Outline

- Introduction and Overview
- Language Features
- Simple Module Design
- Some System-Level Design

SystemC

- A C++ based class library and design environment for system-level design
- Suitable for functional description that might eventually be implemented as either HW or SW
- Open standard
  - Language definition is publicly available
  - Libraries are freely distributed
  - Synthesis tools are an expensive commercial product
- www.systemc.org
Language Architecture (v2.0)

Channels for MoCs
Kahn process networks, SDF, etc

Methodology-specific Channels
Master/Slave library

Elementary Channels
Signal, Timer, Mutex, Semaphore, FIFO, etc

Core Language
Module
Ports
Processes
Events
Interfaces
Channels
Event-driven simulation kernel

Data types
Bits and bit-vectors
Arbitrary precision integers
Fixed-point numbers
4-valued logic types, logic-vectors
C++ user defined types

C++ Language Standard
Glossary

- **Module**
  - Basic building block for structural partitioning
  - Contains ports, processes, data
  - Other modules

- **Process**
  - Basic specification mechanism for functional description
  - Three types
    - `sc_method`: sensitive to some ports/signals, no wait statements
    - `sc_thread`: sensitive to some ports/signals with wait statements
    - `sc_cthread`: sensitive to only clock
 Modules

- Hierarchical entity
- Similar to VHDL’s `entity`

- 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
Module Example

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 sc_in
  • output sc_out
  • bidirectional sc_inout
• Can be any C++ or SystemC type
Port Example

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
Signal Example

SC_MODULE(mymod) {
    /* port 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 of Processes

• 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);
    }
}
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

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

Read a value from the port
Write a value to an output port
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

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

    void toggler();

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

};

Process is simply a method of this class
Instance of this process created
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();
    }
}
```
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

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();
    }
}
```

Relinquish control until the next clock cycle in which the trigger input is 1

Relinquish control until the next clock cycle
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
- Limited version of Esterel’s abort

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 from the beginning when reset is true
SystemC Types

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

- **sc_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
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
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
• Delayed assignment and delta cycles
  • Just like VHDL and Verilog
  • Essential to properly model hardware signal assignments
    • Each assignment to a signal won’t be seen by other processes until the next delta cycle
    • Delta cycles don’t increase user-visible time
    • Multiple delta cycles may occur
Objectives of SystemC 2.0

• Primary goal: Enable System-Level Modeling
  • Systems include hardware, software, or both
  • Challenges:
    • Wide range of design models of computation
    • Wide range of design abstraction levels
    • Wide range of design methodologies
Objectives of SystemC 2.0 (cont’d)

- **Solution in SystemC 2.0**
  - Introduces a small but very general purpose modeling foundation => **Core Language**
  - **Elementary channels**
    - Other library models provided (FIFO, Timers, ...)
    - Even SystemC 1.0 **Signals**
  - Support for various models of computation, methodologies, etc.
    - Built on top of the core language, hence are separate from it
Communication and Synchronization

- SystemC 1.0 *Modules* and *Processes* are still useful in system design
- But communication and synchronization mechanisms in SystemC 1.0 (*Signals*) are restrictive for system-level modeling
  - Communication using queues
  - Synchronization (access to shared data) using mutexes
Communication and Synchronization

• SystemC 2.0 introduces general-purpose
  • Channel
    • A container for communication and synchronization
    • They implement one or more interfaces
  • Interface
    • Specify a set of access methods to the channel
      • But it does not implement those methods
  • Event
    • Flexible, low-level synchronization primitive
    • Used to construct other forms of synchronization
Communication and Synchronization

- Other comm. & sync. models can be built based on the above primitives
  - Examples
    - HW-signals, queues (FIFO, LIFO, message queues, etc) semaphores, memories and busses (both at RTL and transaction-level models)
Communication and Synchronization

Module1

Channel

Module2

Interfaces

Events

Ports to Interfaces
FIFO Modeling Example

Problem definition: FIFO communication channel with blocking read and write operations
class write_if : public sc_interface
{
    public:
    virtual void write(char) = 0;
    virtual void reset() = 0;
};

class read_if : public sc_interface
{
    public:
    virtual void read(char&) = 0;
    virtual int num_available() = 0;
};
FIFO Example (cont.)

class fifo: public sc_channel,
    public write_if,
    public read_if
{
    private:
        enum e {max_elements=10};
        char data[max_elements];
        int num_elements, first;
        sc_event write_event,
            read_event;
    bool fifo_empty() {...};
    bool fifo_full() {...};

    public:
        SC_CTOR(fifo) {
            num_elements = first=0;
        }

    void write(char c) {
        if ( fifo_full() )
            wait(read_event);
        data[<you say>] = c;
        ++num_elements;
        write_event.notify();
    }

    void read(char &c) {
        if( fifo_empty() )
            wait(write_event);
        c = data[first];
        --num_elements;
        first = <you say>;
        read_event.notify();
    }
void reset() {
    num_elements = first = 0;
}

int num_available() {
    return num_elements;
}
}; // end of class declarations
FIFO Example (cont.)

- All channels must
  - be derived from `sc_channel` class
    - SystemC internals (kernel/sc_module.h)
      

```cpp
typedef sc_module sc_channel;
```

  - be derived from one (or more) classes derived from `sc_interface`
    - provide implementations for all pure virtual functions defined in its parent `interfaces`
FIFO Example (cont.)

- Note the following extensions beyond SystemC 1.0
  - `wait()` call with arguments => dynamic sensitivity
    - `wait(sc_event)`
    - `wait(time) // e.g. wait(200, SC_NS);`
    - `wait(time_out, sc_event) // wait(2, SC_PS, e);`

- Events
  - are the fundamental synch. primitive in SystemC 2.0
  - Unlike signals,
    - have no type and no value
    - always cause sensitive processes to be resumed
    - can be specified to occur:
      - immediately/ one delta-step later/ some specific time later
The wait() Function

// wait for 200 ns.
sc_time t(200, SC_NS);
wait( t );

// wait on event e1, timeout after 200 ns.
wait( t, e1 );

// wait on events e1, e2, or e3, timeout after 200 ns.
wait( t, e1 | e2 | e3 );

// wait on events e1, e2, and e3, timeout after 200 ns.
wait( t, e1 & e2 & e3 );

// wait one delta cycle.
wait( SC_ZERO_TIME );
The notify() Method of sc_event

- Possible calls to notify():

  ```c
  sc_event my_event;

  my_event.notify(); // notify immediately

  my_event.notify( SC_ZERO_TIME ); // notify next delta cycle

  my_event.notify( 10, SC_NS ); // notify in 10 ns

  sc_time t( 10, SC_NS );
  my_event.notify( t ); // same
  ```
SC_MODULE(producer) {
    public:
        sc_port<write_if> out;

    SC_CTOR(producer) {
        SC_THREAD(main);
    }

    void main() {
        char c;
        while (true) {
            out->write(c);
            if(...)  
                out->reset();
        }
    }
};

SC_MODULE(consumer) {
    public:
        sc_port<read_if> in;

    SC_CTOR(consumer) {
        SC_THREAD(main);
    }

    void main() {
        char c;
        while (true) {
            in->read(c);
            cout<<
                in->num_available();
        }
    }
};
SC_MODULE(top) {
    public:
        fifo *afifo;
        producer *pproducer;
        consumer *pconsumer;

    SC_CTOR(top) {
        afifo = new fifo("Fifo");
        pproducer=new producer("Producer");
        pproducer->out(afifo);

        pconsumer=new consumer("Consumer");
        pconsumer->in(afifo);
    }
};
FIFO Example (cont.)

- **Note:**
  - **Producer module**
    - `sc_port<write_if> out;`
      - Producer can only call member functions of `write_if` interface
  - **Consumer module**
    - `sc_port<read_if> in;`
      - Consumer can only call member functions of `read_if` interface
      - e.g., Cannot call reset() method of `write_if`
  - **Producer and consumer** are unaware of how the channel works
  - just aware of their respective *interfaces*
  - Channel implementation is hidden from communicating modules
Future Evolution of SystemC

• Expected to be SystemC 3.0
  • Support for RTOS modeling
  • New features in the core language
    • Fork and join threads + dynamic thread creation
    • Interrupt or abort a thread and its children
    • Specification and checking of timing constraints
    • Abstract RTOS modeling and scheduler modeling

• Expected to be SystemC 4.0
  • New features in the core language
    • Support for analog mixed signal modeling
Future Evolution of SystemC (cont.)

- Extensions as libraries on top of the core language
  - Standardized channels for various MOC (e.g. static dataflow and Kahn process networks)
- Testbench development
  - Libraries to facilitate development of testbenches
    - Data structures that aid stimulus generation and response checking
    - Functions that help generate randomized stimulus, etc.
- System level modeling guidelines
  - Library code that helps users create models following the guidelines
- Interfacing to other simulators
  - Standard APIs for interfacing SystemC with other simulators, emulators, etc.