

# **System-on-Chip Design Introduction**

Hao Zheng

Computer Science & Engineering

U of South Florida

# Standard Methodology for IC Design

- 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 Large Digital Systems

- Systems become more complex, pushing us to to design and verify at higher level of abstraction
  - Enable early exploration of system level tradeoffs
  - Enable early verification of entire system
  - Enable verification at higher speed
- SW is playing an increasing role in system design
- Problems:
  - System designers don't know Verilog or VHDL
  - Verilog or VHDL coders don't understand system design

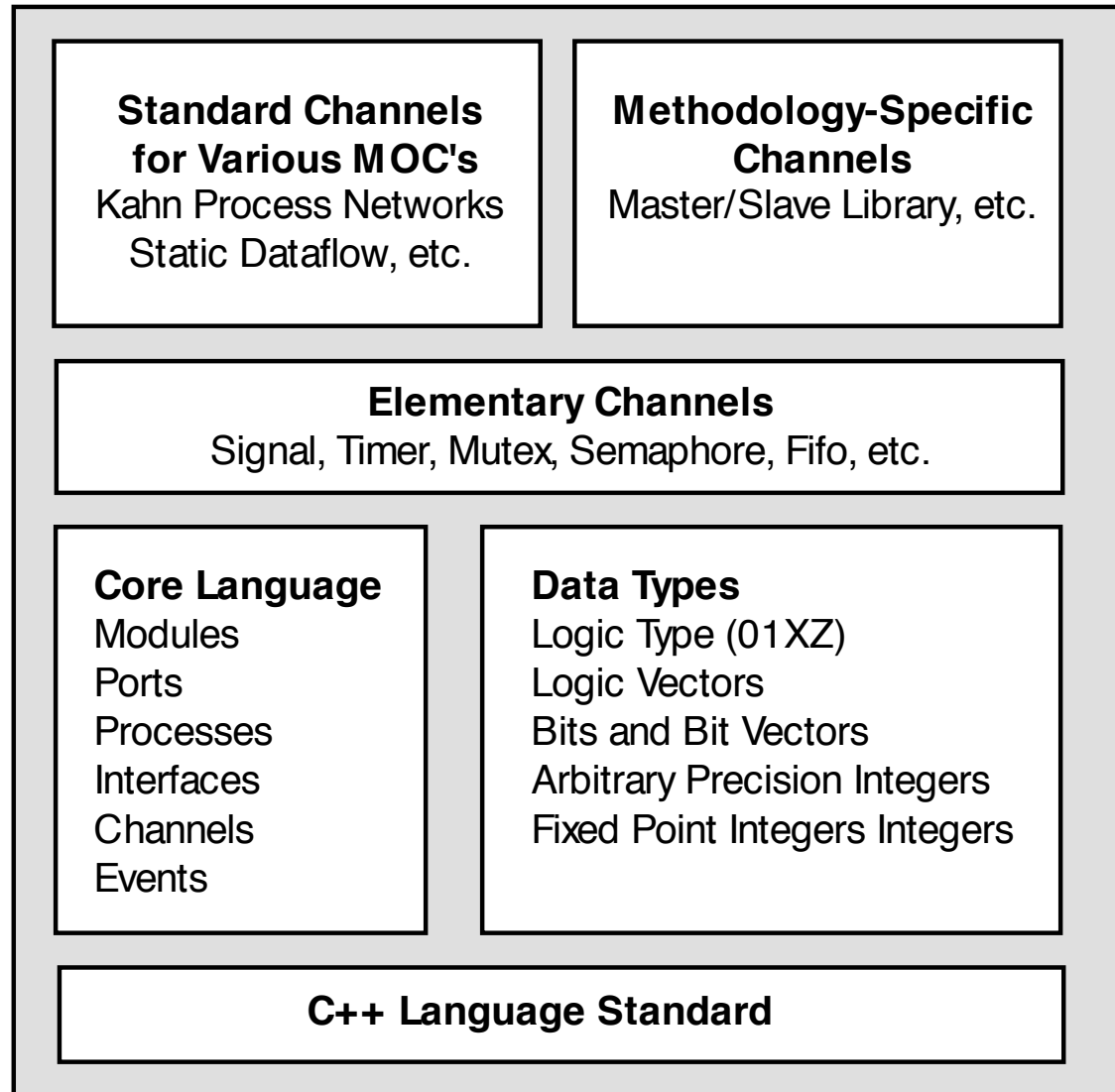
# What Is SystemC?

- A subset of C++ capable of system-level or HW modeling
  - Easy integration of SW/HW in a single model
- A collection of libraries that can be used to simulate SystemC programs
  - Libraries are freely distributed
- Commercial compilers that translates the “synthesis subset” of SystemC into a netlist
- Language definition is publicly available

# SystemC Language Architecture

Upper layers are built on lower layers

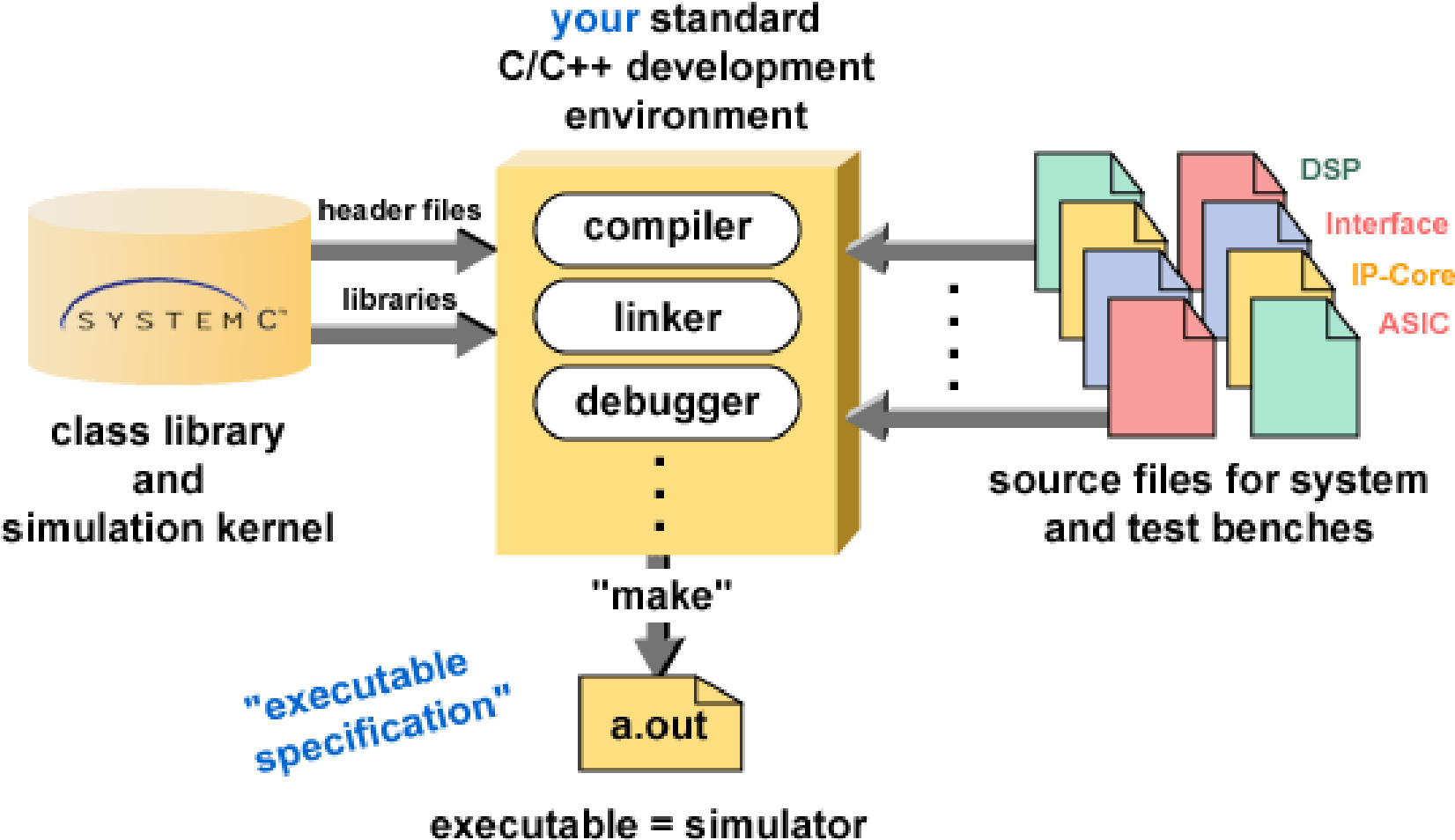
Lower layers can be used without upper layers



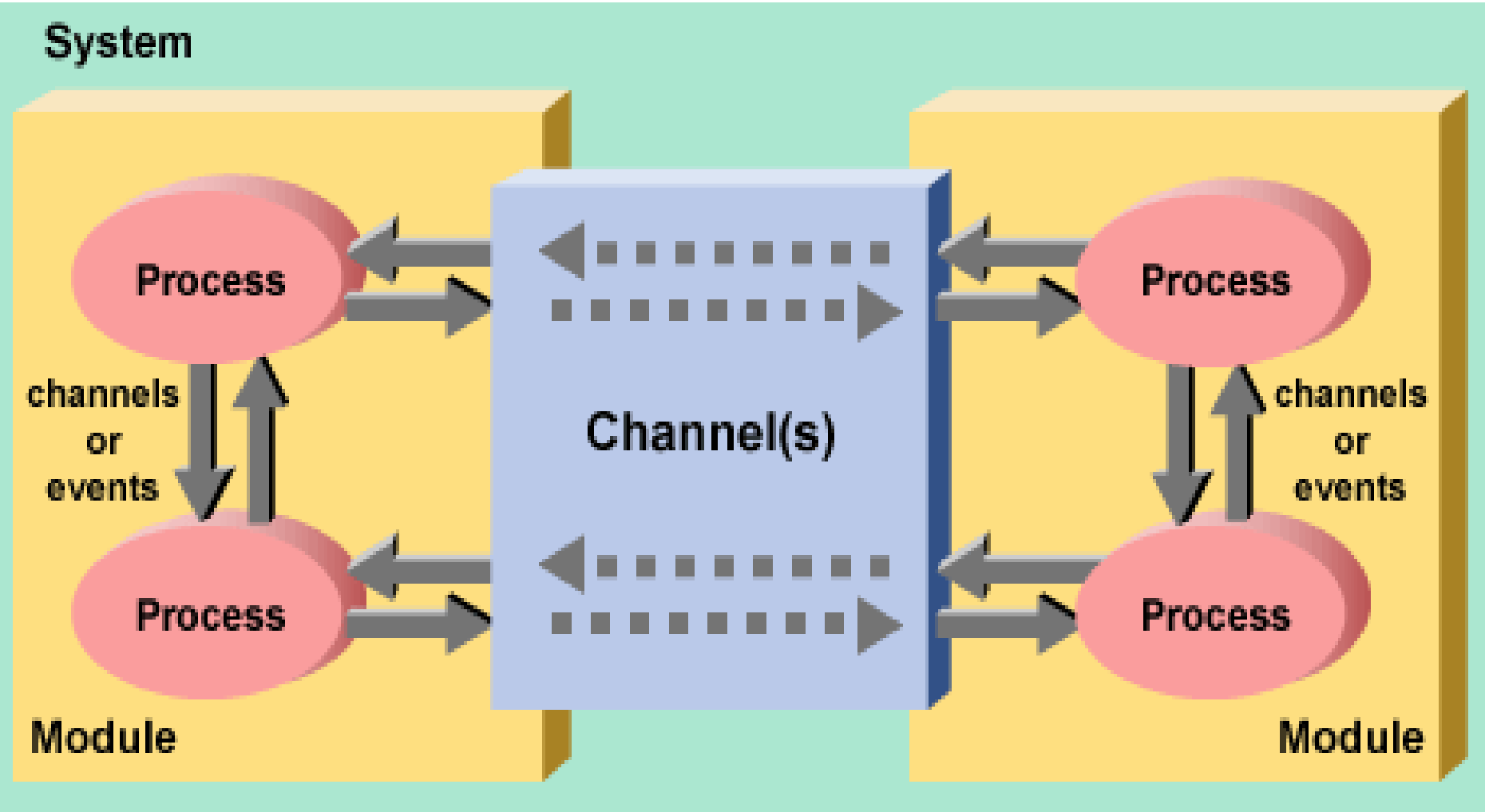
# Benefits

- SystemC provides a single language
  - To describe HW & SW at various abstraction levels
  - To facilitate seamless HW & SW co-simulation
  - To facilitate step-by-step refinement of a system design from high-level down to RTL for synthesis.
- A SystemC model is an executable specification.
  - Offers fast simulation speed for design space exploration

# SystemC Environment



# SystemC Model – Overview





# SystemC Model Overview

- A SystemC model 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.)
- **Channels & interfaces** provide high-level communication models.
- **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)

# Model of Time

- Time units: `sc_time_unit`

`SC_FS`          femtosecond

`SC_PS`          picosecond

`SC_NS`          nanosecond

`SC_US`          microsecond

`SC_MS`          millisecond

`SC_SEC`        second

- Construction of time objects

```
sc_time t1(42, SC_PS)
```

# Modules

- Hierarchical entity
- Similar to entity of VHDL
  
- Actually a C++ class definition
  
- Simulation involves
  - Creating objects of this class
  - Connecting ports of module objects together
  - Processes in these objects (methods) are called by the scheduler to perform the simulation

# Modules

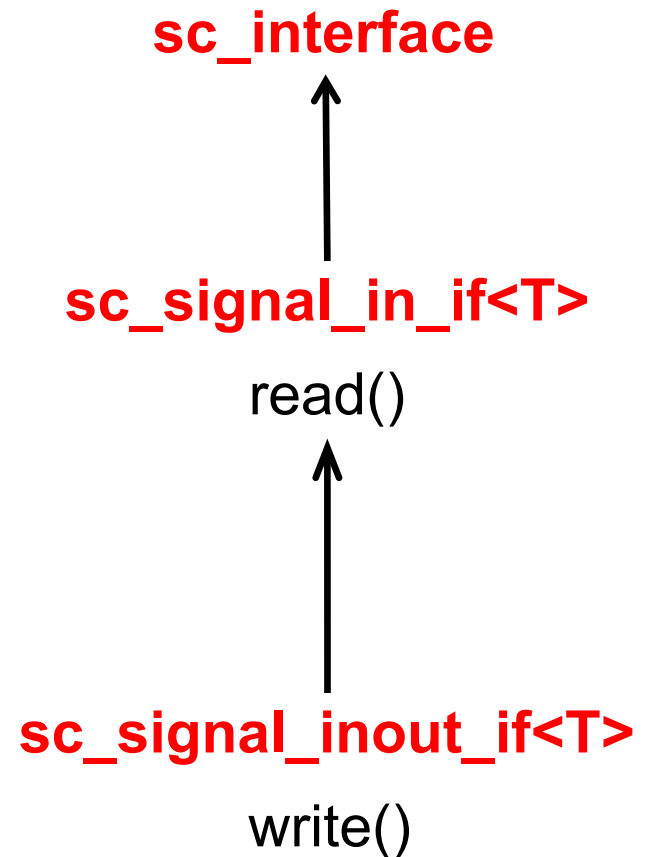
```
SC_MODULE(mymod) {  
    /* port definitions */  
    /* signal definitions */  
    /* clock definitions */  
  
    /* storage and state variables */  
  
    /* process definitions */  
  
    SC_CTOR(mymod) {  
        /* Instances of processes and  
other          modules */  
    }  
};
```

# Ports

- Define the inputs/outputs of each module
- Channels through which data is communicated
- Port consists of a direction
  - input **sc\_in**
  - output **sc\_out**
  - bidirectional **sc\_inout**
- And any C++ or SystemC type
- More general port types: **sc\_port***</interface\*>*

# Interfaces

- Connect ports to channels.
- Each defines a set of operations through which a port can operate a channel.
- Implemented by channels.
- A port accesses a channel through the supported interfaces.



# Ports

```
SC_MODULE(mymod) {  
    sc_in<bool> load, read;  
    sc_inout<int> data;  
    sc_out<bool> full;  
  
    /* rest of the module */  
};
```

# Ports

```
/* port sc_in declaration */
```

```
template<class T>
```

```
class sc_in : public
```

```
sc_port<sc_signal_in_if<T> > ...;
```

```
/* port sc_inout declaration */
```

```
template<class T>
```

```
class sc_inout : public
```

```
sc_port<sc_signal_inout_if<T> > ...;
```



# Signals

- Convey information between processes within a module
- Directionless
  - module ports define direction of data transfer
- Type may be any C++ or built-in type
- A special type of **channel**.

# Signals

```
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 */  
    }  
};
```

# Channels

- Models communications
- In SystemC, a channel is a module with local storage and a set of allowed operations grouped in interfaces.
- Modules are connected by connecting channels to their ports.
- Primitive channels: mutexes, FIFOs, signals
- Hierarchical channels can model more sophisticated communication structures, ie buses.

# Instances of Modules

`/* Each instance is a pointer to an object in the module */`

```
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



# Port Binding

## Positional Port Binding

```
a_module.(p1, p2, ... );
```

*px* is an instance of a port or a channel.

## Named Port Binding

```
a_port.(port or channel instance);
```

```
a_port.bind(port or channel instance);
```

# Named Port Binding – Example

```
SC_MODULE(M) {  
    sc_inout<int> P, Q, R, S;  
    sc_inout<int> *T;  
    SC_CTOR(M) {  
        T = new sc_input<int>;  
        ...  
    };  
};
```

```
SC_MODULE(Top) {  
    sc_inout<int> A, B;  
    sc_signal<int> C, D;  
    M m;  
    SC_CTOR(Top) : m("m") {  
        m.P(A);  
        m.Q.bind(B);  
        m.R(C);  
        m.S.bind(D);  
        m.T->bind(E);  
    };  
};
```

# Processes

- Define functionalities of modules.
- Simulate concurrent behavior.
- Procedural code with the ability to suspend and resume
- Similar to VHDL processes

# Three Types of Processes

- METHOD
  - Models combinational logic
- THREAD
  - Models event-triggered sequential processes
- CTHREAD */\* going away \*/*
  - Models synchronous FSMs
  - A special case of THREAD



# METHOD Processes

- Triggered in response to changes on inputs
- Cannot store control state between invocations
- Designed to model blocks of combinational logic
- Sequential logic can be modeled with additional state variables declared in the modules where the processes are created.

# METHOD Processes

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

```
    void inverter() { out = ~in; }
```

Process is simply a  
method of this class



```
SC_CTOR(onemethod) {
```

```
    SC_METHOD(inverter);
```

Instance of this  
process created



```
    sensitive(in);
```

and made sensitive  
to an input



```
};
```

# 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

```
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.
  - More general than METHOD processes.

# 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()

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

Relinquish control  
until the next  
change of a signal  
on the sensitivity  
list for this process



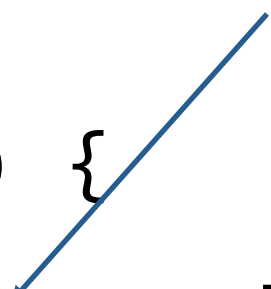
# CTHREAD Processes

- 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



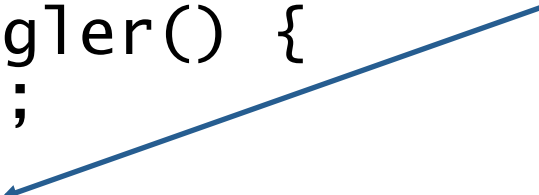


# CTHREAD Processes

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

```
void onemethod::togglер() {
    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 edge in which the trigger input is 1



Relinquish control until the next clock edge



# A CTHREAD for Complex Multiply

```
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());
    }
};
```

# Events

- Events, sensitivity and notification are essential for simulating concurrency in SystemC.
- An event is an object of class `sc_event`.

```
sc_event e;
```

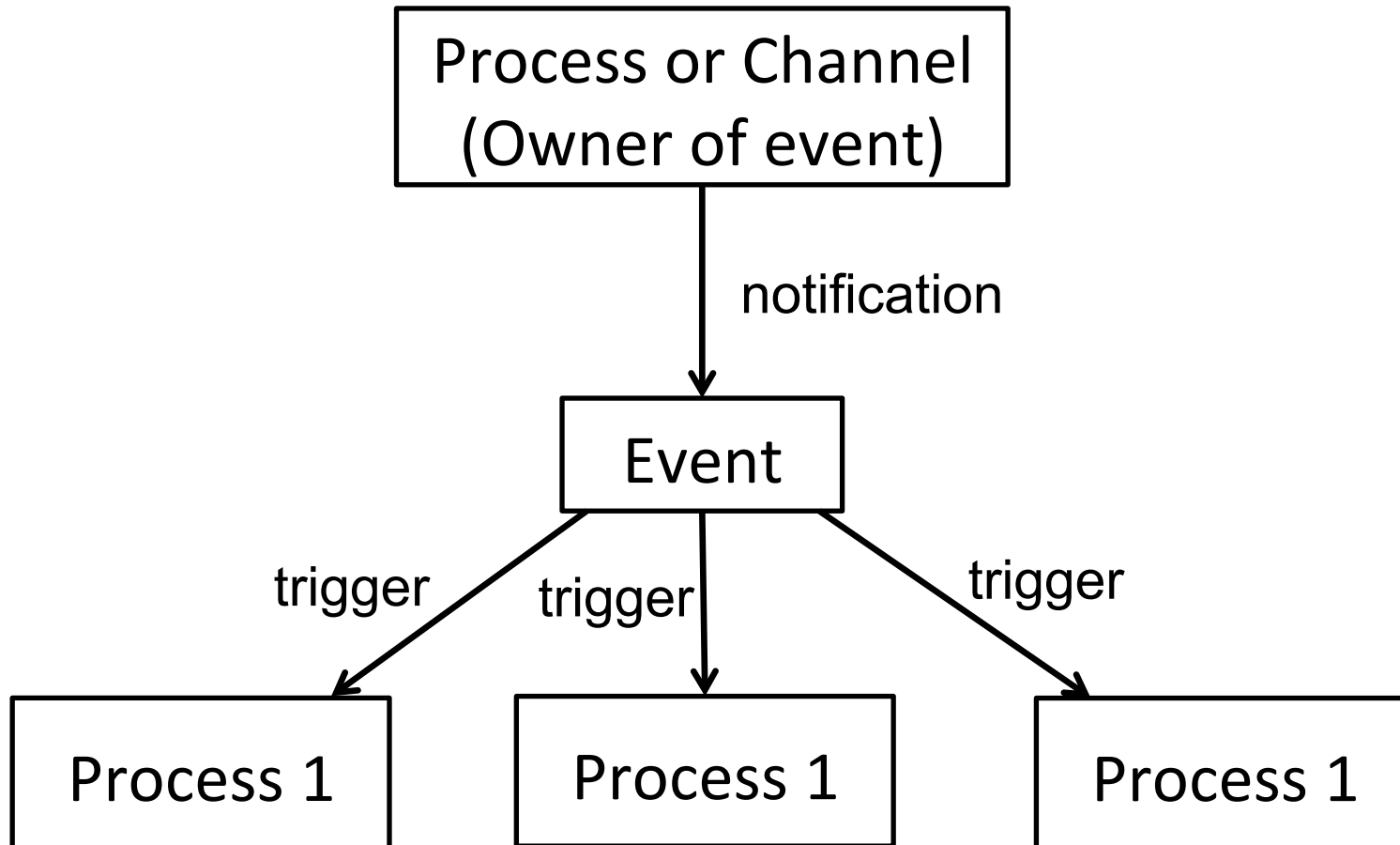
- An event is generated by its owner.

```
e.notify();
```

```
e.notify(SC_ZERO_TIME);
```

```
e.notify(100, SC_NS);
```

# Event Notification and Process Triggering



# Event Notification and Process Triggering

```
sc_signal<bool> s;  
s.initialize(false);  
sc_event e;
```

```
process1 {  
    s.write(true);  
    e.notify(sc_ZERO_TIME);  
}
```

```
process2 {  
    wait(e);  
    bool v = s.read();  
}
```

v gets the new value of s, 'true'.

# Event Notification and Process Triggering

```
sc_signal<bool> s;  
s.initialize(false);  
sc_event e;
```

```
process1 {                               process2 {  
    s.write(true);                       wait(e);  
    e.notify();                           bool v = s.read();  
}                                           }
```

v gets the old value of s, 'false'.

# Sensitivity

- After an event is generated, all processes sensitive on it are triggered.
- Static sensitivity

```
sensitivity << a << b;
```

- Dynamic sensitivity: use `wait(e)` in processes.

```
wait(e1 & e2 & e3);
```

```
wait(e1 | e2 | e3);
```

```
wait(200, SC_NS);
```

# More on Interfaces

- Ports and Interfaces allow the separation of computation from communication.
- All SystemC interfaces derived from `sc_interface`
  - Declared as abstract classes
- Consists of a set of operations
- Specifies only the signature of an operation
  - name, parameter, return value
  - Operations are defined in channels



# Interfaces: Example

```
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;
};
```

# More on Channels

- Models communications
- In SystemC, a channel is a module with local storage and a set of allowed operations grouped in interfaces.
- Modules are connected by connecting channels to their ports.
- Primitive channels: mutexes, FIFOs, signals
- Hierarchical channels can model more sophisticated communication structures, ie buses.

# Channels: Example

```
class fifo : public sc_channel,
            public write_if,
            public read_if
{
private:
    // just a constant in class scope
    const int max = 10;
    char data[max];
    int num_elements, first;
    sc_event write_event, read_event;

public:
    // ** definition of interfaces
};
```

# Channels: Example

```
class fifo :    public sc_channel,  
                public write_if,  
                public read_if  
{  
private:  
    // local data members.  
  
public:  
    SC_CTOR(fifo)() {  
        num_elements = first = 0;  
    }  
    // more on next slide  
};
```

# Channels: Example

```
class fifo :    public sc_channel,
                public write_if,
                public read_if
{
private:
    // local data members.

public:
    void write(char c) {
        if (num_elements == max)
            wait(read_event);
        data[(first+num_elements)%max] = c;
        ++num_elements;
        write_event.notify();
    };
};
```

# Channels: Example

```
class fifo : public sc_channel,
            public write_if,
            public read_if
{
private:
    // local data members.

public:
    void read(char& c) {
        if (num_elements == 0)
            wait(write_event);
        c = data[first];
        --num_elements;
        read_event.notify();
    };
};
```

# Channels: Example

```
class fifo : public sc_channel,  
            public write_if,  
            public read_if  
{  
private:  
    // local data members.  
  
public:  
    void reset() {  
        num_elements = 0;  
    }  
  
    int num_available() {  
        return num_elements;  
    }  
};
```

# A Complete Model

```
// the producer module
class producer : public sc_module
{
public:
    sc_port<write_if> out; // producer output port

    SC_CTOR(producer)      // module constructor
    {
        SC_THREAD(main);  // start the process
    }
    void main()            // the producer process
    {
        char c;
        while (true) {
            ...
            out->write(c); // write c into fifo
            if (...)
                out->reset(); // reset the fifo
        }
    }
};
```



# A Complete Model

```
// the consumer module
class consumer : public sc_module
{
public:
    sc_port<read_if> in; // consumer input port

    SC_CTOR(consumer)    // module constructor
    {
        SC_THREAD(main); // start the process
    }

    void main()          // the consumer process
    {
        char c;
        while (true) {
            in->read(c); // read c from the fifo
            if (in->num_available() > 5)
                ...;    // perhaps speed up processing
        }
    }
};
```

# A Complete Model

```
// the top module
class top : public sc_module
{
public:
    fifo *fifo_inst;          // a fifo instance
    producer *prod_inst; // a producer instance
    consumer *cons_inst; // a consumer instance

    SC_CTOR(top)              // the module constructor
    {
        fifo_inst = new fifo (Fifo1");
        prod_inst = new producer("Producer1");
        // bind the fifo to the producer's port
        prod_inst->out(fifo_inst);

        cons_inst = new consumer("Consumer1");
        // bind the fifo to the consumer's port
        cons_inst->in(fifo_inst);
    }
};
```

# A Complete Model: Top Level

```
int sc_main(int argc, char *argv[])
{
    top("model");

    // some environment definition
    // ...
    sc_start(1000, SC_SEC);
}
```

# Simulation Constructs

```
// start and run simulation forever.  
sc_start();
```

```
// start and run simulation for 1000 seconds  
sc_start(1000, SC_SEC);
```

```
// start and run simulation for 1000 seconds  
// an alternative approach  
sc_time sim_run(1000, SC_SEC);  
sc_start(sim_run)
```

# Simulation Constructs

```
// Returns current simulated time since  
// sc_start() is called.
```

```
sc_time sc_time_stamp();
```

```
// example
```

```
cout << "The current simulation time is"  
      << sc_time_stamp() << endl;
```

```
// Useful to find out performance of a  
// component during simulation
```

```
Sc_time op_start = sc_time_stamp();
```

```
    /* perform some operations */
```

```
sc_time delay = sc_time_stamp() - op_start;
```

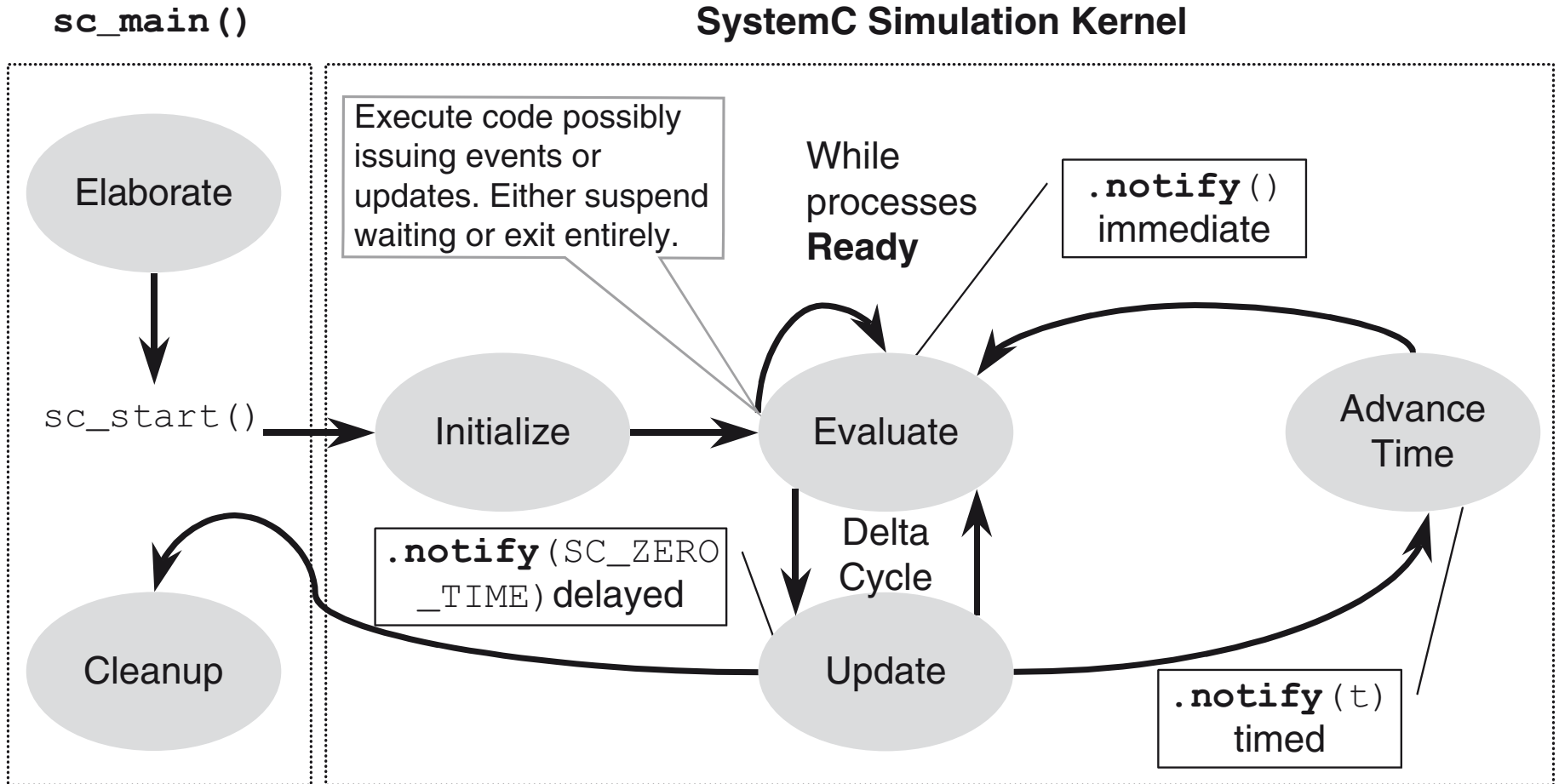
# Simulation Semantics

- A SystemC model is a set of communicating processes.
- In each simulation step, a process is in one of two states:
  - Active – executed until the next `wait()`.
  - Suspended – by calling `wait()`.
- For all active processes, which one to executed next is unknown

# Simulation Semantics

- SystemC is a sequential language.
- Scheduler mimics concurrency.
  - Sequentializes concurrency processes.
- Simulation iterates over the following steps
  - **Evaluate**: active processes are executed one at a time
  - **Update**: change signals, event notification with zero time delay
  - **Advance simulation time**.
- Simulation stops when there are not events scheduled.

# Simulation Semantics



*A delta cycle does not advance simulation time.*



# More on ZERO TIME and Immediate Event Notifications

```
SC_CTOR(example) {  
    SC_THREAD(A);  
    SC_THREAD(B);  
}
```

```
void A() {  
    e.notify();  
    cout << "A sent event e" << endl;  
}
```

```
void B() {  
    wait(e);  
    cout << "B got event e" << endl;  
}
```

# Reading Guide

- *SystemC-1* book: chapter 1 & 2
  - skip section 2.8
- *SystemC-2* book: chapter 1 & 2
- Refer to Chapter 3 of *SystemC-2* book for SystemC data types.