Chapter 5

SystemC Linguistics

Zainalabedin Navabi

SystemC Linguistics

+ SystemC Execution
+ Basic Syntax & Semantics
+ Language Semantics
+ Interfaces

  Examples

  Summary

SystemC Linguistics

- SystemC Execution
  - + Elaboration
  - + Simulation
  - + Running Elaboration & Simulation
  - Simulation Control
- Basic Syntax & Semantics
  - Modules
  - Ports
  - Signals
  - Internal Data Storage
  - + Instances
  - + Port Binding
  - + Constructors
  - + Processes
- Language Semantics
  - Resolved Logic Vector
  - + Types & Operations
  - + Time, Events & Sensitivity Control
- Interfaces
  - Interfaces
  - + Channels
  - Ports
- Examples
- Summary

SystemC Linguistics

- SystemC Execution
  - Elaboration
    - Creating a module hierarchy
    - Bounding ports & exports to channels
    - Setting time resolution
  - Simulation
    - Behavior of the Scheduler
    - Timing of the Scheduler
    - Examples
  - Running Elaboration & Simulation
    - Under Application Control
    - Under Kernel Control

  Simulation Control

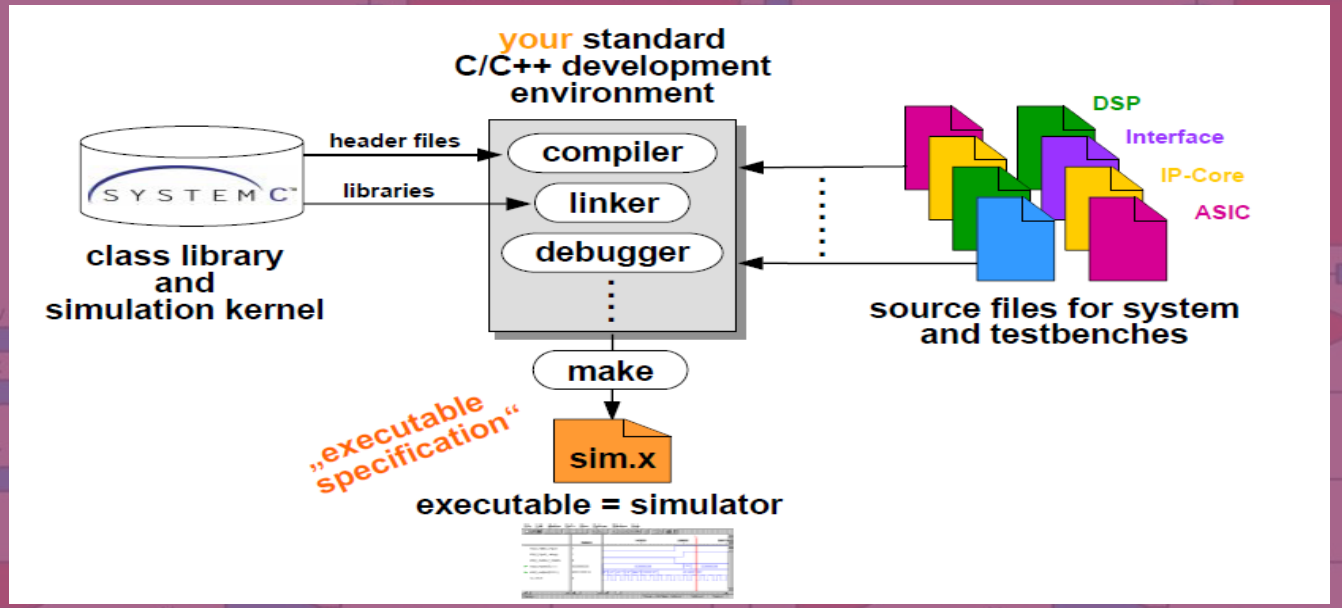
+ Basic Syntax & Semantics
+ Language Semantics
+ Interfaces

Examples

Summary

SystemC Environment

- SystemC implementation
  - A public shell consisting of predefined classes, functions, macros, and so forth that can be used directly by an application
  - A private kernel that implements the core functionality of the class library



SystemC Library Hierarchy

Sc_object

sc_module

sc_process_b

sc_method_process

sc_thread_process

sc_prim_channel

sc_port_base

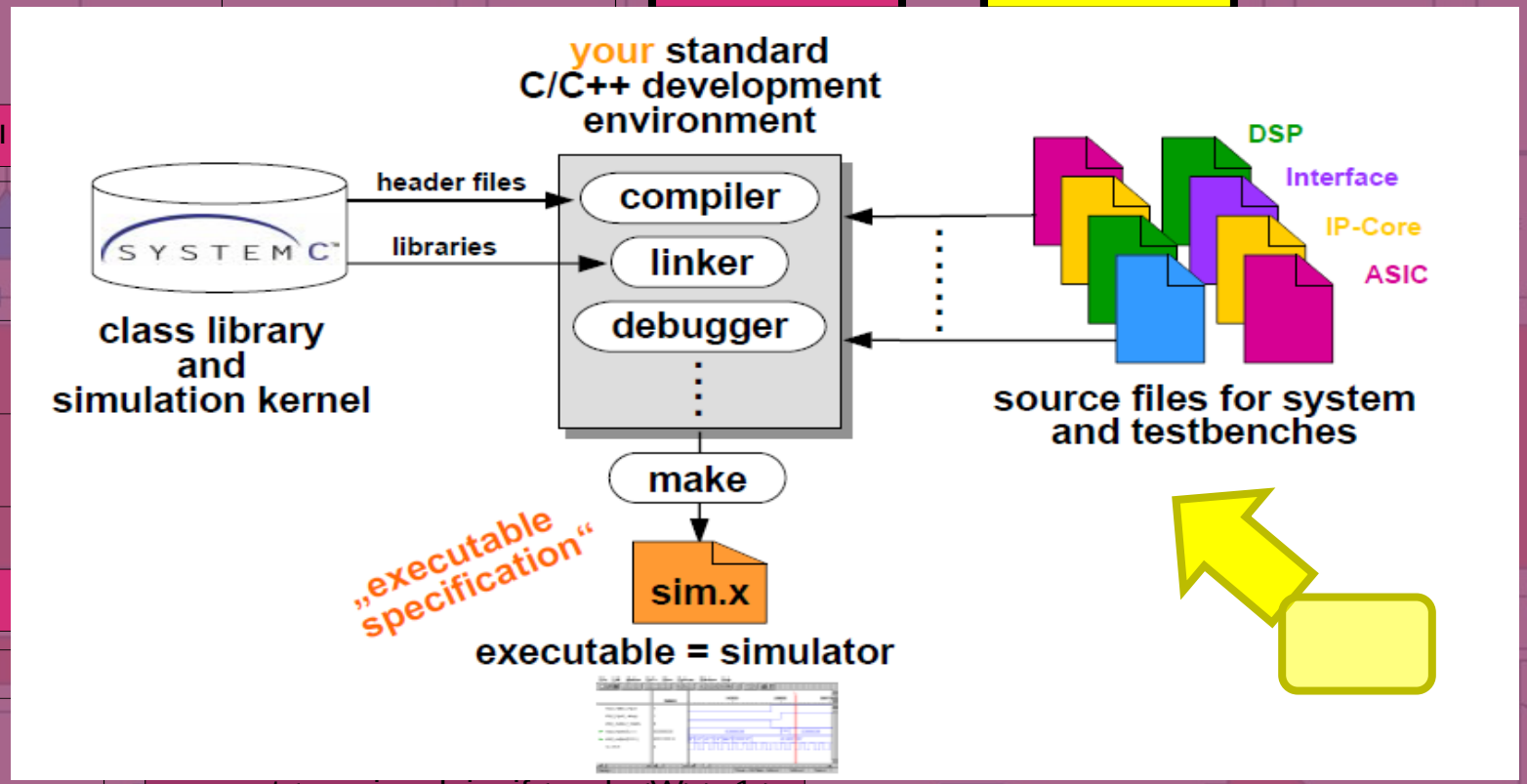
sc_interface

read_if

write_if

sc_mutex_if

sc_mutex



sc_port<sc_signal_in_if<sc_lv<W>>,1 >

sc_in<sc_lv<W> >

sc_in_rv<W>

sc_port_b<sc_fifo_in_if<T> >

sc_port<sc_fifo_in_if<T>,0 >

sc_fifo_in<T>

Elaboration

- Creating a module hierarchy
  - Instances of the classes
    - sc_module
    - sc_port
    - sc_export
    - sc_prim_channel

sc_object

May be created within a module or a sc_main

Can only be created within a module

  - An process instance created by invoking one of the process macros
    - sc_method
    - sc_thread
    - sc_cthread

Process Macros:
1. register the associated function with the kernel
2. use spawned processes for this same purpose
3. provide backward compatibility with earlier versions of SystemC
4. provide clocked threads for hardware synthesis

Elaboration

Binding ports & exports to channels

- Port instances can be bound to
  - channel instances
  - other port instances
  - export instances
- Ports can be bound by
  - name is performed by sc_port
  - position is performed by sc_module

- Export instances can be bound to
  - channel instances
  - other export instances
- Exports can be bound by
  - name is performed by sc_export

A given port instance shall not be bound both by name and by position

Elaboration

- Setting time resolution
  - The simulation time resolution can be set only during elaboration
  - Time resolution can only be set globally. There is no concept of a local time resolution
  - Time shall be represented internally as an integer multiple of the time resolution
  - Every object of class sc_time shall share a single common global time resolution

SystemC Linguistics

- SystemC Execution
  - Elaboration
    - Creating a module hierarchy
    - Bounding ports & exports to channels
    - Setting time resolution
  - Simulation
    - Behavior of the Scheduler
    - Timing of the Scheduler
    - Examples
  - Running Elaboration & Simulation
    - Under Application Control
    - Under Kernel Control

  Simulation Control

+ Basic Syntax & Semantics
+ Language Semantics
+ Interfaces

Examples

Summary

Simulation

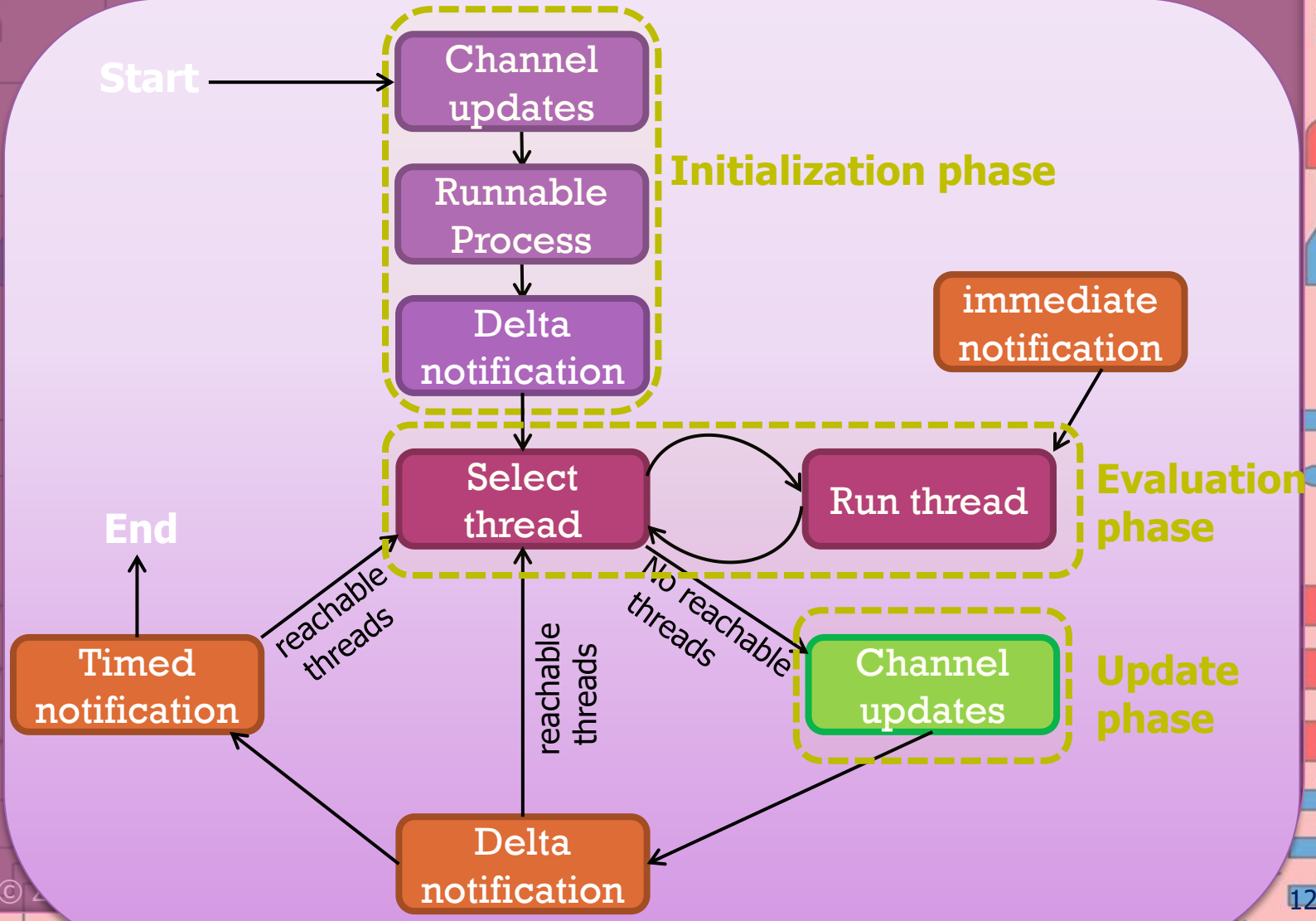
Behavior of the Scheduler

- The scheduler is event-driven and its primary is to trigger or resume the execution of the user processes
- Definitions
  - An **immediate notification** results from a call to member function notify of class sc_event with no arguments
  - A **delta notification** results from a call to member function notify of class sc_event with a zero-valued time argument
  - A **timed notification** results from a call to member function notify of class sc_event with a non-zero-valued time argument
  - A **time-out** results from certain calls to functions wait or next_trigger

Simulation

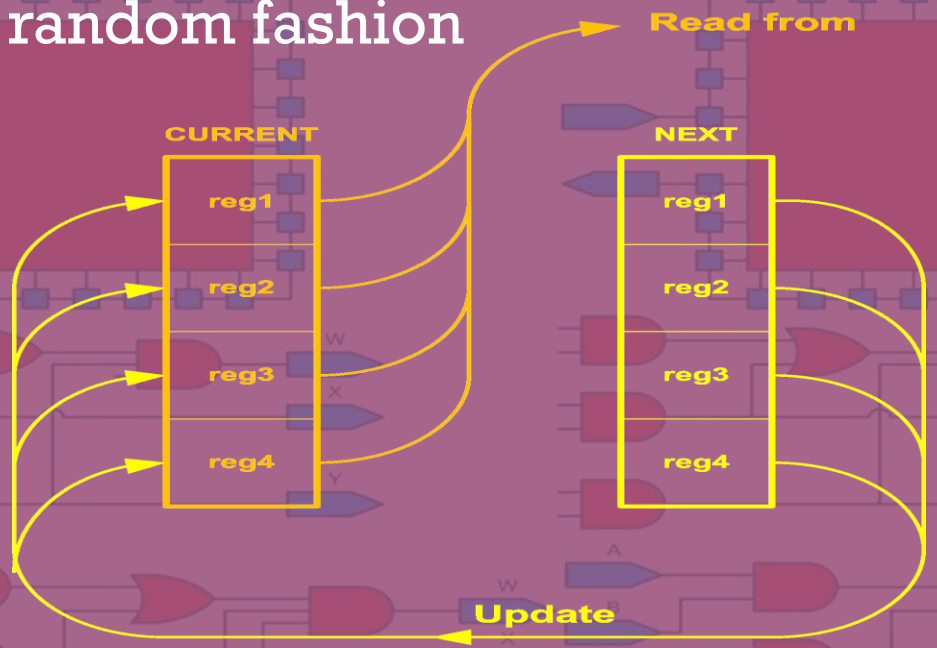
Behavior of the Scheduler

1. Init Add all processes to the set of runnable processes
2. Evaluate Select a ready to run process from the set of runnable processes & resume its execution. May result in more processes ready for execution due to Immediate Notification
3. Repeat 2 until no more processes to run
4. Update
5. If 2 or 4 resulted in delta event notifications, go back to 2
6. No more events, simulation is finished for current time
7. Advance to next simulation time that has pending events. If none, exit
8. Go back to step 2



Simulation

- Behavior of the Scheduler: Evaluation Phase
  - Processes with sensitivity become active when an event occurs on their sensitivity list
  - Processes that become active will be selected for execution in a random fashion

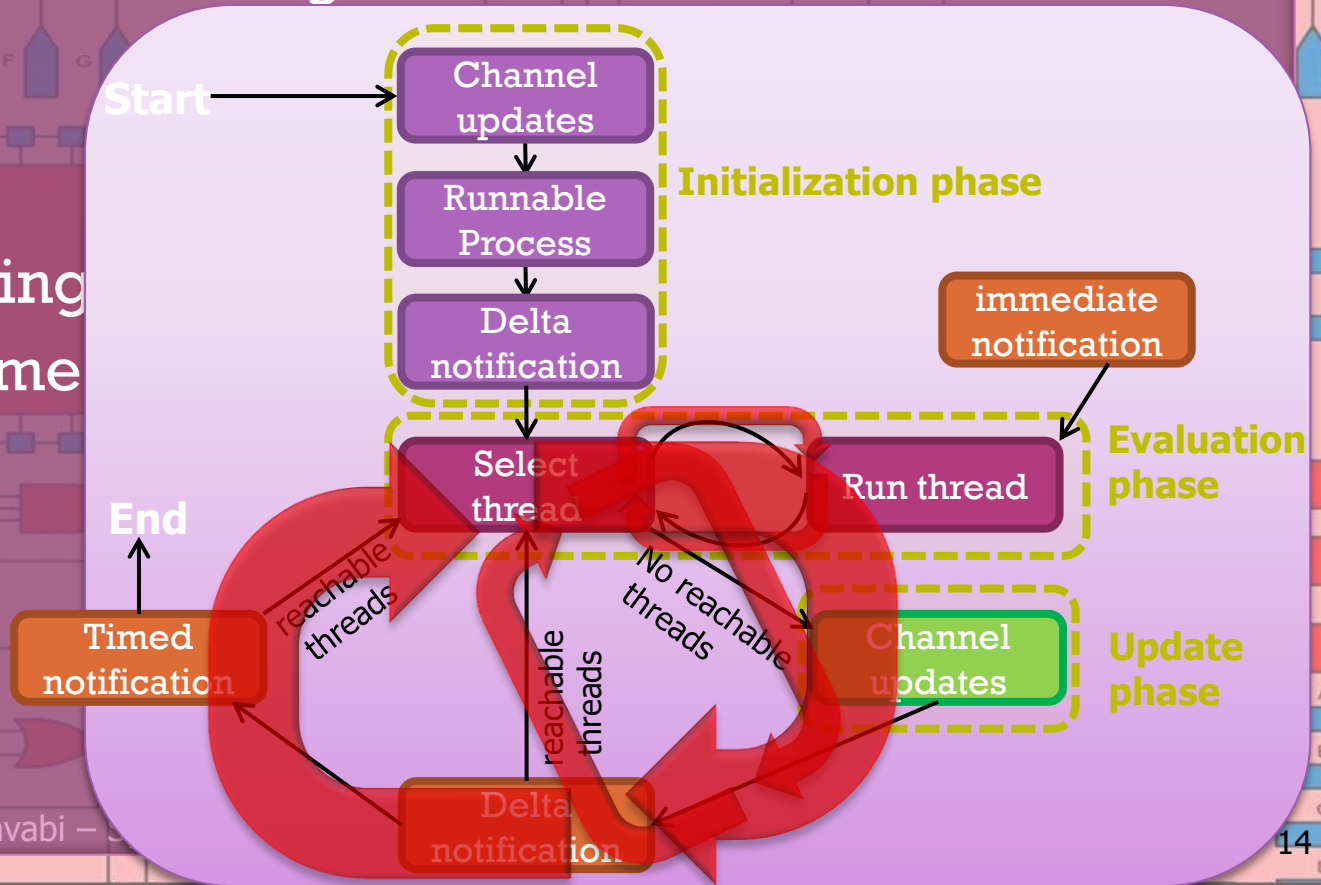


- Randomly select a signal
- Read its value from the CURRENT value table
- Write evaluated values in the NEXT table
- When all done, Update signal values
- If a New value is different from an Old value, put processes with those signals in the evaluation queue

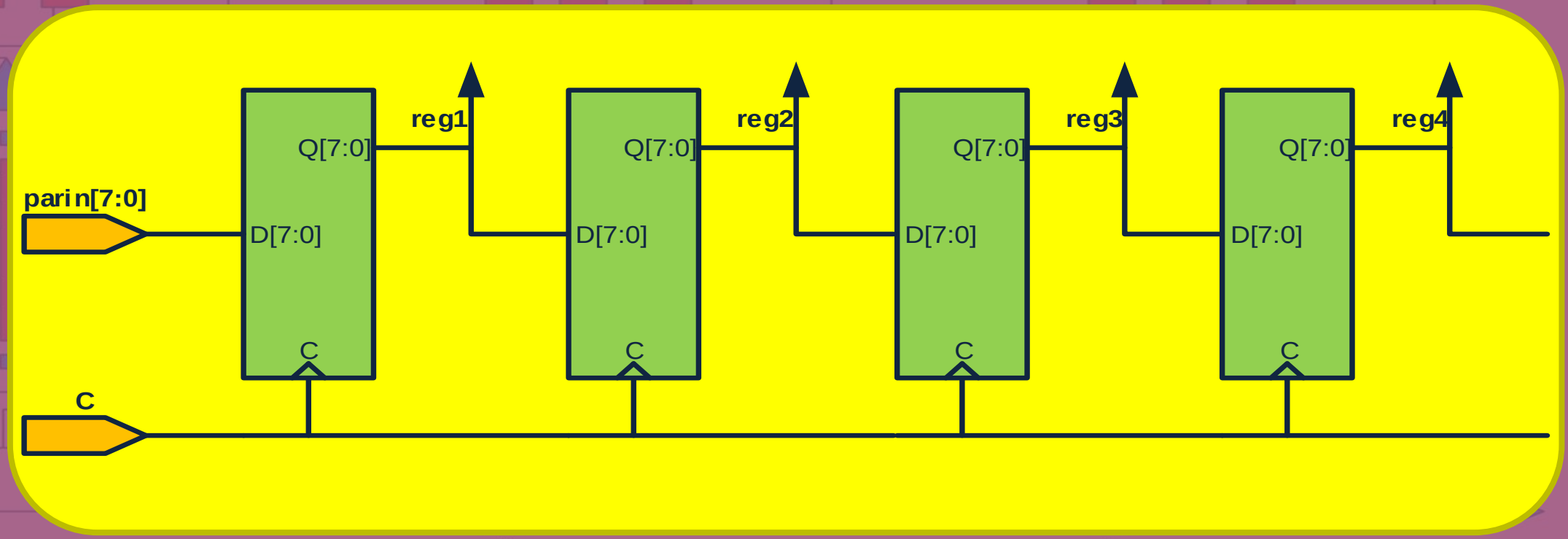
Simulation

Timing of the Scheduler

- A delta cycle is a sequence of steps in the scheduling algorithm consisting of the following steps in the order given
  - an evaluation phase
  - an update phase
  - a delta notification phase
- There is three loops in the scheduling algorithm that advance simulation time by one delta cycle



SystemC Simulation Kernel: Example 1



SystemC Simulation Kernel: Example 1

Class Definition

1

```
#include <systemc.h>

SC_MODULE(noParallelShifters)
{
     sc_in<bool> rst;
     sc_in<bool> clk;
     sc_in<sc_uint<8> > parin;
     sc_out<sc_uint<8> > reg1;
     sc_out<sc_uint<8> > reg2;
     sc_out<sc_uint<8> > reg3;
     sc_out<sc_uint<8> > reg4;

     SC_CTOR(noParallelShifters)
     {
          SC_METHOD(shifting);
          sensitive << rst << clk;
     }
     void shifting();
};
```

2

```
#include <systemc.h>

SC_MODULE(parallelShifters)
{
     sc_in<bool> rst;
     sc_in<bool> clk;
     sc_in<sc_uint<8> > parin;
     sc_out<sc_uint<8> > reg1;
     sc_out<sc_uint<8> > reg2;
     sc_out<sc_uint<8> > reg3;
     sc_out<sc_uint<8> > reg4;

     SC_CTOR(parallelShifters)
     {
          SC_METHOD(shifting);
          sensitive << rst << clk;
     }
     void shifting();
};
```

SystemC Simulation Kernel: Example 1

shifting Implementation

1

```
#include "parallelShifters.h"

void parallelShifters::shifting()
    {
        if (rst)
        {
            reg1 = 0; reg2 = 0; reg3 = 0; reg4 = 0;
        }
        else if (clk->event() && clk)
        {
            reg1 = parin;
            reg2 = reg1;
            reg3 = reg2;
            reg4 = reg3;
        }
    }

```

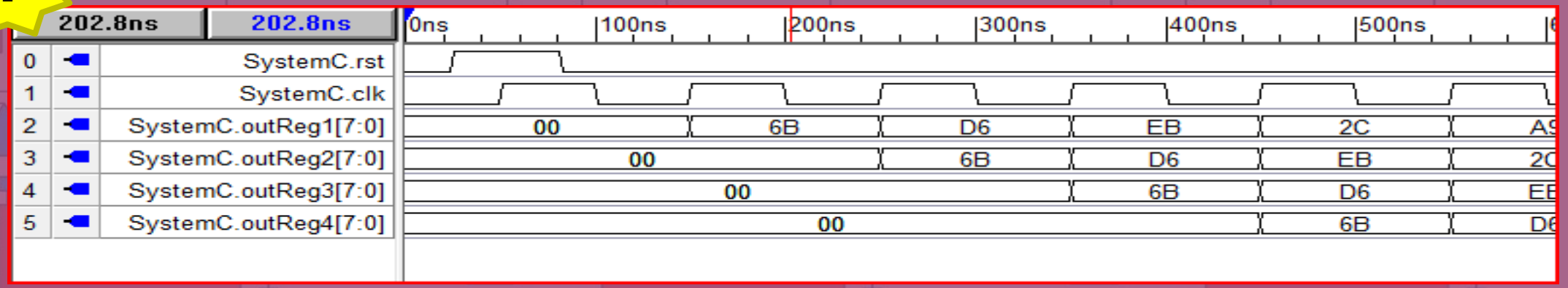
2

```
#include "noParallelShifters.h"

void noParallelShifters::shifting()
    {
        sc_uint<8> reg1var;
        sc_uint<8> reg2var;
        sc_uint<8> reg3var;
        sc_uint<8> reg4var;

        if (rst)
        {
            reg1var=0; reg2var=0; reg3var=0; reg4var=0;
        }
        else if (clk->event() && clk)
        {
            reg1var = parin;
            reg2var = reg1var;
            reg3var = reg2var;
            reg4var = reg3var;
        }
        reg1=reg1var; reg2=reg2var;
        reg3=reg3var; reg4=reg4var;
    }

```

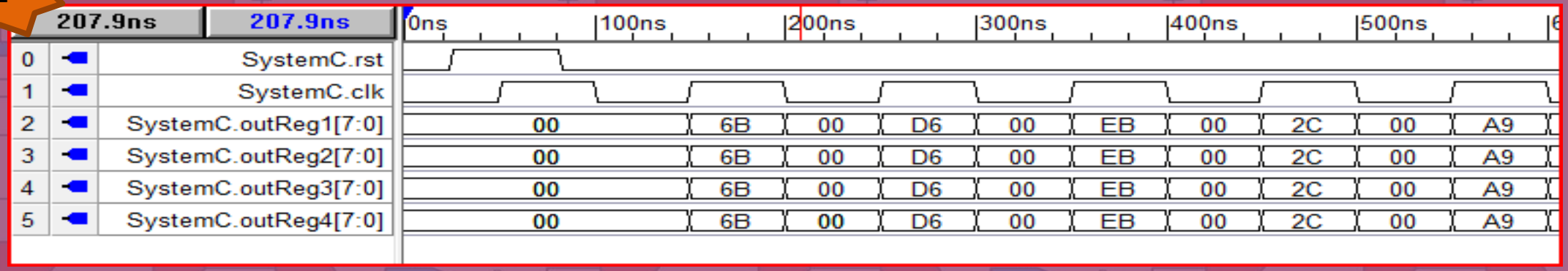
SystemC Simulation Kernel: Example 1

Output

1



2



SystemC Simulation Kernel: Example 2

Class Definition

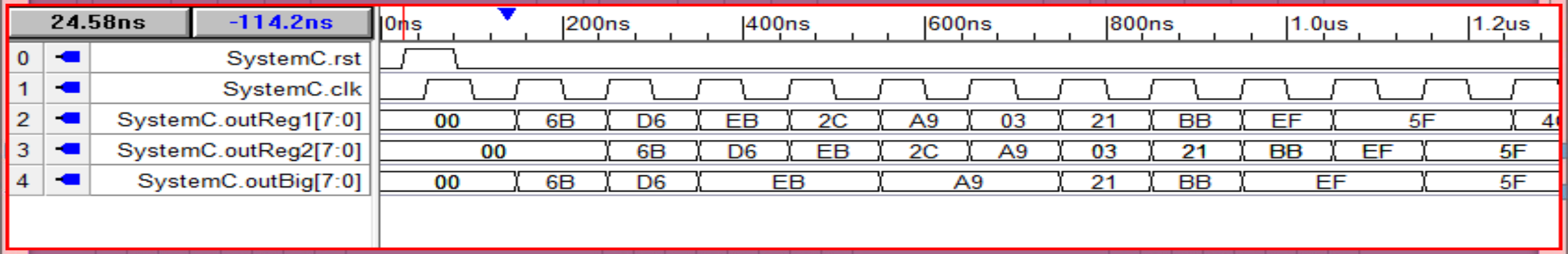
```
#include <systemc.h>

SC_MODULE(largerLast2)
{
     sc_in<bool> rst;
     sc_in<bool> clk;
     sc_in<sc_uint<8> > parin;
     sc_out<sc_uint<8> > reg1;
     sc_out<sc_uint<8> > reg2;
     sc_out<sc_uint<8> > larger;

     SC_CTOR(largerLast2)
     {
          SC_METHOD(saving);
          sensitive << rst.pos() << clk.pos();

          SC_METHOD(comparing);
          sensitive << reg1 << reg2;
     }
     void saving();
     void comparing();
};
```

```
#include "largerLast2.h"

void largerLast2::saving()
     {
          if (rst)
          {
               reg1 = 0; reg2 = 0;
          }
          else
          {
               reg1 = parin;
               reg2 = reg1;
          }
     }

void largerLast2::comparing()
     {
          if (reg1 > reg2)
          {
               larger = reg1;
          }
          else
          {
               larger = reg2;
          }
     }
```

Implementation

SystemC Simulation Kernel: Example 2

Output



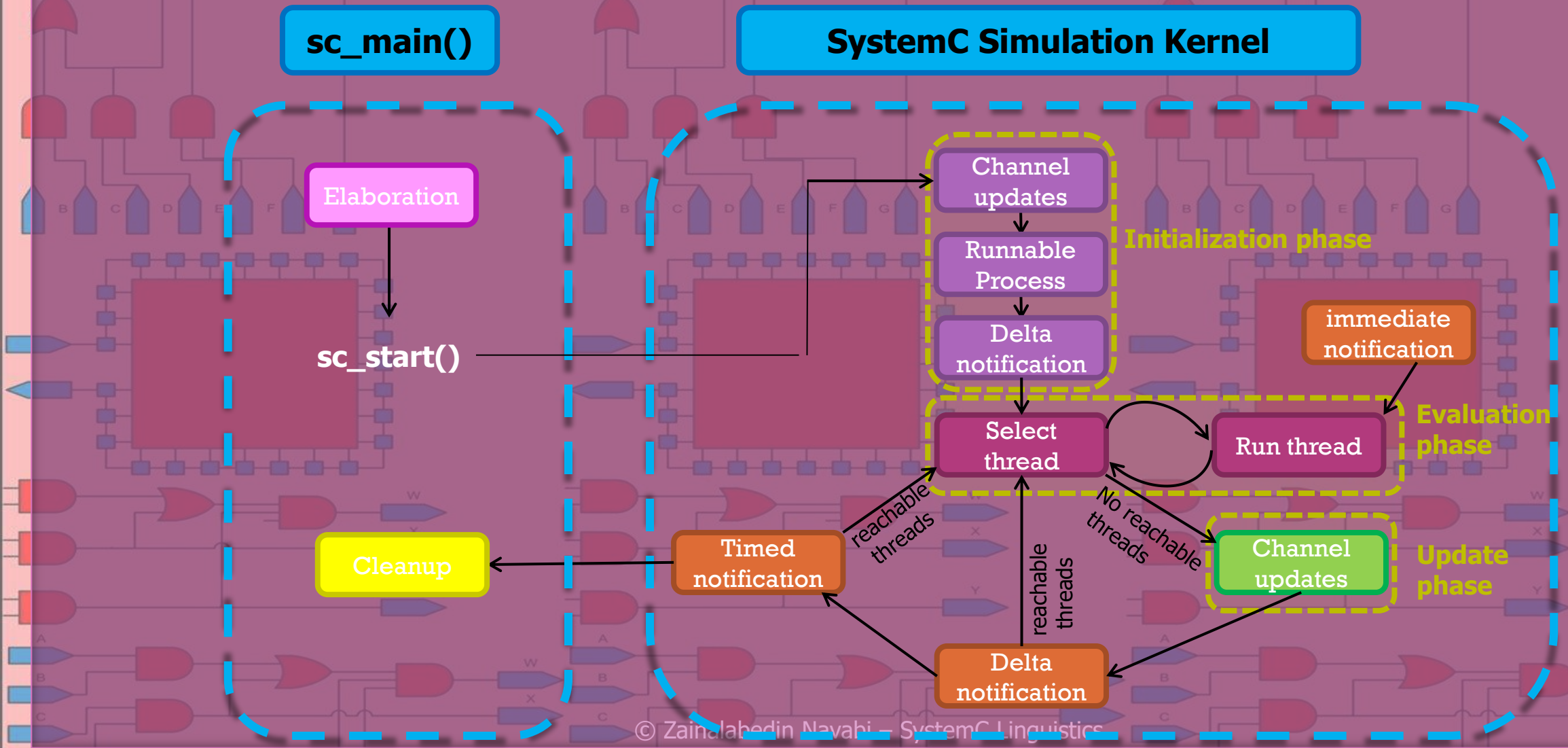
SystemC Linguistics

- SystemC Execution
  - Elaboration
    - Creating a module hierarchy
    - Bounding ports & exports to channels
    - Setting time resolution
  - Simulation
    - Behavior of the Scheduler
    - Timing of the Scheduler
    - Examples
  - Running Elaboration & Simulation
    - Under Application Control
    - Under Kernel Control
  - Simulation Control
- + Basic Syntax & Semantics
- + Language Semantics
- + Interfaces
- Examples
- Summary

Running Elaboration & Simulation

- Under Application Control
  - Using sc_main
    - Elaboration consists of the execution of the sc_main function from the start of sc_main to the point immediately before the first call to the function sc_start.
    - Syntax: int sc_main(int argc, char* argv[]);
  - Using sc_start
    - Function sc_start may be called from function sc_main
    - Syntax: void sc_start();
      void sc_start(const sc_time&);
      void sc_start(double , sc_time_unit);
- Under kernel Control

Running Elaboration & Simulation



SystemC Linguistics

- SystemC Execution
  - Elaboration
    - Creating a module hierarchy
    - Bounding ports & exports to channels
    - Setting time resolution
  - Simulation
    - Behavior of the Scheduler
    - Timing of the Scheduler
    - Examples
  - Running Elaboration & Simulation
    - Under Application Control
    - Under Kernel Control
  - Simulation Control
- + Basic Syntax & Semantics
- + Language Semantics
- + Interfaces
- Examples
- Summary

SystemC Control

- Function sc_stop, sc_set_stop_mode, and sc_get_stop_mode
- Functions to detect pending activity
- Function sc_pause
- Function sc_time_stamp
- Function sc_delta_count
- Function sc_is_running
- Function sc_get_status

```
namespace sc_core {
    enum sc_stop_mode{
        SC_STOP_FINISH_DELTA ,SC_STOP_IMMEDIATE
    };
    extern void sc_set_stop_mode(sc_stop_mode mode);
    extern sc_stop_mode sc_get_stop_mode();
    void sc_stop();
    bool sc_pending_activity();
    ...
    void sc_pause();
    const sc_time& sc_time_stamp();
    const sc_dt::uint64 sc_delta_count();
    bool sc_is_running();
    sc_status sc_get_status();
    }
```

SystemC Linguistics

+ SystemC Execution
– Basic Syntax & Semantics
  - Modules
  - Ports
  - Signals
  - Internal Data Storage
  – Instances
    - Method 1
    - Method 2
  – Port Binding
    - Positional Port Binding
    - Named Port Binding
  – Constructor
    - SC_CTOR
    - SC_HAS_PROCESS
  – Processes
    - SC_THREAD
    - SC_METHOD
  + Language Semantics
  + Interfaces
  - Examples
  - Summary

Modules

- A module is a structural entity, which can contain
  - Processes
  - Ports
  - Channels
  - Member functions not registered as processes
  - Instances of other modules
- A module is the foundation of structural hierarchy
- Modules allow designers to hide internal data representation and algorithms from other modules
- Designers are forced to use public interfaces to other modules, thus making the entire system easier to change and maintain

Modules: *sc_module* Class Definition

```
154 class sc_module
155 : public sc_object, public sc_process_host
156 {
157     friend class sc_module_name;
158     friend class sc_module_registry;
159     friend class sc_object;
160     friend class sc_port_registry;
161     friend class sc_process_b;
162     friend class sc_simcontext;
173 protected:
174     //...
189     // constructor
190     sc_module( const char* nm );
191     sc_module( const std::string& nm );
192     sc_module( const sc_module_name& nm ); /* for those used to old
193     sc_module();
194
195 public:
222     //...
223     // static sensitivity for SC_THREADs and SC_CTHREADs
224     void wait() { ... }
227     // dynamic sensitivity for SC_THREADs and SC_CTHREADs
228     void wait( const sc_event& e ) { ... }
263     //...
264     // static sensitivity for SC_METHODs
265     void next_trigger() { ... }
269     // dynamic sensitivty for SC_METHODs
270     void next_trigger( const sc_event& e )
271         { ::sc_core::next_trigger( e, simcontext() ); }
272     //...
339     // These are protected so that user derived classes can refer to them.
340     sc_sensitive     sensitive;
341     sc_sensitive_pos sensitive_pos;
342     sc_sensitive_neg sensitive_neg;
343     //...
432 };
```

sc_module.h

Various constructor forms
We use the one with a simple module name

Member functions for method sensitivity

Modules: *sc_module* Class Definition

```
154 class sc_module
155 : public sc_object, public sc_process_host
156 {
157     friend class sc_module_name;
158     friend class sc_module_registry;
159     friend class sc_object;
160     friend class sc_port_registry;
161     friend class sc_process_b;
162     friend class sc_simcontext;
```

SC_MODULE macro makes it easier to declare an object of sc_module class

#define SC_MODULE (*module_name*) class *module_name*: public sc_module

Module Declaration

```
SC_MODULE (module_name){
    port declarations;
    internal signal declarations;
    concurrent part declarations;
    member data declarations;
    member function declarations;
    module constructor;
};
```

Using the macro

```
class module_name: public sc_module {
    port declarations;
    internal signal declarations;
    concurrent part declarations;
    member data declarations;
    member function declarations;
    module constructor;
};
```

Explicit use of classes

SystemC Linguistics

+ SystemC Execution
– Basic Syntax & Semantics
  - Modules
  - Ports
  - Signals
  - Internal Data Storage
  - – Instances
    - Method 1
    - Method 2
  - – Port Binding
    - Positional Port Binding
    - Named Port Binding
  - – Constructor
    - SC_CTOR
    - SC_HAS_PROCESS
  - – Processes
    - SC_THREAD
    - SC_METHOD
  - + Language Semantics
  - + Interfaces
  - Examples
  - Summary

Ports

- Ports are the external interface of a module
- Ports pass data to and from the processes of a module, and trigger actions within the module
- Ports declare as : in, out or inout
- You declare the data type of the ports as any C++ data type, SystemC data type, or user defined type
- sc_in, sc_out, and sc_inout are predefined by SystemC

```
SC_MODULE (module_name) {
    sc_in <bool> a, b;
    sc_out <sc_lv <8> > cv;
    sc_inout <int> c;
    sc_port <ud_type> p;
};
```

A space is required by the C++ compiler

Ports

- Ports are members of the module
- Ports are always bound to signals except for port-to-port binding
- To read/write a value from/to a port
  - Use assignment
  - Use the ->read()/->write() method
- When a port is read, the value of the signal connected to the port is returned
- When a port is written, the new value will be written to the signal when the process performing the wire operation has finished execution, or has been suspended

.read()/.write() are overloaded

SystemC Linguistics

+ SystemC Execution
– Basic Syntax & Semantics
  - Modules
  - Ports
  - Signals
  - Internal Data Storage
  – Instances
    - Method 1
    - Method 2
  – Port Binding
    - Positional Port Binding
    - Named Port Binding
  – Constructor
    - SC_CTOR
    - SC_HAS_PROCESS
  – Processes
    - SC_THREAD
    - SC_METHOD
  + Language Semantics
  + Interfaces
  - Examples
  - Summary

Signals

- Signals are used for communication
- A signal connects the port of one module to the port of another
- Signals aren't declared as in, out, or inout
- You declare the data type of the signals as any C++ data type, SystemC data type, or user defined type
- The direction of the data transfer depends on the port kinds of the connecting components

```
SC_MODULE (module_name) {
   //port declarations
   sc_signal <bool> a;
   sc_signal <int> b;
   sc_signal <sc_lv <8> > vc;
};
```

A space is required by the C++ compiler

Signals

- Signals may be
  - members of a module (used for internal communication)
  - used at top-level to connect the modules
- All processes executing during a time step will see the old value of the signal (This semantic is the same as VHDL signal operation)
- To read/write a value from/to a port
  - Use assignment
  - Use the .read()/.write() method

```
sc_signal <int> sig;
int a;
...
a = sig;
sig = 10;
```

recommended

```
sc_signal <int> sig;
int a;
...
a = sig.read();
sig.write(10);
```

SystemC Linguistics

+ SystemC Execution
– Basic Syntax & Semantics
  - Modules
  - Ports
  - Signals
  - Internal Data Storage
  – Instances
    - Method 1
    - Method 2
  – Port Binding
    - Positional Port Binding
    - Named Port Binding
  – Constructor
    - SC_CTOR
    - SC_HAS_PROCESS
  – Processes
    - SC_THREAD
    - SC_METHOD
+ Language Semantics
+ Interfaces
  Examples
  Summary

Internal Data Storage

- Local variables can be declared for storage of data within a module
- Internal Data Storage can be of any C++ data type, SystemC data type, or user defined type
- They are not visible outside the module
- They don't advance delta time

```
SC_MODULE (module_name) {
    //port declarations
    //signal declarations
    int var;
};
```

SystemC Linguistics

+ SystemC Execution
– Basic Syntax & Semantics
  - Modules
  - Ports
  - Signals
  - Internal Data Storage
  - Instances
    - Method 1
    - Method 2
  - Port Binding
    - Positional Port Binding
    - Named Port Binding
  - Constructor
    - SC_CTOR
    - SC_HAS_PROCESS
  - Processes
    - SC_THREAD
    - SC_METHOD
+ Language Semantics
+ Interfaces
  Examples
  Summary

Instances: Method 1

- Using constructor initialization list

```
#include "submodule_name.h"
SC_MODULE (module_name)
{
   //port declarations
   //signal declarations
   //internal data storage declarations

     submodule_name instance_name;

   //member data and function declarations

   SC_CTOR (module_name): instance_name("instance_name")
   {
     instance_name.submod_port_name(module_port_name);
       . . .
   }
};
```

ModuleInst1.h

Instantiation

Port Binding

Instances: Method 2

- Using pointers and dynamic memory allocation

```
#include "submodule_name.h"
SC_MODULE (module_name)
{
   //port declarations
   //signal declarations
   //internal data storage declarations

      submodule_name *instance_name;

   //member data and function declarations
   SC_CTOR (module_name)
   {
      instance_name = new submodule name("instance name");
      instance_name->submod_port_name(module_port_name);
      . . .
   }
};
```

ModuleInst2.h

Instantiation

Port Binding

SystemC Linguistics

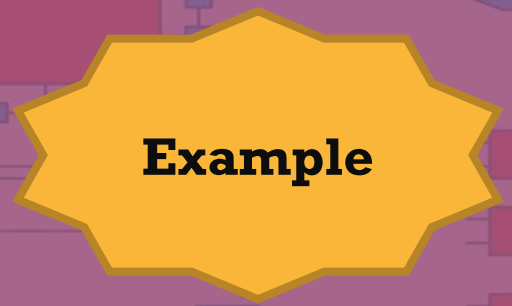
+ SystemC Execution
– Basic Syntax & Semantics
  - Modules
  - Ports
  - Signals
  - Internal Data Storage
  – Instances
    - Method 1
    - Method 2
  – Port Binding
    - Positional Port Binding
    - Named Port Binding
  – Constructor
    - SC_CTOR
    - SC_HAS_PROCESS
  – Processes
    - SC_THREAD
    - SC_METHOD
  + Language Semantics
  + Interfaces
  - Examples
  - Summary

Port Binding

- There are two ways to bind ports
  - Positional Port Binding
    - Implicitly binds a port to a channel
    - It works very well for small instantiations with few ports (64 or fewer ports)
    - The order and number of ports is very important
    - Syntax: instance_name (module_port_name1, module_port_name2, …);
  - Named Port Binding
    - Explicitly binds a port to a channel
    - It's better for instantiations with a large number of ports
    - The order doesn't matter
    - Syntax 1: instance_name->submod_port_name(module_port_name);
    - Syntax 2: instance_name.submod_port_name(module_port_name);

Port Binding: Positional

```
SC_MODULE(M)
{
    sc_inout<int> P, Q, R; // Ports
    ...
};
```




```
SC_MODULE(Top)
{
    sc_inout <int> A, B;
    sc_signal<int> C;
    M m; // Module instance
    SC_CTOR(Top): m("m")
    {
        m1(A, B, C); // Binds P-to-A, Q-to-B, R-to-C
    }
    ...
};
```

Port Binding: Named

```
SC_MODULE(M)
{
    sc_inout<int> P, Q, R, S; // Ports
    ...
};
```

Example

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

SystemC Linguistics

+ SystemC Execution
– Basic Syntax & Semantics
  - Modules
  - Ports
  - Signals
  - Internal Data Storage
  – Instances
    - Method 1
    - Method 2
  – Port Binding
    - Positional Port Binding
    - Named Port Binding
  – Constructor
    - SC_CTOR
    - SC_HAS_PROCESS
  – Processes
    - SC_THREAD
    - SC_METHOD
  + Language Semantics
  + Interfaces
  - Examples
  - Summary

Constructor

- Initializations
- Allocating sub-modules
  - Using C++ new
  - Like hardware instantiation
  - Passes instance name to an allocated sc_module
- Connecting sub-modules
  - Referencing member variables in C++
  - Like hardware port map
- Registering module processes
  - Performs initial function call
  - Mimicking hardware concurrency
- . . .

Implements concurrent bodies required for hardware modeling

Constructor

- Attaching static sensitivity to processes
  - Links with SystemC simulation kernel
  - Defines hardware sensitivity list
    - For clocking
    - For combinational circuit inputs
- Set initial values
  - Cannot be changed during simulation
- Pass class pointers for shared access
  - Memory access
  - Communication protocol access
  - Global signals, or shared variables

Constructor: SC_CTOR

- SC_CTOR: Inline constructor
  - Easy to use
  - All done in sc_module

```
SC_MODULE (module_name) {
   ...
   SC_CTOR (module_name) : Optional_initializations {
      //necessary variable initializations,
      //memory allocations,
      //process registrations
      ...
   } //constructor implementation
   ...
}; //module definition
```

Constructor: SC_HAS_PROCESS

- SC_HAS_PROCESS: Alternative constructor format
  - Allows passing additional arguments
    - Perhaps for a configurable module
  - Implementation can go in a *.cpp* file

```
SC_MODULE (module_name) {
   ...
   SC_HAS_PROCESS (module_name);
      module_name(sc_module_name instname, other_args…);

   // other module tasks
};
```

```
module_name::module_name (sc_module_name instname, other_args…)
                         :sc_module(instname), other_arg_inits) {
   ...
   // constructor implementation
   ...
}
```

Constructor: SC_HAS_PROCESS

- SC_HAS_PROCESS: Example (.h)

```
19          REG1 = new udCounterRaEL("REG1_Instance", -1);
```

```
1  // udCounterRaEL: A parametric UP-Down counter with asynchronous reset, elanble
   and load
2
3  #include <systemc.h>
4
5  SC_MODULE(udCounterRaEL)
6  {
7      sc_in<sc_logic> rst, clk, cen, pld;
8      sc_in<sc_lv<8> > parin;
9      sc_out<sc_lv<8> > cntout;
10
11
12     SC_HAS_PROCESS(udCounterRaEL);
13         udCounterRaEL (sc_module_name NAME, sc_int<8> NUM);
14
15     void counting();
16     private:
17         sc_int<8> cntVal;
18
19 };
20
```

Constructor: SC_HAS_PROCESS

- SC_HAS_PROCESS: Example (.cpp)

```
1  #include "udCounterRaEL.h"
2
3  udCounterRaEL::udCounterRaEL (sc_module_name NAME, sc_int<8> NUM)
4                  : sc_module(NAME), cntVal(NUM)
5  {
6      cout << "Count value: [ " << cntVal << " ] \n";
7      SC_THREAD(counting);
8         sensitive << rst << clk;
9  }
10
11 void udCounterRaEL::counting() {
12     while(true) {
13         if (rst=='1') {
14             cntout = sc_lv<8>(0);
15         }
16         else if (clk->event() && (clk=='1')) {
17             if(pld=='1') cntout = parin;
18             else if(cen=='1') cntout=(sc_uint<8>)cntout+cntVal;
19         }
20         wait();
21     }
22 }
```

SystemC Linguistics

+ SystemC Execution
– Basic Syntax & Semantics
  - Modules
  - Ports
  - Signals
  - Internal Data Storage
  – Instances
    - Method 1
    - Method 2
  – Port Binding
    - Positional Port Binding
    - Named Port Binding
  – Constructor
    - SC_CTOR
    - SC_HAS_PROCESS
  – Processes
    - SC_THREAD
    - SC_METHOD
+ Language Semantics
+ Interfaces
  Examples
  Summary

Processes

- Processes:
  - Accept no arguments
  - Produce no output
- Must be contained within a module
- Are registered as processes with the SystemC kernel, using a process declaration in the module constructor
- Can have static and dynamic sensitivity

```
SC_MODULE (module_name) {
    //port declarations
    //signal declarations
    ...
 public:
    void method_process ();
    void thread_process ();

    SC_CTOR (module_name) {
        SC_METHOD (method_process);
        //sensitivity list
        SC_THREAD (thread_process);
        //sensitivity list
    }
};
```

These are concurrent processes being declared

Process Registration

Processes: SC_THREAD

- SC_THREAD (Like Verilog initial, or VHDL PROCESS w/o sensitivity)
- Can have static sensitivity (in the constructor)
- Can have dynamic sensitivity (in its implementation)
- Exit from an SC_THREAD suspends it forever (cannot be resumed)
- Infinite loop prevents exit, makes it possible to resume

```
SC_MODULE (module_name) {
   ...
public:
   void thread_process ();

   SC_CTOR (module_name) {
      SC_THREAD (thread_process);
      //static sensitivity list
   }
};
```

```
void module_name::thread_process () {
     // Thread implementation
    // Can use dynamic sensitivity
   // Such as wait(. . .)
}
```

List of events that resume its operation after a wait().

Processes: SC_THREAD

- An SC_THREAD is called only once, and it suspends itself
- If not suspended forever, can be resumed by static or dynamic sensitivity
- wait() suspends it to be resumed by static sensitivity
- Other forms of wait(. . .) cause suspensions to be resumed with dynamic sensitivity
- Because it can be suspended, it can implement *blocking processes*
- Declared variables in an SC_THREAD persist for the entire simulation

Processes: SC_THREAD

- SC_THREAD: Example (.h)

```
1  // udCounterRaEL: A parametric UP-Down counter with asynchronous reset, elanble
   and load
2
3  #include <systemc.h>
4
5  SC_MODULE(udCounterRaEL)
6  {
7      sc_in<sc_logic> rst, clk, cen, pld;
8      sc_in<sc_lv<8> > parin;
9      sc_out<sc_lv<8> > cntout;
10
11
12     SC_HAS_PROCESS(udCounterRaEL);
13         udCounterRaEL (sc_module_name NAME, sc_int<8> NUM);
14
15     void counting();
16     private:
17         sc_int<8> cntVal;
18
19 };
20
```

Processes: SC_THREAD

- SC_THREAD: Example (.cpp)

```
1  #include "udCounterRaEL.h"
2
3  udCounterRaEL::udCounterRaEL (sc_module_name NAME, sc_int<8> NUM)
4                  : sc_module(NAME), cntVal(NUM)
5  {
6      cout << "Count value: [ " << cntVal << " ] \n";
7      SC_THREAD(counting);
8         sensitive << rst << clk;
9  }
10
11 void udCounterRaEL::counting() {
12     while(true) {
13         if (rst=='1') {
14             cntout = sc_lv<8>(0);
15         }
16         else if (clk->event() && (clk=='1')) {
17             if(pld=='1') cntout = parin;
18             else if(cen=='1') cntout=(sc_uint<8>)cntout+cntVal;
19         }
20         wait();
21     }
22 }
```

Only static sensitivity is used here.

Processes: SC_METHOD

- SC_METHOD
  - Like Verilog always, or VHDL PROCESS w/ sensitivity, a VHDL concurrent procedure
- Can have static sensitivity (in the constructor)
- Cannot have dynamic sensitivity (no wait inside)
- An SC_METHOD with sensitivity never suspends
- w/o sensitivity, runs only once

```
SC_MODULE (module_name) {
  ...
public:
  void method_process ();

  SC_CTOR (module_name) {
    SC_METHOD (method_process);
    //static sensitivity list
  }
};
```

```
void module_name::method_process () {
   // Method implementation
  // Can manipulate its sensitivity
 // with next_trigger(…)
}
```

List of events that resume its operation after a round of execution.

Processes: SC_METHOD

- An SC_METHOD is called once for every event on its static sensitivity
- An SC_METHOD always runs from the beginning to the end
- Cannot be used as a *blocking process*
- Declared variables are re-established (no history)

Processes: SC_METHOD

- SC_METHOD: Example (.h)

```
1  // udCounterRaEL: A parametric UP-Down counter with asynchronous reset, elanble
   and load
2
3  #include <systemc.h>
4
5  SC_MODULE(udCounterRaEL)
6  {
7      sc_in<sc_logic> rst, clk, cen, pld;
8      sc_in<sc_lv<8> > parin;
9      sc_out<sc_lv<8> > cntout;
10
11
12     SC_HAS_PROCESS(udCounterRaEL);
13         udCounterRaEL (sc_module_name NAME, sc_int<8> NUM);
14
15     void counting();
16     private:
17         sc_int<8> cntVal;
18
19 };
20
```

Processes: SC_METHOD

- SC_METHOD: Example (.cpp)

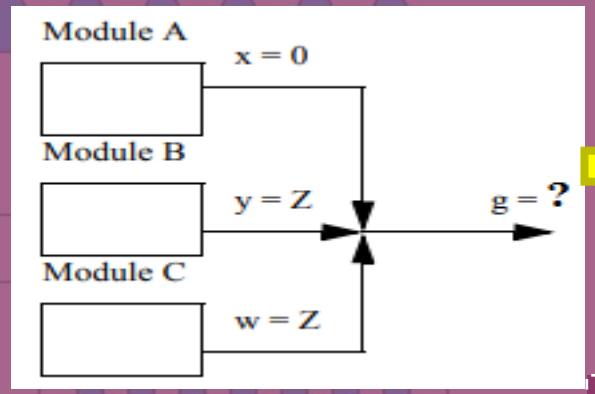
```
1  #include "udCounterRaEL.h"
2
3  udCounterRaEL::udCounterRaEL (sc_module_name NAME, sc_int<8> NUM)
4                  : sc_module(NAME), cntVal(NUM)
5  {
6      cout << "Count value: [ " << cntVal << " ] \n";
7      SC_METHOD(counting);
8         sensitive << rst.pos() << clk.pos();
9  }
10
11 void udCounterRaEL::counting() {
12     if (rst=='1') {
13         cntout = sc_lv<8>(0);
14     }
15     else if (clk=='1') {
16         if(pld=='1') cntout = parin;
17         else if(cen=='1') cntout=(sc_uint<8>)cntout+cntVal;
18     }
19 }
```

SystemC Linguistics - Cont.

+ SystemC Execution
+ Basic Syntax & Semantics
– Language Semantics
  Resolved Logic Vector
  – Types & Operations
    • Basic Type
    • Type Operation
  – Time, Events & Sensitivity Control
    • Time
    • Events
    • Sensitivity
      • Static Sensitivity
      • Dynamic Sensitivity
+ Interfaces
  Examples
  Summary

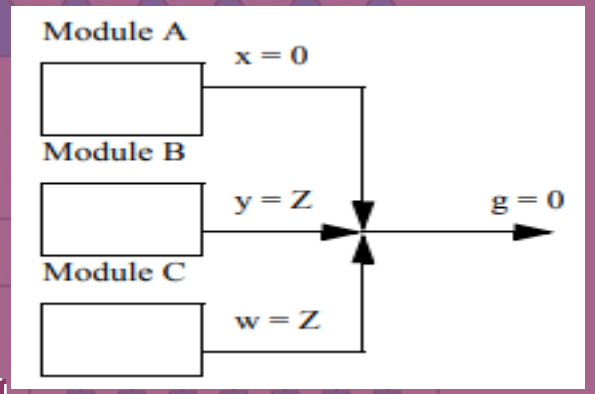
Resolved Logic Vector

- When a signal is driven by more than one deriver, bus resolution occurs
- SystemC uses a Resolved Logic Vector signal type to handle this issue



Resolution of logic values

	0	1	Z	X
0	0	X	0	X
1	X	1	1	X
Z	0	1	Z	X
X	X	X	X	X




```
SC_MODULE (circuit) {
    sc_in_rv <n> a;
    sc_out_rv <n> b;
    sc_inout_rv <n> c;

    sc_signal_rv<n> sig;
};
```

Resolved Port

Resolved Signal

SystemC Linguistics - Cont.

+ SystemC Execution
+ Basic Syntax & Semantics
– Language Semantics
  - Resolved Logic Vector
  - – Types & Operations
    - Basic Type
    - Type Operation
  - – Time, Events & Sensitivity Control
    - Time
    - Events
    - Sensitivity
      - Static Sensitivity
      - Dynamic Sensitivity

+ Interfaces
  Examples
  Summary

Basic Types

- All native C++ types are supported within a SystemC application
- SystemC provides additional data type classes within the sc_dt namespace for describing hardware where C++ data types are insufficient
- All SystemC data types T support the streaming operator to print it onto a stream

```
ostream& operator << ( ostream&, T );
```

Basic Types

- Use C++ types as before
  - int
  - unsigned
  - unsigned long
  - short int
  - float
  - double
  - string
  - bool
  - enum

Basic Types

- Use specific SystemC types as well
  - *sc_int<bits>* and *sc_uint<bits>*:
    - for bit representation, mainly with synthesis
  - *sc_logic* and *sc_lv<bits>*:
    - use for '0', '1', 'Z', and 'X' values
  - *sc_bit* and *sc_bv<bits>*:
    - use for logic '0' and '1'

Basic Types

- Use specific SystemC types as well
  - *sc_bigint<bits>* and *sc_biguint<bits>*:
    - integer with bit length specification
  - *sc_fixed<static_args>* and *sc_ufixed<static_args>*:
    - fixed point with static arguments
  - *sc_fix(options)* and *sc_ufix(options)*:
    - fix point with non-static arguments

Basic Types

SystemC data types

Class template	Base class	Generic base class	Representation	Precision
sc_int	sc_int_base	sc_value_base	signed integer	limited
sc_uint	sc_uint_base	sc_value_base	unsigned integer	limited
sc_bigint	sc_signed	sc_value_base	signed integer	finite
sc_biguint	sc_unsigned	sc_value_base	unsigned integer	finite
sc_fixed	sc_fix	sc_fxnum	signed fixed-point finite	finite
sc_ufixed	sc_ufix	sc_fxnum	unsigned fixed-point finite	finite
sc_fixed_fast	sc_fix_fast	sc_fxnum_fast	signed fixed-point finite	limited
sc_ufixed_fast	sc_ufix_fast	sc_fxnum_fast	unsigned fixed-point finite	limited
		sc_fxval	fixed-point	variable
		sc_fxval_fast	fixed-point	limited-variable
	sc_logic		single bit	
sc_bv	sc_bv_base		bit vector	
sc_lv	sc_lv_base		bit vector	

Type operations: bit

- *sc_bit* and *sc_bv<bits>*:

sc_bit operations

sc_bv operations

Bitwise	~	&	|	^	>>	<<
Assignment	=	&=	|=	^=		
Equality	==	!=				
Bit Select	[b]					
Part Select	range()					
Concatenation	(,)					
Reduction	and_reduce()		or_reduce()		xor_reduce()	

Type operations: bit

- *sc_bit* and *sc_bv<bits>*:
  - standard bit and bit vector examples

Declaration	Examples
sc_bit B1, B2, B3;	B2=B1|B3;
	B2|=B1;
sc_bv<8> R1, R2, R3;	B1=R1.and_reduce();
	R3=(B1, B2, R2.range(0,5));
	B2=R1[7];
	R3=(R3[7], R3.range(7,1));
sc_bv<8> R4 = “11000011”;	
sc_signal<sc_bv<64> > iobus, alubus, Reg1;	

Type operations: logic

- *sc_logic* and *sc_lv<bits>*:
  - standard logic bit and logic vector examples

Declaration	Examples
sc_logic B1, B2, B3;	B2=B1|B3;
	B2|=B1;
sc_lv<8> R1, R2, R3;	B1=R1.and_reduce();
	R3=(B1, R2.range(7,1));
	R3="11110000";
	R2= R3.range(0,7);
	R2="0011XXZZ"

Type operations: logic

- *sc_logic* and *sc_lv<bits>*:
  - use to_string() method for printing

Declaration	Examples
sc_lv<8> R1;	R1 = “1100XXZZ”; cout << R1.to_string(); // Prints: “1100XXZZ”
	R1 = 23; cout << R1; // Prints: 00010111

Type operations: to_string

- *to_string(sc_numrep rep);*
  - To turn numbers to a specific format

Declaration	Examples
sc_int<8> R1;	R1 = 25;
	cout << R1.to_string(SC_BIN); // Prints: 0b00011001
	cout << R1.to_string(SC_DEC); // Prints: 0d25
	cout << R1.to_string(SC_OCT); // Prints: 0o031
	cout << R1.to_string(SC_HEX); // Prints: 0x19

  - Can use the same formats for string assignments
    - E.g., R1 = "0d25";

Type operations: int

- *sc_int<bits>* and *sc_uint<bits>*:
  - 1 to 64 bit integers, slower than C++ types

Operations

Bitwise	~	&	|	^	>>	<<			
Arithmetic	+	-	*	/	%				
Assignment	=	+=	-=	*=	/=	%=	&=	|=	^=
Equality	==	!=							
Relational	<	<=	>	>=					
Auto inc, dec	++	--							
Bit Select	[b]								
Part Select	range()								
Concatenation	(,)								

Type operations: int

- *sc_int* and *sc_uint*:
  - for bit representation, mainly with synthesis

Declaration	Examples
sc_int<8> R1, R2, R3;	R2=R1|R3;
	R2=R1+R3;
	R2=R2>>3;
sc_logic leftBit;	leftBit=R2[7];
sc_int<4> R4, R5, R6;	R4=R1.range(7,4);
	(R4, R5) = R1;
sc_uint<8> uR1;	uR1=R1; //conversion
sc_signal<sc_int<64> > iobus, alubus, Reg1;	

Type operations: bigint

- *sc_bigint<bits>* and *sc_biguint<bits>*:
  - any number of bit integers

Operations

Bitwise	~	&	|	^	>>	<<			
Arithmetic	+	-	*	/	%				
Assignment	=	+=	-=	*=	/=	%=	&=	|=	^=
Equality	==	!=							
Relational	<	<=	>	>=					
Auto inc, dec	++	--							
Bit Select	[b]								
Part Select	range()								
Concatenation	(,)								

Type operations: bigint

- *sc_bigint* and *sc_biguint*:
  - for large size integers

Declaration	Examples
sc_biguint<128> R1, R2;	
sc_biguint<64> R3;	R1=R2*R3;
	R2=R1>>3;

Type operations: fixed

- *sc_fixed<wl, iwl, q_mode, o_mode, n_bits>*
- *sc_ufixed<wl, iwl, q_mode, o_mode, n_bits>*
  - static fixed point types

Parameters

wl	**Total word length**
iwl	**Integer part word length**
q_mode	**Quantization mode:** Determines what happens if an operation generates mode precision in the LS-Bits
o_mode	**Overflow mode:** Determines what happens if an operation generates mode precision in the MS-Bits
n_bits	**Number of saturated bits:** Specifies number of bits that will be saturated in case of an overflow; default is 0

Type operations: fixed

- *sc_fixed<wl, iwl, q_mode, o_mode, n_bits>*
- *sc_ufixed<wl, iwl, q_mode, o_mode, n_bits>*
  - static fixed point types

Operations

Bitwise	~	&	|	^						
Arithmetic	+	-	*	/	%	>>	<<			
Assignment	=	+=	-=	*=	/=	&=	^=	|=	<<=	>>=
Equality	==	!=								
Relational	<	<=	>	>=						
Bit Select	[b]									
Part Select	range()									

Type operations: fixed

- *sc_fixed<wl, iwl, q_mode, o_mode, n_bits>*
- *sc_ufixed<wl, iwl, q_mode, o_mode, n_bits>*
  - q_mode: SC_RND, SC_RND_ZERO, SC_TRN, …
  - o_mode: SC_SAT, SC_SAT_ZERO, SC_SAT_SYM, …

Declaration	Example
sc_ufixed<13, 8> a;	**10011001.10011**
sc_ufixed<11, 8, SC_RND> b;	11110000.011 b=a; makes b equal to: 10011001.101
sc_ufixed<9, 6, SC_TRN, SC_SAT> c;	110101.011 c=a; makes c equal to: 111111.100

Round up

Set to max

Truncate

Type operations: fixed

- *sc_fix* var*(wl, iwl, q_mode, o_mode, n_bits)*
- *sc_ufix* var*(wl, iwl, q_mode, o_mode, n_bits)*
  - Non-static fix point types

Parameters

wl	**Total word length**
iwl	**Integer part word length**
q_mode	**Quantization mode:** Determines what happens if an operation generates mode precision in the LS-Bits
o_mode	**Overflow mode:** Determines what happens if an operation generates mode precision in the MS-Bits
n_bits	**Number of saturated bits:** Specifies number of bits that will be saturated in case of an overflow; default is 0

Can use variables to define parameters

Type operations: Conversions

- Conversion between types
  - the = sign is overloaded

Cannot do arithmetic with sc_bv or sc_lv

Declaration	Examples
sc_logic LB1, LB2;	
sc_bit BB1, BB2;	BB1=LB1; LB2=BB2;
sc_bv<16> R1, R2, R3;	
sc_uint<16> I1, I2, I3;	I1=R1; I2=R2; I3=I1+I2; R3=I3;
sc_lv<32> L1, L2, L3;	
sc_uint<32> U1, U2, U3;	U1=L1; U2=L2; U3=U1+U2; L3=U3;

Covert, add, convert back

Types & Operations: Overall

- Equality and bitwise operators can be used for all SystemC data types
- Arithmetic and relational operators can be used with the numeric types only
- The semantics of the equality operators, bitwise operators, arithmetic operators, and relational operators are the same in SystemC as in C++

SystemC Linguistics - Cont.

+ SystemC Execution
+ Basic Syntax & Semantics
– Language Semantics
  - Resolved Logic Vector
  - – Types & Operations
    - Basic Type
    - Type Operation
  - – Time, Events & Sensitivity Control
    - Time
    - Events
    - Sensitivity
      - Static Sensitivity
      - Dynamic Sensitivity

+ Interfaces
  Examples
  Summary

Time

- SystemC uses an integer-valued absolute time model
- Time is internally represented by an unsigned integer of at least 64-bits
- Time starts at 0, and moves forward only

Time: *sc_time*

- Class *sc_time* is special type which is used to represent simulation time and time intervals, including delays and time-outs
- An object of class *sc_time* is constructed from a *double* and an *sc_time_unit*
- Allows arithmetic operations

Time: *sc_time*

- Using *sc_time*
  - *sc_time* t_setup (4, SC_NS);
  - *sc_time* t_setup, t_hold, t_diff, t_short;
  - t_diff = t_setup – t_hold;
  - *sc_time* t_period (7, SC_NS);
  - if (t_period < t_short) {cout <<“too low”;}

Time: time units

- The default value for the default time unit is 1 *ns*
- Enum types that denote various time units are
  - SC_FS = 0
  - SC_PS = 1
  - SC_NS = 2
  - SC_US = 3
  - SC_MS = 4
  - SC_SEC= 5

Time: time resolution

- The time resolution is the smallest amount of time that can be represented by all *sc_time* objects in a SystemC simulation
- Time shall be represented internally as an integer multiple of the time resolution
- The default time resolution is 1 ps

Time: time resolution

- The time resolution can be changed by calling *sc_set_time_resolution* function
  - This function shall only be called during elaboration
  - shall not be called more than once
  - shall not be called after constructing an object of type *sc_time* with a non-zero time value
- User can ascertain the current time resolution by calling the *sc_get_time_resolution()* function

Time: time related methods

- *sc_simulation_time()*; Returns current simulation time as double
- *sc_start(run_time)*; Initialize simulation and advances time
- *sc_time_stamp()*; Returns current simulation time as sc_time
- *wait(time)*;
- *sc_get_default_time_unit()*; Get and set default time unit
- *sc_set_default_time_unit(value, sc_time_u)*;
- *sc_get_default_time_resolution()*; Get and set min time resolution
- *sc_set_default_time_resolution(value, sc_time_unit)*;

Events

- An event is an object of class *sc_event* used for process synchronization
- Determine whether and when a process execution should be triggered or resumed
- Any given event may be notified on many separate occasions
- Can be *hierarchically named* or not

Events

- Objects of class *sc_event* may be constructed during elaboration or simulation
- Event based requires events: declared by
  - sc_event
    - Syntax: sc_event ev1, ev2, ev3, ….;
  - sc_signal sig1;
    - Syntax: sig1->event();
- Can use & and | of various events

Events on signals

Events: *sc_event*

Class definition

sc_event.h

```
 85  class sc_event
 86  {
 87      //...
100  public:
101      sc_event();
102      ~sc_event();
103      void cancel();
104      void notify();
105      void notify( const sc_time& );
106      void notify( double, sc_time_unit );
107      void notify_delayed();
108      void notify_delayed( const sc_time& );
109      void notify_delayed( double, sc_time_unit );
110      //...
113  private:
114      void add_static( sc_method_handle ) const;
115      void add_static( sc_thread_handle ) const;
116      void add_dynamic( sc_method_handle ) const;
117      void add_dynamic( sc_thread_handle ) const;
118      void notify_internal( const sc_time& );
119      void notify_next_delta();
120      bool remove_static( sc_method_handle ) const;
121      bool remove_static( sc_thread_handle ) const;
122      bool remove_dynamic( sc_method_handle ) const;
123      bool remove_dynamic( sc_thread_handle ) const;
124      void reset();
125      void trigger();
126      //...
142  };
```

Events: event notifications

- Events can be notified in three ways
  - notify()
    - Processes sensitive to this event will run in current evaluation phase
  - notify(0)
    - Processes sensitive to this event will run in evaluation phase of next delta cycle
  - notify(t) with t>0
    - Processes sensitive to this event will run during the evaluation phase of some future simulator time

Immediate

Delta-Cycle Delayed

Timed

Events: canceling event notification

- A pending delayed event notification may be canceled using the *cancel()* method
- Immediate event notifications cannot be canceled, since their effect occurs immediately

Example

```
sc_event a, b, c;
sc_time t(10, SC_MS);
a.notify();                    // current delta-cycle
notify(SC_ZERO_TIME, b); // next delta-cycle
notify(t, c);                 // 10 ms delay
...
a.cancel(); // Error! Can't cancel immediate notification
b.cancel(); // cancel notification on event b
c.cancel(); // cancel notification on event c
```

Events: multiple event notification

- A given event shall have no more than one pending notification
- If function *notify* is called for an event that already has a notification pending, only the notification scheduled to occur at the earliest time shall survive

Events: multiple event notification

Example

```
sc_event e;
e.notify(SC_ZERO_TIME); // Delta notification
e.notify(1, SC_NS); // Timed notification ignored due to pending delta
                       notification
e.notify();         // Immediate notification cancels pending delta
                    notification. e is notified
e.notify(2, SC_NS); // Timed notification
e.notify(3, SC_NS); // Timed notification ignored due to earlier pending
                       timed notification
e.notify(1, SC_NS); // Timed notification cancels pending timed
                       notification
e.notify(SC_ZERO_TIME); // Delta notification cancels pending timed
                           notification. e is notified in the next delta
                           cycle
```

Events: methods

- *.posedge_event* returns a reference to an event that is notified whenever the value of the signal changes and the new value of the signal is true or '1'
- *.negedge_event* returns a reference to an event that is notified whenever the value of the signal changes and the new value of the signal is false or '0'
- *.posedge* returns the value true if and only if the value of the signal changes in the update phase of the immediately preceding delta cycle and the new value of the signal is true or '1'
- *.negedge* returns the value true if and only if the value of the signal changes in the update phase of the immediately preceding delta cycle and the new value of the signal is false or '0'

Sensitivity

- In constructor:

```
SC_METHOD (method_process);
//sensitivity list
SC_THREAD (thread_process);
//sensitivity list
```

- Common forms of sensitivity:

```
sensitive << clk_sig;
sensitive << clk_sig.posedge_event();
sensitive << clk_sig.negedge_event();
```

For signals

```
sensitive << clk;
sensitive << clk.pos();
sensitive << clk.neg();
```

```
sensitive_pos << clk;
sensitive_neg << clk;
```

For ports and signals

Sensitivity

- Signal edge detection (two-value logic)

```
SC_MODULE (module_name) {
    . . .
    sc_signal <bool> clk_sig;
    . . .
};
void module_name::some_thread(){
    while(1){
        if(clk_sig.posedge()) . . .;
        wait(clk_sig.negedge_event()) . . .;
    }
}
```

If clk changes from 0 to 1

Wait for a change making clk 0

Static Sensitivity

- Static sensitivity provides the parameters, which would trigger a process statically and is specified during design
- For defining static sensitivity lists
  - Functional Notation
    - Takes a single argument
    - Syntax: sensitive(event);
  - Streaming Style Notation
    - Supports multiple events
    - Syntax: sensitive << event_1 << event_2<< ….;
- Static sensitivity may be enabled or disabled by calling function *next_trigger* or function *wait*

Static Sensitivity

Example

Functional Notation

```
SC_MODULE (module_name) {
   sc_event c;
   sc_event d;
   void thread_proc_name();
   SC_CTOR(module_name) {
      SC_THREAD(thread_proc_name);
// declare static sensitivity list
      sensitive(c);
      sensitive(d);
   }
   ...
}
```

Streaming Style Notation

```
SC_MODULE (module_name) {
   sc_event c;
   sc_event d;
   void thread_proc_name();
   SC_CTOR(module_name) {
      SC_THREAD(thread_proc_name);
// declare static sensitivity list
      sensitive << c << d;
   }
   ...
}
```

Dynamic Sensitivity for SC_THREAD: *wait()*

- *wait()* can be called anywhere in the thread of execution of a thread process
- When the *wait()* method is called:
  - The calling thread process will suspend
  - When one of the events in the sensitivity list is notified
    - The waiting thread process is resumed
    - The static sensitivity of the calling thread process doesn't change

Dynamic Sensitivity for SC_THREAD: *wait()*

- Can use `wait();` for static sensitivity
- Can use `wait(...);` for dynamic sensitivity
  - Time based: `wait(time);`
  - Event based: `wait(event);`
  - Timeout: `wait(time, event);`

These define dynamic sensitivity

- Occurrence of a timeout can be checked by `timed_out()` boolean function
- Wait on events in `SC_THREAD` by:
  - `wait(4, SC_NS);` or `wait(t1);` wait the specified *time*
  - `wait(ev1 & ev2);` wait on *ev1* AND *ev2*
  - `wait(ev3 | ev4);` wait on *ev3* OR *ev4*
  - `wait(t1, ev3 | ev4);` wait on *ev3* OR *ev4* with *t1* timeout

Of course, these can be events on signals.

Dynamic Sensitivity for SC_THREAD: *wait()*

- Forms of wait():
  - *wait*();
    - Wait on events in sensitivity list
  - *wait*(*e1*);
    - Wait on event *e1*
  - *wait*(*e1* | … | *en*);
    - Wait on events *e1*, *e2*, … or *en*
  - *wait*(*delay*, SC_NS);
    - Wait for *delay* ns
  - *wait*(*delay*, SC_NS, *e1*);
    - Wait on event *e1*,
      time out after *delay* ns
  - *wait*(*cc*);
    - Wait *cc* clock cycles, SC_CTHREAD only
  - *wait*(*0*, SC_NS);
    - Wait one delta cycle
  - *wait*(SC_ZERO_TIME);
    - Wait one delta cycle

Dynamic Sensitivity for SC_METHOD: *next_trigger()*

- *next_trigger()* does not suspend the process
- Temporarily sets a sensitivity list only for next time the process executes again
- next_trigger() returns immediately, without passing control to another process
- Multiple next_trigger() calls allowed in one activation of an SC_METHOD process
  - The last next_trigger() call determines the sensitivity for the next activation

Dynamic Sensitivity for SC_METHOD: *next_trigger()*

- Can use `next_trigger();` to re-establish static sensitivity
- Can use `next_trigger(...);` for dynamic sensitivity
  - Time based: `next_trigger(time);`
  - Event based: `next_trigger (event);`
  - Timeout: `next_trigger (time, event);`

These define dynamic sensitivity

- Specify dynamic sensitivity in SC_METHOD by:
  - `next_trigger(4, SC_NS);` or `next_trigger (t1);` when it completes, it restarts after the specified time
  - `next_trigger(ev1 & ev2);` restarts with *ev1* AND *ev2*
  - `next_trigger(ev3 | ev4);` restarts with *ev1* AND *ev2*
  - `next_trigger(t1, ev3 | ev4);` restarts with *ev3* OR *ev4 unless* timed out

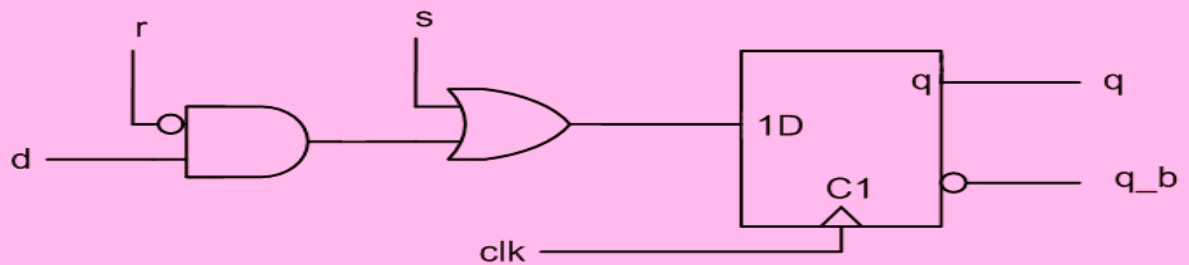
Example: Flip-flop

```
SC_MODULE (d_ff) {
   sc_in <bool> d, clk, s, r;
   sc_out <bool> q, q_b;

   SC_CTOR (d_ff) {
      SC_THREAD (dff_func);
      sensitive << clk.pos();
   }
   void dff_func ();
};
```

There are several alternatives…

- Need to insert delays
- Wait requires SC_THREAD
- Gets activated only once

```
void d_ff::dff_func ()
{
   while (1) {
      if (s.read()) {
         wait (4, SC_NS); q = true;
         wait (3, SC_NS); q_b = false;
      } else if (r.read()) {
         wait (4, SC_NS); q = false;
         wait (3, SC_NS); q_b = true;
      } else {
         wait (4, SC_NS); q = d.read();
         wait (3, SC_NS); q_b = !d.read();
      }
      wait (); // suspends the process until next positive edge of clk
   }
}
```



Example: Flip-flop- Alternative 1

```
SC_MODULE (d_ff) {
   sc_in <bool> d, clk;
   sc_out <bool> q;

   SC_CTOR (d_ff) {
      SC_METHOD (dff_func);
         sensitive << clk;
   }
   void dff_func ();
};
```

```
void d_ff::dff_func ()
{
   if (clk->event() && (clk=='1')) {
      q = d->read();
   }
}
```

Sensitive to both edges of clk

Then check if the edge was positive

```
void d_ff::dff_func ()
{
   if (clk.posedge()) {
      q = d.read();
   }
}
```

Example: Flip-flop- Alternative 2

```
SC_MODULE (d_ff) {
   sc_in <bool> d, clk;
   sc_out <bool> q;

   SC_CTOR (d_ff) {
      SC_METHOD (dff_func);
         sensitive << clk.pos;
   }
   void dff_func ();
};
```

Sensitive to positive edges of clk

Then just assign d to q

```
void d_ff::dff_func ()
{
   q = d;
}
```

Example: Flip-flop- Alternative 3

```
SC_MODULE (d_ff) {
   sc_in <bool> d, clk;
   sc_out <bool> q;

   SC_CTOR (d_ff) {
      SC_THREAD (dff_func);
         sensitive << clk.pos();
   }
   void dff_func ();
};
```

Sensitive to positive edges of clk

Wait for another positive edge

```
void d_ff::dff_func ()
{
   while (1) {
      q = d;
      wait();
   }
}
```

Example: Flip-flop- Alternative 4

```
SC_MODULE (d_ff) {
   sc_in <bool> d, clk;
   sc_out <bool> q;

   SC_CTOR (d_ff) {
      SC_THREAD (dff_func);
         sensitive << clk.pos();
         dont_initialize();
   }
   void dff_func ();
};
```

Sensitive to positive edges of clk

Don't run the THREAD the first time

Wait for another on the sensitivity list

```
void d_ff::dff_func ()
{
   while (1) {
      q = d;
      wait();
   }
}
```

Example: Flip-flop- Alternative 5

```
SC_MODULE (d_ff) {
    sc_in <bool> d, clk;
    sc_out <bool> q;

    SC_CTOR (d_ff) {
        SC_THREAD (dff_func);
    }
    void dff_func ();
};
```

Sensitive to positive edges of clk

Wait for positive edge event

```
void d_ff::dff_func ()
{
    while (1) {
        q = d;
        wait(clk.posedge_event());
    }
}
```

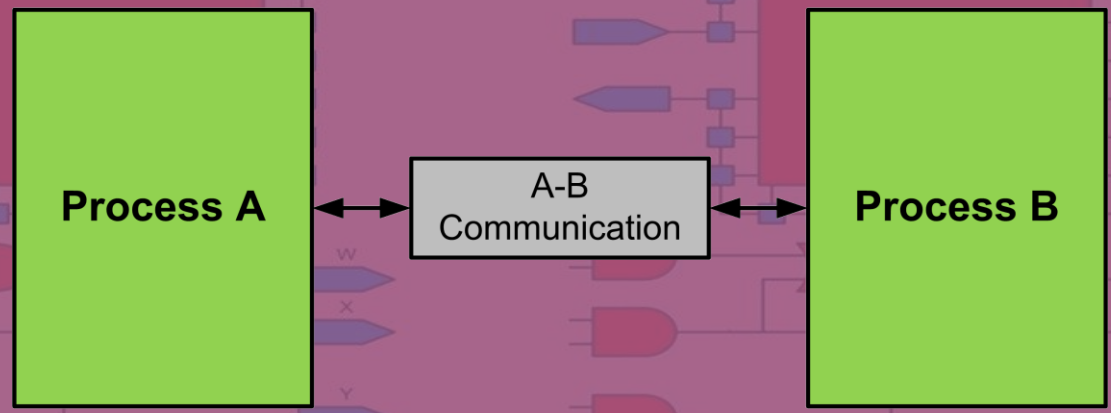
SystemC Linguistics

+ SystemC Execution
+ Basic Syntax & Semantics
+ Language Semantics
– Interfaces
  - Interfaces
  – Channels
    - Primitive channels
    - Hierarchical channels
    - Predefined Channels
      - sc_signal
      - sc_mutex
      - sc_event_queue
  - Ports

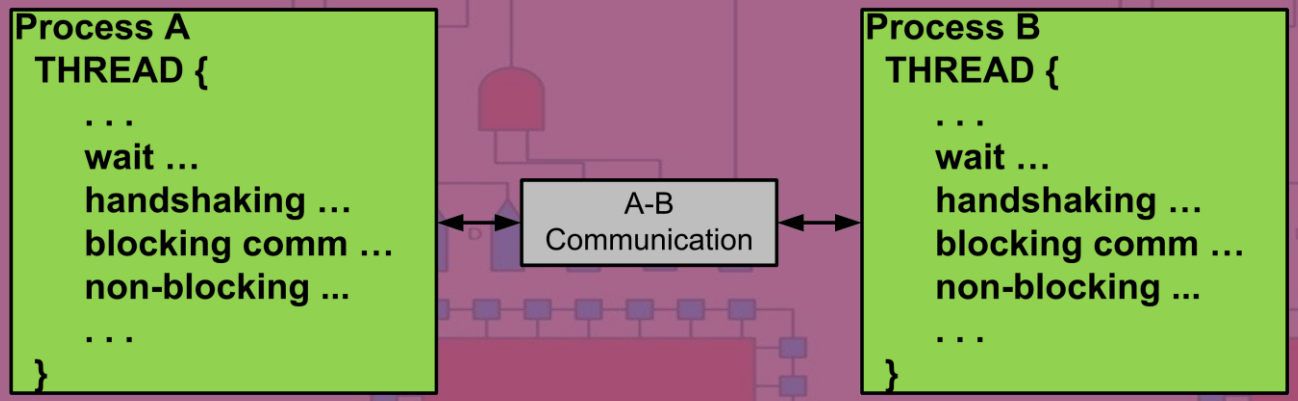
Examples
Summary

Interfaces

- Communication is defined by two thins:
  - Interfaces
  - Implementation of those interfaces in the channel
- Processes communicate without having to worry about type of communication
  - A simple wire
  - A bus
  - A bus with handshaking
  - Burst or block data transfer
- Simply perform
  - ABCommunication.read()
  - ABCommunication.write()



Interfaces



- Handle handshaking independent of processes
- Perform blocking and non-blocking communication
- Embed handshaking signals inside the communications
- Embed data size inside communications

Interfaces

C++ Interfaces

- SystemC interfaces are based on C++ interface classes
- Interfaces are a good example of polymorphism in C++
- They define a set of pure virtual methods, but do not implement these methods
- Are used to declare sets of methods that are required to be implemented by a derived class

Interfaces: Example

```
16  class registerInterface { //This is an interface class
17      public:
            registerInterface () {};
            ~registerInterface () {};
20          virtual void clk ()=0;
21          virtual void set (char* val)=0;
22  };
```

Pure abstract class

Pure virtual methods

Polymorphism Example

```
24  class dReg : public registerInterface {
25      char* d;
26      char* q;
27      public:
28          dReg (char* D="0", char* Q="0");
29          ~dReg ();
30          virtual void clk ();
31          void set (char* val);
32  };
33
34  class dRegE : public dReg { //Enable
35      char* e;
36      public:
37          dRegE (char* D, char* E, char* Q);
38          ~dRegE ();
39          virtual void clk ();
40  };
```

Inheritance from registerInterface

Implementing pure virtual methods

```
42  class uCnt : public registerInterface {
        char* q;
        public:
            uCnt (char* Q="0");
            ~uCnt ();
47          virtual void clk ();
48          void set (char* val);
    };

    class uCntE : public uCnt {
        char* e;
        public:
            uCntE (char* E, char* Q);
            ~uCntE ();
56          virtual void clk ();
57  };
```

Interfaces: Example

```
11  int main ()
12  {   int ij;
13      cout << "Starting . . . \n";
14      vector (DataIn, 8); vector (RegOut, 8);
15
16      line(e); line(g); line(l);
17      vector(DataOut, 8);
18      vector (zero, 8); bval(0,zero);
19
20      registerInterface* REG = new dReg (DataIn, RegOut);
21      registerInterface* CNT = new uCntE (e, DataOut);
22
23      load(REG, 6);
24      load(CNT, 7);
25
26      nBitComparator* CMP=new nBitComparator (DataIn, RegOut, l, e, g);
27
28      cout << "Initial count value is: " << DataOut << "\n";
29
30      do{
31          cout << "Enter 8 bit for DataIn: "; cin >> DataIn;
32          CMP->evl();
33          if (*(e)=='0') load(CNT, 0);
34          REG->clk();
35          CNT->clk();
36          cout << "Consequtive data: " << ival (DataOut) << "\n";
37          cout << "\n" << "Continue (0 or 1)?"; cin >> ij;
38      } while (ij >0);
39  }
```

```
4  void load(registerInterface* funReg, const int value)
5  {
6      char* bcnt=new char [8]; *(bcnt+8)='\0';
7      bval(value, bcnt);
8      funReg->set(bcnt);
9  }
```

Instantiation

Use *load* for dReg and uCntE.

Polymorphism Example

Interfaces: Example

Partial.cpp file of the example

```
62  dReg::dReg (char* di, char* qo) {
63      int i=0;
64      this->d = di;
65      this->q = qo;
66      for (i=0;i<7;i++){
67          *(q+i)='X';
68      }
69  }
70  dReg::~dReg() {delete d; delete q;}
71  void dReg::clk(){
72      vcopy(q, d);
73  }
74  void dReg::set(char* val){
75      vcopy(q, val);
76  }
77
78  dRegE::dRegE (char* di, char* en, char* qo) :dReg (di,qo) {
79      this->e = en;
80  }
81  dRegE::~dRegE() {delete e;}
82  void dRegE::clk(){
83      if (*(e+0)=='1') dReg::clk();
84  }
```

```
86  uCnt::uCnt (char* qo) {
87      int i=0;
88      this->q = qo;
89      for (i=0;i<7;i++){
90          *(q+i)='X';
91      }
92  }
93  uCnt::~uCnt() {delete q;}
94  void uCnt::clk(){
95      int icnt;
96      char* bcnt=new char [8]; *(bcnt+8)='\0';
97
98      icnt=ival(q)+1;
99      bval(icnt, bcnt);
100     vcopy(q,bcnt);
101 }
102 void uCnt::set(char* val){
103     vcopy(q, val);
104 }
105
106 uCntE::uCntE(char*en, char* qo):uCnt(qo) {
107     this->e = en;
108 }
109 uCntE::~uCntE() {delete e;}
110 void uCntE::clk(){
111     if (*(e+0)=='1') uCnt::clk();
112 }
```

Polymorphism Example

Interfaces

- SystemC Interfaces
  - All interfaces are directly or indirectly derived from the base class *sc_interface*
  - We will have SystemC interfaces that are based on *sc_interface*, and provide virtual interfaces
  - Interfaces specify a set of access methods to the channels
  - We use channels for communications that are based on SystemC interfaces
  - SystemC channels reference pure virtual method declarations of interfaces
  - A SystemC channel class implements one or more SystemC interfaces

Interfaces: Base Class

sc_interface.h

(Global Scope)

```
45  class sc_interface
    {
    public:
        // register a port with this interface (does nothing by default)
        virtual void register_port( sc_port_base& port_,
                    const char*    if_typename_ );
        // get the default event
        virtual const sc_event& default_event() const;
        // destructor (does nothing)
        virtual ~sc_interface();
    protected:
        // constructor (does nothing)
        sc_interface();
    private:
        // disabled
        sc_interface( const sc_interface& );
        sc_interface& operator = ( const sc_interface& );
    private:
        static sc_event m_never_notified;
    };
```

sc_interface.h

Abstract class for interfaces

Public Member Functions: Two virtual methods do nothing by default. Can be defined by channels

Default constructor

Disabled member functions: Copy constructor, Default assignment operator

sc_interface class definition

Interfaces

- There are a number of interfaces provided by SystemC
  - sc_fifo_in_if
  - sc_fifo_out_if
  - sc_mutex_if
  - sc_semaphore_if
  - sc_signal_in_if
  - sc_signal_inout_if

Interface

⊙ *sc_signal_in_if*

sc_signal_ifs.h

(Global Scope)

```
54  template <class T>
55  class sc_signal_in_if
56  : virtual public sc_interface
57  {
58  public:
59      // get the value changed event
60      virtual const sc_event& value_changed_event() const = 0;
61      // read the current value
62      virtual const T& read() const = 0;
63      // get a reference to the current value (for tracing)
64      virtual const T& get_data_ref() const = 0;
65      // was there a value changed event?
66      virtual bool event() const = 0;
67  protected:
68      // constructor
69      sc_signal_in_if(){}
70  private:
71      // disabled
72      sc_signal_in_if( const sc_signal_in_if<T>& );
73      sc_signal_in_if<T>& operator = ( const sc_signal_in_if<T>& );
74  };
```

sc_signal_ifs.h

Template class

Abstract class inherited from *sc_interface*

Public Member Functions: Pure virtual methods must be defined by channels

Create a *sc_signal_in_if* instance

Disabled member functions: Copy constructor, Default assignment operator

gives read access to the value of a signal

Interfaces

⊙ *sc_signal_inout_if*

sc_signal_ifs.h

(Global Scope)

```
231  template <class T>
     class sc_signal_inout_if
     : public sc_signal_in_if<T>, public sc_signal_write_if<T>
234  {
235  protected:
236      // constructor
237      sc_signal_inout_if(){}
238  private:
239      // disabled
240      sc_signal_inout_if( const sc_signal_inout_if<T>& );
241      sc_signal_inout_if<T>& operator = ( const sc_signal_inout_if<T>& );
242  };
```

sc_signal_ifs.h

Template class

Abstract class derived from a further interface for write access

Create a *sc_signal_in_if* instance

Disabled member functions:
Copy constructor,
Default assignment operator

gives both read and write access to the value of a signal

SystemC Linguistics

+ SystemC Execution
+ Basic Syntax & Semantics
+ Language Semantics
– Interfaces
  - Interfaces
  - – Channels
    - Primitive channels
    - Hierarchical channels
    - Predefined Channels
      - sc_signal
      - sc_mutex
      - sc_event_queue
  - Ports

Examples
Summary

Channels

- A container class for communication and synchronization
- They implement one or more *interfaces*
- Different channels may implement the same interface in different ways
- Channels provide the communication between
  - modules or within a module
  - between processes
- A channel implements all the methods of the inherited interface classes
- There are *primitive channels* and *hierarchical channels*

Channels

- Primitive Channels
  - have no SystemC structures
  - shall implement one or more interfaces
  - support the request-update method of access
  - are derived from the base class called *sc_prim_channel()*
  - *sc_prim_channel()* provides two methods for implementation of the request-update scheme
    - *request_update()* is a non-virtual function which can be called during the evaluate phase of a delta-cycle
    - *update()* is a virtual function that must be specified by the derived channel as its behavior is dependent upon the derived channel's functionality

Channels

- Primitive Channels

```
class sc_prim_channel
{
   protected:
   sc_prim_channel( const char* name = 0 ); // constructor with name
   const char* name() const;  // get the name
   void request_update(); // request_update() method to be executed
                          during the update step
   virtual void update() {} // the update method (does nothing by default)
   virtual ~sc_prim_channel() {} // destructor (does nothing by default)
}
```

Channels

- Hierarchical Channels
  - have SystemC structures
  - shall implement one or more interfaces
  - structures may include ports, instances of modules, other channels, and processes (greater flexibility)
  - are derived from the base class called *sc_module()*

Channels

- Predefined Channels
  - SystemC contains several predefined primitive channels:
    - *sc_signal*
    - *sc_mutex*
    - *sc_event_queue*
    - *sc_semaphore*
    - *sc_fifo*
    - …

Channels

⊙ Predefined Channels:

sc_signal

A primitive channel that implements the *sc_signal_inout_if* interface

Create a *sc_signal* instance with the string name initialized to name_

A *sc_signal* may be written by only one process, but may be read by multiple processes

sc_signal writes and reads follows evaluate-update semantics suitable for describing hardware

Disabled member functions: Copy constructor

sc_signal.h

Primitive channels are derived from *sc_prim_channel*() base class

Checks to ensure at most only one out or inout port is connected to the *sc_signal* instance

Add *sc_signal* objects to the VCD file

sc_signal.h

sc_core

```
72  template <class T>
73  class sc_signal
74      : public sc_signal_inout_if<T>,
75      public sc_prim_channel
    {
    public:
        // constructors, destructor
        sc_signal();
        explicit sc_signal(const char* name_);
        virtual ~sc_signal();

        // methods
        virtual void register_port(sc_port_base&, const
            char*);
        virtual const sc_event& default_event() const;
        virtual const sc_event& value_changed_event() const;
        virtual const T& read() const;
        virtual const T& get_data_ref() const;
        virtual bool event() const;
        virtual void write(const T&);
        operator const T& () const;
        sc_signal<T>& operator = (const T& a);
        sc_signal<T>& operator = (const sc_signal<T>& a);
        const T& get_new_value() const;
        void trace(sc_trace_file* tf) const;
        virtual void print(ostream&) const;
        virtual void dump(ostream&) const;
        static const char* const kind_string;
        virtual const char* kind() const;
    protected:
        virtual void update();
        void check_writer();
    private:
        // disabled
        sc_signal(const sc_signal<T>&);
    };
```

Channels

- Predefined Channels: *sc_signal*

Example

```
SC_MODULE (bCircuit) {
   sc_in <bool> a, b;
   sc_out <int> w;
   …
   sc_signal <bool> d;
   sc_signal <int> e;
   sc_signal <sc_lv <8> > dv;
   …
};
```

sc_signal_in_if is an interface class for the sc_signal class, it is derived from *sc_interface*

sc_signal is a primitive channel

sc_signal has a template parameter for type

Channels

- Predefined Channels: *sc_signal*
  - sc_signal methods are
    - write(..): write value
    - read(): read value
    - event(): was there an event (in bool)
    - default_event(): get the event
  - For signals of type bool and sc_logic
    - posedge_event(): get the event
    - negedge_event()
    - posedge(): was there an event (in bool)
    - negedge()
    - delayed(): get the delayed signal

Channels

- Predefined Channels: *sc_mutex*
  - A mutex is an object used to let multiple program threads share a common resource without colliding
  - Any process that needs the resource must *lock()* the mutex waits until lock occurs
  - The process *unlock()* the mutex when done
  - Using *trylock()* allows a process to get the mutex if availble (non-blocking)
  - There is no event that tells when an *sc_mutex* is freed

Channels

⊙ Predefined Channels: *sc_mutex*

sc_mutex.h

```
(Global Scope)
46  class sc_mutex
47  : public sc_mutex_if,
48    public sc_prim_channel
    {
50  public:
51      // constructors and destructor
52      sc_mutex();
53      explicit sc_mutex( const char* name_ );
54      virtual ~sc_mutex();
55      // interface methods
56      // blocks until mutex could be locked
57      virtual int lock();
58      // returns -1 if mutex could not be locked
59      virtual int trylock();
60      // returns -1 if mutex was not locked by caller
61      virtual int unlock();
62      virtual const char* kind() const;
63  protected:
64      // support methods
65      bool in_use() const;
66  private:
67      // disabled
68      sc_mutex( const sc_mutex& );
69      sc_mutex& operator = ( const sc_mutex& );
70  };
```

implements the *sc_mutex_if* interface

Primitive channels are derived from *sc_prim_channel*() base class

A process may lock the *mutex*. Only the process that locked the *mutex* may unlock it

If multiple processes attempt to lock an unlocked *mutex* during the same delta-cycle, only one will be successful. The unsuccessful processes will be suspended

Disabled member functions:
Copy constructor
Default assignment operator

A *sc_signal* may be written by only one process, but may be read by multiple processes

is used for a mutual-exclusion lock for access to a shared resource

Channels

⊙ Predefined Channels: *sc_mutex*

Abstract class inherited from *sc_interface*

Implemented by the *sc_mutex* channel

Public Member Functions: Pure virtual methods must be defined by the *sc_mutex* channel

Create a *sc_mutex_if* instance

Disabled member functions: Copy constructor, Default assignment operator

sc_mutex_if.h

sc_mutex_if.h | sc_mutex.h

(Global Scope)

```
43  class sc_mutex_if
44  : virtual public sc_interface
45  {
46  public:
47      virtual int lock() = 0;
48      virtual int trylock() = 0;
49      virtual int unlock() = 0;
50  protected:
51      // constructor
52      sc_mutex_if(){}
53  private:
54      // disabled
55      sc_mutex_if( const sc_mutex_if& );
56      sc_mutex_if& operator = ( const sc_mutex_if& );
57  };
```

sc_mutex interface

Channels

- Predefined Channels: *sc_mutex*
  - sc_mutex methods are
    - virtual *int lock();*
      - If the mutex is unlocked
        - Member function *lock()* shall lock the mutex and return
      - If the mutex is locked
        - Suspend until the mutex is unlocked (by another process)
        - Implements a blocking process
      - Member function *lock()* shall unconditionally return the value 0
    - …

Implements a blocking process

Channels: Cont.

- *virtual int trylock();*
  - If the mutex is unlocked
    - Member function *trylock()* shall lock the mutex
    - Shall return the value **0**
  - If the mutex is locked
    - Member function *trylock()* shall immediately return the value -1
    - The mutex shall remian locked
- *virtual const char* kind() const;*
  - Member fucntion *kind()* shall return the string "sc_mutex"
- If multiple processes attempt to lock the mutex in the same delta cycle
  - the process instance that is given the lock in that delta cycle is "non-deterministic"
  - relies on the order in which processes are resumed within the evaluation phase

Implements a non-blocking process

Channels: Cont.

- *virtual int unlock();*
  - If mutex unlocked,
    - member function *unlock* shall return the value **-1**
    - the mutex shall remain unlocked
  - If mutex locked by another process
    - member function *unlock* shall return the value **-1**
    - the mutex shall remain locked
  - If mutex locked by the calling process
    - member function *unlock* shall unlock the mutex
    - shall return the value **0**
- *virtual int unlock();*
  - If several processes are suspended and waiting for the mutex
    - unlock allows only one process to unlock (nondeterministic)
    - remaining processes shall suspend again

Channels

- Predefined Channels: sc_event_queue
  - Class *sc_event_queue* represents an event queue
  - Like class *sc_event*, an event queue has a member function *notify*
  - Unlike an *sc_event*, an event queue is a hierarchical channel and can have multiple notifications pending
  - *sc_event_queue* objects can only be constructed during elaboration

Channels

Predefined Channels: sc_event_queue

Example

```
sc_event_queue EQ;

SC_CTOR(Mod){
    SC_THREAD(T);
    SC_METHOD(M);
    sensitive << EQ;
    dont_initialize();
}
void T(){
    EQ.notify(2, SC_NS); // M runs at time 2ns
    EQ.notify(1, SC_NS); // M runs at time 1ns, 1st or 2nd delta cycle
    EQ.notify(SC_ZERO_TIME); // M runs at time 0ns
    EQ.notify(1, SC_NS); // M runs at time 1ns, 2nd or 1st delta cycle
}
```

Channels

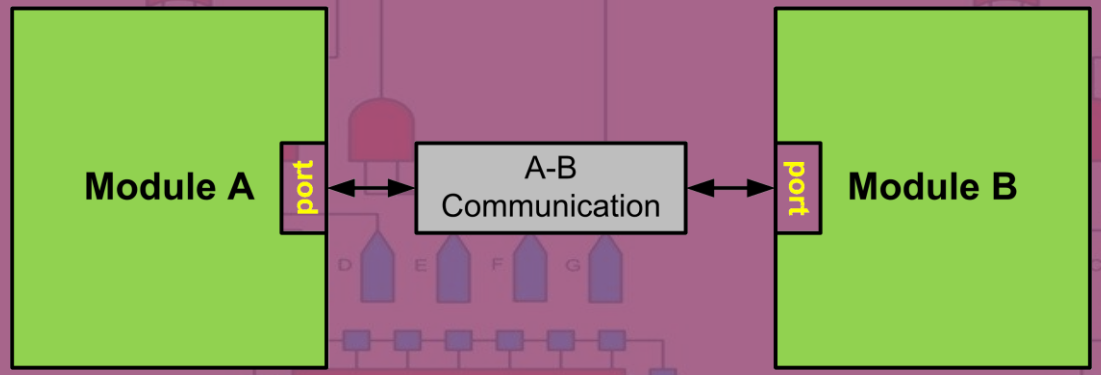
- Predefined Channels: sc_event_queue
  - sc_event_queue methods are
    - virtual void *notify*(double , sc_time_unit);
    - virtual void *notify*(const sc_time&);
    - virtual void *cancel_all*();
    - virtual const sc_event& *default_event*() const;

SystemC Linguistics

+ SystemC Execution
+ Basic Syntax & Semantics
+ Language Semantics
– Interfaces
  - Interfaces
  - Channels
    - Primitive channels
    - Hierarchical channels
    - Predefined Channels
      - sc_signal
      - sc_mutex
      - sc_event_queue
  - Ports

Examples
Summary

Ports



- Ports allow modules to access channels, i,e., ease module communications
- Modules can also communicate through external variables, pointer passing, or signals
- Ports are specialized to use specific types of interfaces
- A ports is a class that is inherited from an interface
- It is also templated with an interface (like a VHDL mode)
- Ports allow channel access across modules

Ports: Types

- Basic port types:
  - *sc_in<T>*, *sc_out<T>*, and *sc_inout<T>*
  - They are all derived from the base class *sc_port*
  - Each provides a set of interface methods, such as *read()* and *write()*
    - These ports call the corresponding interface method of the attached channel

```
sc_in <bool> p;
```

Ports: *sc_port*

sc_port.h

```
{} sc_core
373  template <class IF, int N = 1>
374  class sc_port
         : public sc_port_b<IF>
376  {
377      // typdefs
378      typedef sc_port_b<IF> base_type;
         typedef sc_port<IF, N> this_type;
380  public:
381      // constructors, destructor
382      sc_port();
383      explicit sc_port(const char* name_);
384      explicit sc_port(IF& interface_);
385      sc_port(const char* name_, IF& interface_);
386      explicit sc_port(base_type& parent_);
387      sc_port(const char* name_, base_type& parent_);
388      sc_port(this_type& parent_);
389      sc_port(const char* name_, this_type& parent_);
390      virtual ~sc_port();
391      static const char* const kind_string;
392      virtual const char* kind() const;
393  private:
394      // disabled
395      sc_port(const this_type&);
         this_type& operator = (const this_type&);
397  };
```

An *sc_port* instance is associated with an interface of type IF

N signifies the maximum number of interfaces that may be attached to the port

Various forms of constructors

Disabled member functions:
Copy constructor,
Default assignment operator

A port may not be bound after elaboration

Ports

sc_signal_ports.h

sc_core::sc_inout<T>

```
75  template <class T>
76  class sc_in
77      : public sc_port<sc_signal_in_if<T>, 1>
78  {
79  public:
80      // constructors and destructor
81      sc_in();
82      sc_in(const char* name_);
83      sc_in(const sc_signal_in_if<T>& interface_);
84      // ...
85      virtual ~sc_in();
86      // methods
87      void bind(const sc_signal_in_if<T>& interface_);
88      void operator () (const
89          sc_signal_in_if<T>& interface_);
90      void bind(sc_port< sc_signal_in_if<T> >& parent_);
91      sc_event_finder& value_changed() const;
92      void operator () (
93          sc_port< sc_signal_in_if<T> >& parent_);
94      // ...
95      const sc_event& default_event() const;
96      const sc_event& value_changed_event() const;
97      const T& read() const;
98      void add_trace(sc_trace_file*,
99          const sc_string&) const;
100 };
```

sc_signal_ports.h

Port for use with *sc_signal* channels that is specialized with *sc_signal_in_if<T>* interface

Various forms of constructors

For port to channel binding

For port to port binding

Returns a reference to an event that occurs when new_value on a write is different from current_value

Returns a reference to current_value

Add *sc_in* objects to the VCD file

sc_in
class definition

Ports

sc_signal_ports.h

sc_core::sc_inout<T>

```
102  template <class T>
103  class sc_out
104      : public sc_inout<T>
105  {
106  public:
107      // typedefs
108      typedef T data_type;
109      typedef sc_out<data_type>this_type;
110      typedef sc_inout<data_type> base_type;
111      typedef typename base_type::in_if_type in_if_type;
112      typedef typename base_type::in_port_type in_port_type;
113      typedef typename base_type::inout_if_type inout_if_type;
114      typedef typename base_type::inout_port_type
115          inout_port_type;
116  public:
117      // constructors & destructor
118      sc_out();
119      explicit sc_out(const char* name_);
120      explicit sc_out(inout_if_type& interface_);
121      // ...
122      virtual ~sc_out();
123      this_type& operator = (const data_type& value_);
124      // ...
125  private:
126      // disabled
127      sc_out(const this_type&);
128  };
```

Port for use with *sc_signal* channels that is derived with *sc_inout<T>* port

Has the same functionality as an *sc_inout* port

typedef to use further down

Various forms of constructors

Disabled member functions: Copy constructor

sc_signal_ports.h

sc_out class definition

Ports

Port for use with *sc_signal* channels that is specialized with *sc_signal_inout_if<T>* interface

Various forms of constructors

Various forms of methods

sc_signal_ports.h

sc_signal_ports.h

sc_core::sc_in<T>

```
132  template <class T>
     class sc_inout
134      : public sc_port<sc_signal_inout_if<T>, 1>
135  {
136  public:
137      // constructors and destructor
138      sc_inout();
139      sc_inout(const char* name_);
140      // ...
141      virtual ~sc_inout();
142      // methods
143      const sc_event& default_event() const;
144      const sc_event& value_changed_event() const;
145      const T& read() const;
146      operator const T& () const;
147      bool event() const;
148      sc_inout<T>& write(const T& value_);
149      sc_inout<T>& operator = (const T& value_);
150      // ...
151      void add_trace(sc_trace_file*,
152          const sc_string&) const;
153  };
```

sc_inout class definition

Ports

- Ports enable a module, and hence its processes, to access a channel's interface

The port base class

```
sc_port <interface <type>, N > p;
//N = number of channels that can be connected to the port
```

```
sc_in <bool> p;
```

sc_in is a specialized port for the sc_signal class

sc_port<sc_signal_in_if<bool>,1> p;

SystemC Linguistics

+ SystemC Execution
+ Basic Syntax & Semantics
+ Language Semantics
+ Interfaces
  Examples
  Summary

Examples

sc_fifo_ifs.h | sc_fifo.h | sc_fifo_ports.h

(Global Scope)

```
1  template <class T>
2  class sc_fifo_in_if
3      : virtual public sc_interface
4  {
5  public:
6      virtual void read(T&) = 0;
7      virtual T read() = 0;
8      virtual bool nb_read(T&) = 0;
9      virtual int num_available() const = 0;
10     virtual const sc_event&
11         data_written_event() const = 0;
12 private:
13     // disabled
14     sc_fifo_in_if(const sc_fifo_in_if<T>&);
15     sc_fifo_in_if<T>&
16         operator = (const sc_fifo_in_if<T>&);
17 };
```

sc_fifo_ifs.h

All interfaces are inherited from *sc_interface*

Implemented by the *sc_fifo* channel

Public Member Functions: Pure virtual methods must be defined by the *sc_fifo* channel

Disabled member functions: Copy constructor, Default assignment operator

FIFO interfaces definition

Examples

sc_fifo_ifs.h | sc_fifo.h | sc_fifo_ports.h

(Global Scope)

```
19  template <class T>
20  class sc_fifo_out_if
21      : virtual public sc_interface
22  {
23  public:
24      virtual void write(const T&) = 0;
25      virtual bool nb_write(const T&) = 0;
26      virtual int num_free() const = 0;
27      virtual const sc_event& data_read_event() const = 0;
28  private:
29      // disabled
30      sc_fifo_out_if(const sc_fifo_out_if<T>&);
31      sc_fifo_out_if<T>& operator =
32          (const sc_fifo_out_if<T>&);
33  };
```

sc_fifo_ifs.h

All interfaces are inherited from *sc_interface*

Implemented by the *sc_fifo* channel

Public Member Functions:
Pure virtual methods must be defined by the *sc_fifo* channel

Disabled member functions:
Copy constructor,
Default assignment operator

FIFO interfaces definition

Examples

sc_fifo_ifs.h | sc_fifo.h | sc_fifo_ports.h

{} sc_core

```
48  template <class T>
49  class sc_fifo
51      : public sc_fifo_in_if<T>,
52      public sc_fifo_out_if<T>,
        public sc_prim_channel
    {
54  public:
55      // constructors and destructor
56      explicit sc_fifo(int size_ = 16);
57      explicit sc_fifo(const char* name_, int size_ = 16);
58      virtual ~sc_fifo();
59      // interface methods
60      virtual void read(T&);
61      virtual T read();
        virtual bool nb_read(T&);
63      virtual int num_available() const;
64      virtual const sc_event& data_written_event() const;
65      virtual void write(const T&);
66      virtual bool nb_write(const T&);
67      virtual int num_free() const;
68      virtual const sc_event& data_read_event() const;
69  protected:
70      virtual void update();
71  private:
72      // disabled
73      sc_fifo(const sc_fifo<T>&);
74      sc_fifo& operator = (const sc_fifo<T>&);
75  };
```

A primitive channel that implements the *sc_fifo_in_if* and *sc_fifo_out_if* interfaces

Primitive channels are derived from *sc_prim_channel*() base class

Returns the number of elements that are currently in the FIFO

Returns the number of free spaces currently in the FIFO

FIFO Channel definition

sc_fifo.h

Size refers to the maximum number of entries the FIFO may have

Size may be explicitly set. Otherwise, default value is 16

writes and reads follow the evaluate-update semantics

Blocking and non-blocking reads and writes

Disabled member functions: Copy constructor, Default assignment operator

Examples

Port for use with *sc_fifo* channels that is specialized with *sc_fifo_in_if<T>* interface

sc_fifo_ports.h

```
sc_fifo_ifs.h   sc_fifo.h   sc_fifo_ports.h
sc_fifo_out<T>
299  class sc_fifo_in
         : public sc_port<sc_fifo_in_if<T>, 0>
301  {
302  public:
303      // constructors and destructor
304      sc_fifo_in();
305      sc_fifo_in(const char* name_);
306      sc_fifo_in(sc_fifo_in_if<T>& interface_);
307      sc_fifo_in(const char* name_,
308          sc_fifo_in_if<T>& interface_);
309      sc_fifo_in(sc_port_b
310          <sc_fifo_in_if<T> >& parent_);
311      sc_fifo_in(const char* name_,
312          sc_port_b<sc_fifo_in_if<T> >& parent_);
313      sc_fifo_in(sc_fifo_in<T>& parent_);
314      sc_fifo_in(const char* name_,
315          sc_fifo_in<T>& parent_);
316      virtual ~sc_fifo_in();
317      // methods
318      void read(T& value_);
319      T read();
320      bool nb_read(T& value_);
321      int num_available() const;
322      const sc_event& data_written_event() const;
323      sc_event_finder& data_written() const;
324      static const char* const kind_string;
325      virtual const char* kind() const
326  private:
327      // disabled
328      sc_fifo_in(const sc_fifo_in<T>&);
329      sc_fifo_in<T>& operator = (const sc_fifo_in<T>&);
330  };
```

Create a *sc_fifo_in* instance

methods for convenience in accessing the FIFO channel connected to the port

Disabled member functions:
Copy constructor,
Default assignment operator

FIFO ports definition

Examples

sc_fifo_ifs.h | sc_fifo.h | sc_fifo_ports.h

sc_fifo_in<T>

sc_fifo_ports.h

```
337  template <class T>
338  class sc_fifo_out
339      : public sc_port<sc_fifo_out_if<T>, 0>
340  {
341  public:
342      // constructors and destructor
343      sc_fifo_out();
344      sc_fifo_out(const char* name_);
345      sc_fifo_out(sc_fifo_out_if<T>& interface_);
346      sc_fifo_out(const char* name_,
347          sc_fifo_out_if<T>& interface_);
348      sc_fifo_out(sc_port_b
349          <sc_fifo_out_if<T> >& parent_);
350      sc_fifo_out(const char* name_,
351          sc_port_b<sc_fifo_out_if<T> >& parent_);
352      sc_fifo_out(sc_fifo_out<T>& parent_);
353      sc_fifo_out(const char* name_,
354          sc_fifo_out<T>& parent_);
355      virtual ~sc_fifo_out();
356      // methods
357      void write(const T& value_);
358      bool nb_write(const T& value_);
359      int num_free() const;
360      const sc_event& data_read_event() const;
361      sc_event_finder& data_read() const;
362      static const char* const kind_string;
363      virtual const char* kind() const;
364  private:
365      // disabled
366      sc_fifo_out(const sc_fifo_out<T>&);
367      sc_fifo_out<T>& operator = (const sc_fifo_out<T>&);
368  };
```

Port for use with *sc_fifo* channels that is specialized with *sc_fifo_out_if<T>* interface

Create a *sc_fifo_out* instance

methods for convenience in accessing the FIFO channel connected to the port

Disabled member functions:
Copy constructor,
Default assignment operator

FIFO ports definition

SystemC Linguistics

+ SystemC Execution
+ Basic Syntax & Semantics
+ Language Semantics
+ Interfaces
+ Examples
- Summary

Summary

- SystemC Execution
- Basic Syntax & Semantics
- Language Semantics
- Interfaces
- Examples