool posedge() const;

const sc_event& posedge_event() const;
```

### Return Type and Method

<table>
<thead>
<tr>
<th>Return type</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc_event_finder&amp;</td>
<td>value_changed()</td>
</tr>
<tr>
<td></td>
<td>pos()</td>
</tr>
<tr>
<td></td>
<td>neg()</td>
</tr>
<tr>
<td>bool</td>
<td>event()</td>
</tr>
<tr>
<td></td>
<td>posedge()</td>
</tr>
<tr>
<td></td>
<td>negedge()</td>
</tr>
<tr>
<td>sc_event&amp;</td>
<td>value_changed_event()</td>
</tr>
<tr>
<td></td>
<td>posedge_event()</td>
</tr>
<tr>
<td></td>
<td>negedge_event()</td>
</tr>
</tbody>
</table>
Accessing the command line

• Not always able to access command-line from sc_main
• `sc_argc()` and `sc_argv()[]` provide universal access

```c
for (int i = 1; i < sc_argc(); ++i) {
    string arg(sc_argv()[i]);
    if (arg == "-flag") {
        flag = true;
    } else if (arg == "-num" && i+1 < sc_argc()){
        number = atoi(sc_argv()[++i]);
    }
}
```

• Alternatives include
  • read from file
  • `cstr = getenv("ENVIRONMENT_VARIABLE");` //may be nullptr
Debugging

• Stream I/O: operator<< overloaded for built-in types
  ```
  cout << "data=" << data << endl;
  ```

• Error reporting
  ```
  SC_REPORT_INFO("msg_type", "message");
  ```

• Writing trace files for waveform viewers
  ```
  sc_trace(trace_file, data, "data");
  ```

• Software debuggers and EDA tools
  ```
  ddd run.x
  ```
Error Reporting

SC_REPORT_INFO ("msg_type", "msg");
SC_REPORT_WARNING("msg_type", "msg");
SC_REPORT_ERROR ("msg_type", "msg");
SC_REPORT_FATAL ("msg_type", "msg");

Prints msg_type + msg
also file, line, process, time
also throws exception
calls abort()
Handling Reports

• Use the macros

    SC_REPORT_WARNING("", "Just a friendly warning");
    SC_REPORT_ERROR("Me", "Big trouble");
    SC_REPORT_FATAL("Me", "You're dead");

• Customize the report handler

    sc_report_handler::set_actions(SC_WARNING, SC_DO NOTHING);
    sc_report_handler::set_actions("Me", SC_DISPLAY);
    sc_report_handler::set_actions("Me", SC_WARNING, SC_LOG);
    sc_report_handler::set_log_file_name("log.txt");
    sc_report_handler::stop_after(SC_WARNING, 5);
#include <sstream>
...

ostringstream oss;
oss << "Big trouble, address = "
    << addr << " data = " << data << ends;
SC_REPORT_ERROR("/DOULOS/SUPERBUS", oss.str().c_str());

- e.g. report from Proof-of-Concept simulator

Info: (I804) /IEEE_Std_1666_DEPRECATED:
sc_start(double) deprecated, use sc_start(sc_time) or sc_start()

- Suppress all such reports

sc_report_handler::set_actions("/IEEE_Std_1666_DEPRECATED",
    SC_DO NOTHING);
Actions and Severity

**Actions**

<table>
<thead>
<tr>
<th>SC_UNSPECIFIED</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC_DO NOTHING</td>
</tr>
<tr>
<td>SC_THROW</td>
</tr>
<tr>
<td>SC_LOG</td>
</tr>
<tr>
<td>SC_DISPLAY</td>
</tr>
<tr>
<td>SC_CACHE_REPORT</td>
</tr>
<tr>
<td>SC_INTERRUPT</td>
</tr>
<tr>
<td>SC_STOP</td>
</tr>
<tr>
<td>SC_ABORT</td>
</tr>
</tbody>
</table>

**Behaviour of the default handler**

- **No action**
- **No action but inhibit lower priority actions**
- **Throw exception**
- **Write message to log file**
- **Write message to standard output**
- **Cache the report**
- **Call sc_interrupt_here()**
- **Call sc_stop()**
- **Call abort()**

**Default actions**

- **SC_INFO : SC_LOG | SC_DISPLAY**
- **SC_WARNING : SC_LOG | SC_DISPLAY**
- **SC_ERROR : SC_LOG | SC_CACHE_REPORT | SC_THROW**
- **SC_FATAL : SC_LOG | SC_DISPLAY | SC_CACHE_REPORT | SC_ABORT**

*Each action is a bit mask – can be OR'd*
Accellera Systems Initiative

Current Accellera Board Members
- AMD
- ARM
- Cadence Design Systems Inc
- **Freescale Semiconductor**
  - Intel Corporation
  - Mentor Graphics
- NXP Semiconductors
- Qualcomm Inc
- Renesas Mobile
- ST Microelectronics
- Synopsys
- Texas Instruments

Website http://www.accellera.org

European and North American SystemC Users Groups (and others)
- http://www-ti.informatik.uni-tuebingen.de/~systemc
- http://www.nascug.org
For More FREE Information

• IEEE 1666


• ASI SystemC 2.3.1

  www.accellera.org

• On-line tutorials

  www.doulos.com/knowhow/systemc
Hardware Design

- VHDL
- Verilog
- SystemVerilog

Embedded Systems and ARM

- C
- C++
- UML
- RTOS
- Linux
- Yocto
- ARM Cortex
- Python
- Android
- OpenCL
- OpenGL

ESL & Verification

- SystemC
- TLM-2.0
- SystemVerilog
- OVM/VMM/UVM
- Perl
- Tcl/Tk