SystemC 1.3
Languages for Embedded Systems

Prof. Stephen A. Edwards

Columbia University

March 2009
Even Verilog or VHDL’s behavioral modeling is not high-level enough

People generally use C or C++
Standard Methodology for ICs

System-level designers write a C or C++ model
Written in a stylized, hardware-like form
Sometimes refined to be more hardware-like
C/C++ model simulated to verify functionality
Model given to Verilog/VHDL coders
Verilog or VHDL specification written
Models simulated together to test equivalence
Verilog/VHDL model synthesized
Designing Big Digital Systems

Every system company was doing this differently
Every system company used its own simulation library
“Throw the model over the wall” approach makes it easy to introduce errors

Problems:
System designers don’t know Verilog or VHDL
Verilog or VHDL coders don’t understand system design
C and C++ are being used as ad-hoc modeling languages

Why not formalize their use?

Why not interpret them as hardware specification languages just as Verilog and VHDL were?

SystemC developed at my former employer Synopsys to do just this
What Is SystemC?

A subset of C++ that models/specifies synchronous digital hardware

A collection of simulation libraries that can be used to run a SystemC program

A compiler that translates the “synthesis subset” of SystemC into a netlist
What Is SystemC?

Language definition is publicly available
Libraries are freely distributed
Compiler is an expensive commercial product
See www.systemc.org for more information
Quick Overview

A SystemC program consists of module definitions plus a top-level function that starts the simulation.

Modules contain processes (C++ methods) and instances of other modules.

Ports on modules define their interface.

Rich set of port data types (hardware modeling, etc.).

Signals in modules convey information between instances.

Clocks are special signals that run periodically and can trigger clocked processes.

Rich set of numeric types (fixed and arbitrary precision numbers).
Modules

Hierarchical entity
Similar to Verilog’s module
Actually a C++ class definition
Simulation involves

- Creating objects of this class
- They connect themselves together
- Processes in these objects (methods) are called by the scheduler to perform the simulation
SC_MODULE(mymod) {
    /* port definitions */
    /* signal definitions */
    /* clock definitions */

    /* storage and state variables */

    /* process definitions */

    SC_CTOR(mymod) {
        /* Instances of processes and modules */
    }
};
Ports

Define the interface to each module

Channels through which data is communicated

Port consists of a direction

- input \texttt{sc\_in}
- output \texttt{sc\_out}
- bidirectional \texttt{sc\_inout}

and any C++ or SystemC type

```cpp
SC_MODULE(mymod) {
    sc_in<bool> load, read;
    sc_inout<int> data;
    sc_out<bool> full;

    /* rest of the module */
};
```
Signals

Convey information between modules within a module

Directionless: module ports define direction of data transfer

Type may be any C++ or built-in type

```cpp
SC_MODULE(mymod) {
    /* ... */
    /* signal definitions */
    sc_signal<sc_uint<32>> s1, s2;
    sc_signal<bool> reset;

    /* ... */
    SC_CTOR(mymod) {
        /* Instances of modules that connect to the signals */
    }
};
```
Instances of Modules

Each instance is a pointer to an object in the module

```cpp
SC_MODULE(mod1) { ... };

SC_MODULE(mod2) { ... };

SC_MODULE(foo) {
    mod1* m1;
    mod2* m2;
    sc_signal<int> a, b, c;

    SC_CTOR(foo) {
        m1 = new mod1("i1");  (*m1)(a, b, c);
        m2 = new mod2("i2");  (*m2)(c, b);
    }
}
```

Connect instance’s ports to signals
Processes

Only thing in SystemC that actually does anything

Procedural code with the ability to suspend and resume

Methods of each module class

Like Verilog’s initial blocks

Three types:

- METHOD: Models combinational logic
- THREAD: Models testbenches
- CTHREAD: Models synchronous FSMs
METHOD Processes

Triggered in response to changes on inputs
Cannot store control state between invocations
Designed to model blocks of combinational logic
METHOD Processes

SC_MODULE(onemethod) {
  sc_in<bool> in;
  sc_out<bool> out;

  void inverter();

  SC_CTOR(onemethod) {
    SC_METHOD(inverter);
    sensitive(in);
  }
};

Process is simply a method of this class
Create an instance of this process
Trigger when in changes
METHOD Processes

Invoked once every time input “in” changes
Should not save state between invocations
Runs to completion: should not contain infinite loops
Not preempted

```c
void onemethod::inverter() {
    bool internal;
    internal = in;
    out = ~internal;
}
```

Read a value from a port
Write a value to an output
THREAD Processes

Triggered in response to changes on inputs
Can suspend itself and be reactivated
Method calls wait to relinquish control
Scheduler runs it again later
Designed to model just about anything
THREAD Processes

```cpp
SC_MODULE(onemethod) {
    sc_in<bool> in;
    sc_out<bool> out;

    void toggler();

    SC_CTOR(onemethod) {
        SC_THREAD(toggler);
        sensitive << in;
    }
};
```

- Process a method of the class
- Create an instance of the process
- Alternate sensitivity list notation
THREAD Processes

Reawakened whenever an input changes
State saved between invocations
Infinite loops should contain a `wait()`

```cpp
void onemethod::toggler() {
  bool last = false;
  for (;;) {
    last = in; out = last; wait();
    last = ~in; out = last; wait();
  }
}
```

Relinquish control until the next change of signal on this process's sensitivity list
CTHREAD Processes

Triggered in response to a single clock edge
Can suspend itself and be reactivated
Method calls wait to relinquish control
Scheduler runs it again later
Designed to model clocked digital hardware
CThread Processes

```cpp
SC_MODULE(onemethod) {
    sc_in_clk clock;
    sc_in<bool> trigger, in;
    sc_out<bool> out;

    void toggler();

    SC_CTOR(onemethod) {
        SC_CTHREAD(toggler, clock.pos());
    }
};
```

Instance of this process created and relevant clock edge assigned
CTHREAD Processes

Reawakened at the edge of the clock
State saved between invocations
Infinite loops should contain a wait()

```cpp
void onemethod::toggler() {
    bool last = false;
    for (;;) {
        wait_until(trigger.delayed() == true);
        last = in; out = last;
        wait();
        last = ~in; out = last;
        wait();
    }
}
```
A CTHREAD for Complex Multiply

```cpp
struct complex_mult : sc_module {
    sc_in<int> a, b, c, d;
    sc_out<int> x, y;
    sc_in_clk clock;

    void do_mult() {
        for (;;) {
            x = a * c - b * d;
            wait();
            y = a * d + b * c;
            wait();
        }
    }

    SC_CTOR(complex_mult) {
        SC_CTHREAD(do_mult, clock.pos());
    }
};
```
Watching

A CTHREAD process can be given reset-like behavior

```cpp
SC_MODULE(onemethod) {
    sc_in_clk clock;
    sc_in<bool> reset, in;

    void toggler();

    SC_CTOR(onemethod) {
        SC_CTHREAD(toggler, clock.pos());
        watching(reset.delayed() == true);
    }
};

Process will be restarted when reset is true
```
Local Watching

It’s hard, but the SystemC designers managed to put a more flexible version of abort in the language.

Ugly syntax because they had to live with C++

Only for SC_CTHREAD processes
Local Watching

```c
void mymodule::myprocess() {

W_BEGIN
    watching(reset.delayed() == true);
W_DO
    /* do something */
W_ESCAPE
    /* code to handle the reset */
W_END

}
```
SystemC Types

SystemC programs may use any C++ type along with any of the built-in ones for modeling systems

- c_bit, sc_logic
  Two- and four-valued single bit
- sc_int, sc_unint
  1 to 64-bit signed and unsigned integers
- sc_bigint, sc_biguint
  arbitrary (fixed) width signed and unsigned integers
- sc_bv, sc_lv
  arbitrary width two- and four-valued vectors
- sc_fixed, sc_ufixed
  signed and unsigned fixed point numbers
Numeric Types

- **Integers**
  - Precise
  - Manipulation is fast and cheap
  - Poor for modeling continuous real-world behavior

- **Floating-point numbers**
  - Less precise
  - Better approximation to real numbers
  - Good for modeling continuous behavior
  - Manipulation is slow and expensive

- **Fixed-point numbers**
  - Worst of both worlds
  - Used in many signal processing applications
Integers, Floating-point, Fixed-point

Integer

Fixed-point

Floating-point \times 2
Using Fixed-Point Numbers

High-level models usually use floating-point for convenience

Fixed-point usually used in hardware implementation because they are much cheaper

Problem: the behavior of the two are different

How do you make sure your algorithm still works after it has been converted from floating-point to fixed-point?

SystemC’s fixed-point number classes facilitate simulating algorithms with fixed-point numbers
SystemC’s Fixed-Point Types

```c
sc_fixed<8, 1, SC_RND, SC_SAT> fpn;
```

- 8 is the total number of bits in the type
- 1 is the number of bits to the left of the decimal point
- SC_RND defines rounding behavior
- SC_SAT defines saturation behavior
Rounding

What happens when your result doesn’t land exactly on a representable number?

Rounding mode makes the choice
Round up at 0.5
What you expect?
Round toward zero
Less error accumulation
Truncate
Easiest to understand
Overflow

What happens if the result is too positive or too negative to fit in the result?

Saturation? Wrap-around?

Different behavior appropriate for different applications
Saturate
Sometimes desired
SC_SAT_ZERO

Set to zero
Odd Behavior
SC_WRAP

Wraparound
Easiest to implement
SystemC Semantics

Cycle-based simulation semantics

Resembles Verilog, but does not allow the modeling of delays

Designed to simulate quickly and resemble most synchronous digital logic
Clocks

The only thing in SystemC that has a notion of real time

Only interesting part is relative sequencing among multiple clocks

Triggers SC_CTHREAD processes or others if they decided to become sensitive to clocks
Clocks

sc_clock clock1("myclock", 20, 0.5, 2, false);

20

2 0.5 of 20
SystemC 1.0 Scheduler

Assign clocks new values

Repeat until stable

- Update the outputs of triggered SC_CTHREAD processes
- Run all SC_METHOD and SC_THREAD processes whose inputs have changed

Execute all triggered SC_CTHREAD methods. Their outputs are saved until next time
Clock updates outputs of SC_CTHREADs

SC_METHODs and SC_THREADs respond to this change and settle down

Bodies of SC_CTHREADs compute the next state
Why Clock Outputs?

Why not allow Mealy-machine-like behavior in FSMs?

Difficult to build large, fast systems predictably

Easier when timing worries are per-FSM

Synthesis tool assumes all inputs arrive at the beginning of the clock period and do not have to be ready

Alternative would require knowledge of inter-FSM timing
Implementing SystemC

Main trick is implementing SC_THREAD and SC_CTHREAD’s ability to call wait()

Implementations use a lightweight threads package

```
/* ... */
wait();
/* ... */
```

Instructs thread package to save current processor state (register, stack, PC, etc.) so this method can be resumed later
Other trick is `wait_until()

```cpp
wait_until(continue.delayed() == true);
```

Expression builds an object that can check the condition.

Instead of context switching back to the process, scheduler calls this object and only runs the process if the condition holds.
Determinism in SystemC

Easy to write deterministic programs in SystemC

- Don’t share variables among processes
- Communicate through signals
- Don’t try to store state in SC_METHODs

Possible to introduce nondeterminism

- Share variables among SC_CTHREADs: They are executed in nondeterministic order
- Hide state in SC_METHODs: No control over how many times they are invoked
- Use nondeterministic features of C/C++
Synthesis Subset of SystemC

At least two

“Behavioral” Subset

- Implicit state machines permitted
- Resource sharing, binding, and allocation done automatically
- System determines how many adders you have

Register-transfer-level Subset

- More like Verilog
- You write a “+”, you get an adder
- State machines must be listed explicitly
Do People Use SystemC?

Not as many as use Verilog or VHDL

Growing in popularity

People recognize advantage of being able to share models

Most companies were doing something like it already

Use someone else’s free libraries? Why not?
Conclusions

C++ dialect for modeling digital systems
Provides a simple form of concurrency:
Cooperative multitasking
Modules
Instances of other modules
Processes
Conclusions

**SC_METHOD**

- Designed for modeling purely functional behavior
- Sensitive to changes on inputs
- Does not save state between invocations

**SC_THREAD**

- Designed to model anything
- Sensitive to changes
- May save variable, control state between invocations

**SC_CTHREAD**

- Models clocked digital logic
- Sensitive to clock edges
- May save variable, control state between invocations
Conclusions

Perhaps even more flawed than Verilog

Verilog was a hardware modeling language forced into specifying hardware

SystemC forces C++, a software specification language, into modeling and specifying hardware

SystemC 2.0 quite a change: moved to a more flexible, event-driven modeling style. Modeling, not synthesis the main focus.

Will it work? Time will tell.