SystemC Overview

Moushumi Ganguly
System Level Design, Synopsys Inc.
April 16, 2001
Agenda

• SystemC
  – Why
  – What
  – Features of SystemC

• SystemC Simulation
  – How
  – Co-Simulation

• Synopsys tools based on SystemC
  – System Studio
  – SystemC Compiler
  – Design Flows
Why system level design language?

• Systems combine HW and SW
  – SW complexity dominates
  – common SW/HW language helps co-design

• System level modeling is done in C/C++
  – simulates fast: needed to verify complex behaviors

• Language standard needed
  – to promote tool interoperability
  – to enable IP exchange & integration
Can C++ be used as it is?

- **Concurrency**: Hardware systems are inherently concurrent, i.e. they operate in parallel.
- **Notion of time**: Time sequenced operations.
- **Hardware style communication**: Signals, protocols, etc.
- **Reactivity**: Hardware is inherently reactive.
- **Hardware data types**: Bit type, multi-valued logic type, signed and unsigned integer types and fixed-point types.
...But it can be extended

- SystemC is a library of C++ classes
  - Processes (for concurrency)
  - Clocks (for time)
  - Hardware data types (bit vectors, 4-valued logic, fixed-point types, arbitrary precision integers)
  - Waiting and watching (for reactivity)
  - Modules, ports, signals (for hierarchy)

- A light-weight simulation kernel
What is SystemC?

... a language to describe HW/SW systems at multiple levels of abstraction

... a C++ library to standardize C-based system modeling

  clocks    interrupts    HW data types
  concurrency    hierarchy    ...

... a basic simulation library
Industry alignment

- Original Contributor: 3
- Charter members: 58
- Open SystemC Initiative Steering Group: 13
  - Synopsys, CoWare, Cadence, ARM, Motorola, NEC, Fujitsu, Ericsson, Infineon, Sony, Lucent, STM, TI

www.systemc.org
A year of Strong SystemC Adoption

SystemC Adoption
7,000+ individuals
500+ organizations

SystemC V1.0 Released

1999 2000
SystemC Roadmap

SystemC 1.0 (2000)  Cycle Accurate
SystemC 2.0 (2001)  Timed Functional
SystemC 3.0  (2002)  Untimed Functional

Design Exploration
Performance Analysis
HW/SW partitioning

Task Partitioning
(2002) SystemC 3.0
Target RTOS/Core

Abstr. RTOS

RTOS

RTL

BCA

RTL

UTF

TF

HW / SW Partition

Refine communication

Refine behavior

Software

Hardware

Assign ‘execution time’

SystemC Roadmap
SystemC 1.0 Features

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

- Concurrency
  - Processes

- Hardware Data Types
  - bit vectors, arbitrary precision signed and unsigned integers, fixed-point numbers

- Notion of Time
  - Clocks

- Reactive Behavior
  - Watching

- Communication
  - Signals, protocols
SystemC Classes - Modules and Ports

• **Modules** (*sc_module*)
  – Fundamental structural entity
  – Contain processes
  – Contain other modules (creating hierarchy)

• **Ports** (*sc_in<>*, *sc_out<>*, *sc_inout<>*)
  – Modules have ports
  – Ports have types
  – A process can be made sensitive to ports/signals
SystemC Classes - Processes

• Processes
  – Functionality is described in a process
  – Processes run concurrently
  – Code inside a process is sequential
  – Methods as well as threads supported as processes
    • SC_METHOD
    • SC_THREAD
    • SC_CTHREAD
Process Type : Method

• Sensitive to a set of signals/ports
• Invoked when any of these changes
• Cannot be suspended

// Header File
SC_MODULE(my_module){
  // ports
  sc_in<int> a;
  sc_in<bool> b;
  sc_out<int> x;
  sc_out<int> y;
  // Internal signals
  sc_signal<bool> c;
  sc_signal<int> d;
  // Method process
  void my_method_proc();
}

// Constructor
SC_CTOR(my_module) {
  // register method process
  SC_METHOD(my_method_proc);
  sensitive << a << c << d;
  sensitive_pos << b;
}

// Implementation File
// Process Body
void my_module :: my_method_proc() {
  x = a + d;
  y = b / c;
}
Process Type: Thread

- Similar to methods except may be suspended
- Sensitive to a set of signals/ports
- Is invoked when any of these changes

// Header File
SC_MODULE(my_module) {
    // ports
    sc_in<int> a;
    sc_in<bool> b;
    sc_out<int> x;
    sc_out<int> y;
    // Internal signals
    sc_signal<bool> c;
    sc_signal<int> d;
    // Thread process
    void my_thread_proc();
    // Constructor
    SC_CTOR(my_module) {
        // register thread process
        SC_THREAD(my_thread_proc);
        sensitive << a << c << d;
        sensitive_pos << b;
    }
};

// Implementation File
// Process Body
void my_module :: my_thread_proc() {
    while (true) {
        x = a + d;
        wait();
        y = a /d;
        wait();
    }
}
Process Type : Cthread

- Special type of thread process
- Can be sensitive to only one edge of one clock
- Invoked at the active edge of the clock

```c
// Header File
SC_MODULE(my_module){
    // ports
    sc_in_clk clock;
    sc_in<int> a;
    sc_in<bool> b;
    sc_out<int> x;
    sc_out<int> y;
    // Internal signals
    sc_signal<bool> c;
    sc_signal<int> d;
    // CTthread process
    void my_cthread_proc();
}

// Constructor
SC_CTOR(my_module) {
    // register thread process
    SC_CTHREAD(my_cthread_proc, clock.pos());
}

// Implementation File
// Process Body
void my_module :: my_cthread_proc() {
    while (true) {
        x = a + d;
        wait();
        y = a /d;
        wait();
    }
}
```
Signals and Clocks

• **Signals** (*sc_signal<>*)
  - Processes communicate with each other through signals
  - Ports of modules are connected using signals

• **Clocks** (*sc_clock, sc_signal<bool]*)
  - Timekeepers of the system - time advances from one clock edge to the next
  - Multiple clocks (with arbitrary phase relationship) allowed
Port read/write operations

• Read and write using read() and write() methods

SC_MODULE (module_name) {
    // ports
    sc_in<int> a;
    sc_out<int> b;
    int c;
    void entry() {
        b write(10); // write 10 to the port
        if (a read() < 5) { // Read the port
            c = a; // Read the port again
        }
    }
    // rest of module
}

SC_MODULE (module_name) {
    // ports
    sc_in<int> a;
    sc_out<int> b;
    int c;
    void entry() {
        b = 10; // write 10 to the port
        if (a < 5) { // Read the port
            c = a; // Read the port again
        }
    }
    // rest of module
}
Datatypes

• C++ built in data types may be used
  long, int, short, char, unsigned long, unsigned int,
  unsigned short, unsigned char, float, double, long
  double, and bool

• SystemC provides other types that are needed

  Scalar types: sc_bit, sc_logic
  Integer types: sc_int, sc_uint, sc_bigint, sc_biguint
  Bit and logic vector types: sc_bv, sc_lv
  Fixed point: sc_fixed, sc_ufixed, sc_fix, sc_ufix
Reactivity

- Waiting (wait(), wait_until(...))
- Watching (watching(...))
  - Watching for an event while doing something else (local or global)
# C++ features can be used

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>Derive new classes from a base class</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>Overloading functionality for different data types</td>
</tr>
<tr>
<td>Pointers</td>
<td>Dynamically accessing memory location</td>
</tr>
<tr>
<td>Multi-dimensional Arrays</td>
<td>Abstract multi-dimensional memories</td>
</tr>
<tr>
<td>Global Functions</td>
<td>Code reduction, clearer</td>
</tr>
<tr>
<td>Structures</td>
<td>Related variables can be grouped</td>
</tr>
<tr>
<td>Pre-processing</td>
<td>Powerful for defining: constants, macros (#define) conditional compilation (#ifdef)</td>
</tr>
<tr>
<td>Complex scoping rules</td>
<td>Powerful scoping like for (int i=0; i&lt;8; i++)</td>
</tr>
</tbody>
</table>
Adder with Register

#include "systemc.h"

SC_MODULE(adder_reg) {
    sc_in<sc_int<8> > a;   // input port
    sc_in<sc_int<8> > b;    // input port
    sc_out<sc_int<9> > c; // output port
    sc_in<bool> clock;       // clock

    // Internal signal
    sc_signal<sc_int<9> > temp;

    // Adder process
    void add() {
        temp = a + b;
    }

    // Register update process
    void reg() {
        c = temp;
    }

    // Constructor
    SC_CTOR(adder_reg) {
        SC_METHOD(add);
        sensitive << a << b;

        SC_METHOD(reg);
        sensitive_pos << clock;
    }
};

// Register update process
void reg()
{  c = temp;  }
SystemC 2.0 Features

• SystemC 1.0
  – Static sensitivity
    • Processes are made sensitive to a fixed set of signals during elaboration

• SystemC 2.0
  – Static sensitivity
  – Dynamic sensitivity
    • The sensitivity (activation condition) of a process can be altered during simulation (after elaboration)
    • Main features: events and extended wait() method
Waiting

wait(); // as in SystemC 1.0
wait(event); // wait for event
wait(e1 | e2 | e3); // wait for first event
wait(e1 & e2 & e3); // wait for all events
wait(200, SC_NS); // wait for 200ns

// wait with timeout
wait(200, SC_NS, e1 | e2);
wait(200, SC_NS, e1 & e2);
SystemC 2.0 Features

• SystemC 1.0
  – Fixed set of communication channels (sc_signal, ...) and ports (sc_in, sc_out, ...).

• SystemC 2.0
  – user-defined
    • interfaces
    • channels
    • ports
  – richer set of predefined channels (HW signals, FIFO, semaphore, mutex, ...)

  Define your own bus, message queue, ... etc.
Agenda

• SystemC
  – Why
  – What
  – Features of SystemC

• SystemC Simulation
  – How
  – Co-Simulation

• Synopsys tools based on SystemC
  – System Studio
  – SystemC Compiler
  – Design Flows
Simulating SystemC Descriptions

SystemC Class Library

Header Files

SystemC Description

Simulation Kernel

C++ Compiler (debugger)

Executable = Simulator
SystemC Simulation Semantics

• “Evaluate then update” algorithm
  – each process that has events at its inputs is executed … output signal values are not updated
  – signal values are updated after all processes are executed

• Order of process execution does not determine simulation result
  – no coding-order dependence

• Each evaluate-update cycle is a “Delta cycle”
Debugging with SystemC

Standard C/C++ debugging tools

Unix, Linux: gdb, ddd, xgdb, dbx
NT: Visual C++ development environment

Use C++ I/O statements in your code (streaming)
SystemC data types know how to display themselves using C++ I/O

Trace waveforms for post-simulation viewing

sc_create_vcd_trace_file("filename")
format can be VCD, WIF, or ISDB
Trace signals or variables with:
sc_trace(tracefile, signal_or_variable, "display name")
SystemC-HDL Co-Simulation

Import

Export

VCS, Scirocco, MTI, ...

Co-sim Interface

Verilog VHDL

Co-sim Interface
Testbench

// Header File
#include "systemc.h"

SC_MODULE(adder_reg_tb) {
    sc_in_clk clock;
    sc_out<sc_int<8> > a;
    sc_out<sc_int<8> > b;
    sc_in<sc_int<9> > c;
    sc_int<16> num;

    // Stimulus and Monitor processes
    void monitor();
    void stim();

    // Constructor
    SC_CTOR(adder_reg_tb) {
        SC_METHOD(monitor);
        sensitive << a << b;
        SC_THREAD(stim);
        sensitive_pos << clock;
    }
};

// Implementation File
#include "adder_reg_tb.h"

void add_reg_tb :: monitor () {
    cout << "Time is : " << sc_time_stamp() <<", c = " << c << " \n" ;
}

int clknum = 0 ;

void add_reg_tb :: stim {
    a.write(0);
    b.write(0);
    clknum ++;
    while(true) {
        a.write(num);
        num ++;
        clknum ++;
        wait();
    }
}
```c
#include "systemc.h"
#include "add_reg.h"
#include "add_reg_tb.h"
int sc_main(int ac, char *av[]) {
    sc_signal<sc_int<8> > a;
    sc_signal<sc_int<8> > b;
    sc_signal<sc_int<9> > c;
    sc_clock clock("Clock", 10, 0.5, 0);
    // Initialize inputs
    a = 0;
    b = 0;
    // Instantiate all modules
    add_reg add_reg_inst("add_reg_inst");
    add_reg_inst.a(a);
    add_reg_inst.b(b);
    add_reg_inst.clock(clock);
    add_reg_inst.c(c);
    add_reg_tb add_reg_tb_inst("add_reg_tb_inst");
    add_reg_tb_inst.a(a);
    add_reg_tb_inst.b(b);
    add_reg_tb_inst.clock(clock);
    add_reg_tb_inst.c(c);
    sc_trace_file *tf = sc_create_vcd_trace_file("sysc");
    sc_trace(tf, clock.signal(), "Clock");
    sc_trace(tf, a, "a");
    sc_trace(tf, b, "b");
    sc_trace(tf, c, "c");
    sc_start(300);
    return(0);
}
```
Agenda

• SystemC
  – Why
  – What
  – Features of SystemC

• SystemC Simulation
  – How
  – Co-Simulation

• Synopsys tools based on SystemC
  – System Studio
  – SystemC Compiler
  – Design Flows
SystemC based tools from Synopsys

CoCentric™ System Studio

C/SystemC

Performance Exploration

HW/SW co-design

Processor Model

SystemC Exec. Specification

SystemC Synthesizable Model

CoCentric™ SystemC Compiler

SW C-code

Chip verification

SW implementation
CoCentric™ SystemC Compiler

**Complete synthesis from SystemC to hardware**

- **Synthesis from SystemC**
  - refine & synthesize from C/C++ executable spec
  - path to FPGAs for System designers
  - Synopsys Integration

- **Complete**
  - Behavioral & RTL
  - SoCs, ASICs, FPGAs
SystemC ASIC Flow

- C/C++ Specification
- Synthesizable Behavioral/RTL SystemC
- Target Library
- Constraints
- Refinement

Functional Simulation

Architectural Simulation

SystemC Compiler

Design Compiler

Physical Compiler Backend Tools

System Verification

Co-Simulation Interface

C++ Debugger

C++ Compiler

RTL SystemC

RTL HDL

Gate level netlist

Netlist & constraints

VCS Scirocco MTI

Formality Prime Time
SystemC FPGA Flow

- C/C++ Specification
- Synthesizable Behavioral/RTL SystemC
- Target Library
- Constraints

Refinement

Functional Simulation

Architectural Simulation

SystemC Compiler

FPGA Compiler II

FPGA Backend Tools

Gate level netlist

VCS Scirocco MTI

Co-Simulation Interface

System Verification

C++ Debugger

C++ Compiler

RTL SystemC

RTL HDL
Conclusion

• SystemC
  – Why
  – What
  – A Standard HW/SW Co-Design
  – C++ for HW (1.0)
  – System modeling (2.0)

• SystemC Simulation
  – SystemC
  – Co-Simulation
  – Fast
  – HDL import & export

• Synopsys tools
  – System Studio
  – SystemC Compiler
  – Co-Design verification/Analysis
  – SystemC HW Synthesis
  – Synopsys Integration
Thank You!

Questions/Comments
moushumi@synopsys.com
650-584-4631