Analysis, Design and Modeling in SystemC

SystemC Capabilities

- **Modern electronic systems consist of hardware and software.**
  - **Each has become very complex.**
  - **Their interaction (tradeoffs to meet customers' needs) has become increasingly complex.**
- **SystemC supports hardware-software co-design and the description of the architecture of complex systems consisting of both hardware and software components.**
- **It supports the description of**
  - **hardware,**
  - **software, and**
  - **interfaces**

in a C++ environment.

Hardware Software Co-design

- **In complex systems, software developers must often wait for the hardware design to be finalized before they can begin detailed coding.**
- **Software developers must also wait for devices (ICs and printed circuit boards) to be manufactured to test their code in a realistic environment.**
  - **This dependency creates a long critical path that may add so much financial risk to a project that it is never started.**

What is SystemC

- SystemC is a C++ class library and a methodology that you can use to effectively create a model of
  - software algorithms,
  - hardware architecture, and
  - interfaces of your SoC (System On a Chip) and system-level designs.

- The SystemC Class Library has been developed by a group of companies forming the Open SystemC Initiative (OSCI).

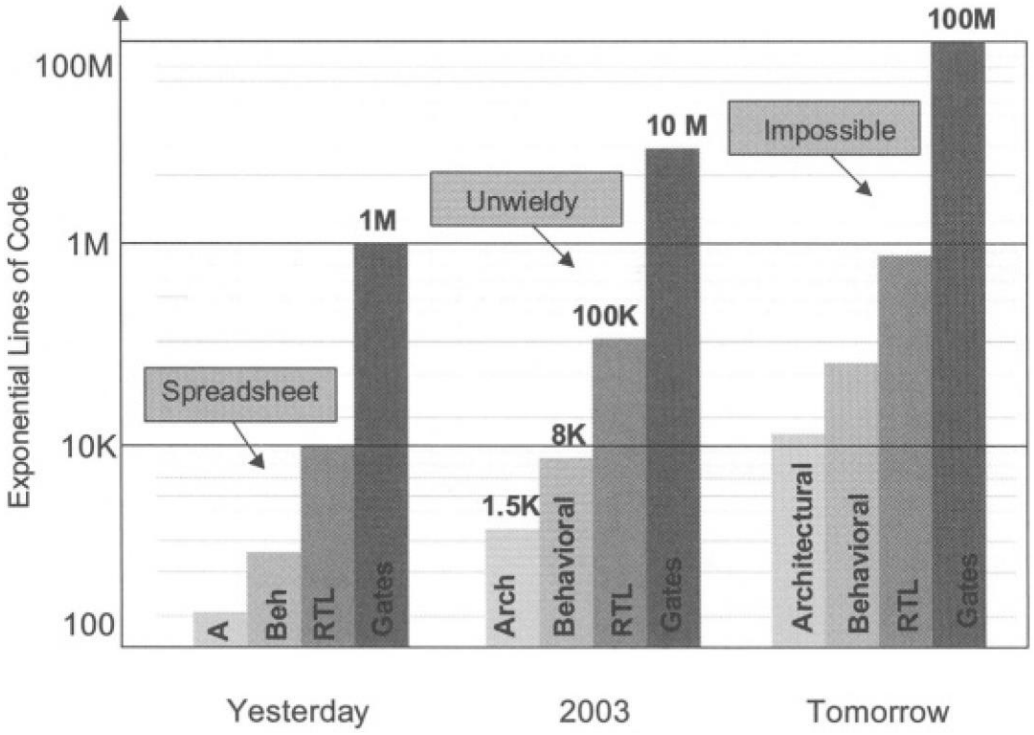
- You can use SystemC and standard C++ development tools to
  - create a system-level model,
  - quickly simulate to validate and optimize the design,
  - explore various algorithms, and
  - provide the hardware and software development team with an executable specification of the system.
    - An executable specification is essentially a C++ program that exhibits the same behavior as the system when executed.

Benefits of Executable Specification

- Avoids inconsistency and errors and helps ensure completeness of the specification.
  - because you are essentially creating a program that behaves the same way as the system.
- Ensures unambiguous interpretation
  - Whenever implementers are in doubt about the design, they can run the executable specification to determine what the system is supposed to be doing.
- Helps validate system functionality before implementation begins.
- Helps create early performance models of the system and validate system performance.
- Its testbench can be refined (or used as is) to test the implementation of the specification.
  - This can provide tremendous benefits to implementers and drastically reduce the time for implementation verification.

System Complexity

- Three sample designs from three generations
  - **Tomorrows systems are being designed today.**



Levels of Abstraction

Level of Abstraction	Verilog	VHDL	C/C++
System Level	Not Suitable	Poor	Very Good
High Level (Behavioral)	Good	Very Good	Good
Medium Level (RTL)	Very Good	Very Good	Poor
Low Level (Gates)	Good	Poor	Not Suitable

- **Raising LOA is necessary for managing complexity.**

SystemC History

- The first stage, release 1.0 (presently at version 1.0.2) provides all the necessary modeling facilities to describe systems similar to those which can be described using a hardware description language, such as VHDL.
- Version 1.0 provides a simulation kernel, data types appropriate for fixed point arithmetic, communication channels which behave like pieces of wire (signals), and modules to break down a design into smaller parts.
- In Release 2.0 (presently at version 2.0.1), the class library has been extensively re-written to provide an upgrade path into true system level design.
- Features that were “built-in” to version 1.0, such as signals, are now built upon an underlying structure of channels, interfaces, and ports.
- August 2004: version 2.1 was introduced (major clean up, some enhancement).
- **On December 12, 2005, the IEEE approved the IEEE 1666 standard for SystemC.**
- In future, Version 3.0 of the class library will be extended to cover modeling of operating systems, to support the development of models of embedded software.

SystemC Features

- **Modules:**
  - **A container class called a module: a hierarchical entity that can have other modules or processes contained in it.**
- **Processes:**
  - **are used to describe functionality.**
  - **are contained inside modules.**
- **Ports:**
  - **Modules have ports through which they connect to other modules.**
  - **Single-direction and bidirectional ports.**
- **Signals:**
  - **SystemC supports resolved and unresolved signals.**

SystemC Features

- **Rich set of port and signal types:**
  - **To support modeling at different levels of abstraction, from the functional to the RTL.**
    - **This is different than languages like Verilog that only support specific types as port and signal types.**
- **Rich set of data types:**
  - **to support multiple design domains and abstraction levels.**
    - **The fixed precision data types allow for fast simulation,**
    - **the arbitrary precision types can be used for computations with large numbers, and**
    - **the fixed-point data types can be used for DSP applications.**

SystemC Features

- **Clocks:**
  - **SystemC has the notion of clocks (as special signals).**
  - **Multiple clocks, with arbitrary phase relationship, are supported.**

- **Cycle-based simulation:**
  - **SystemC includes an ultra light-weight cycle-based simulation kernel that allows high-speed simulation.**

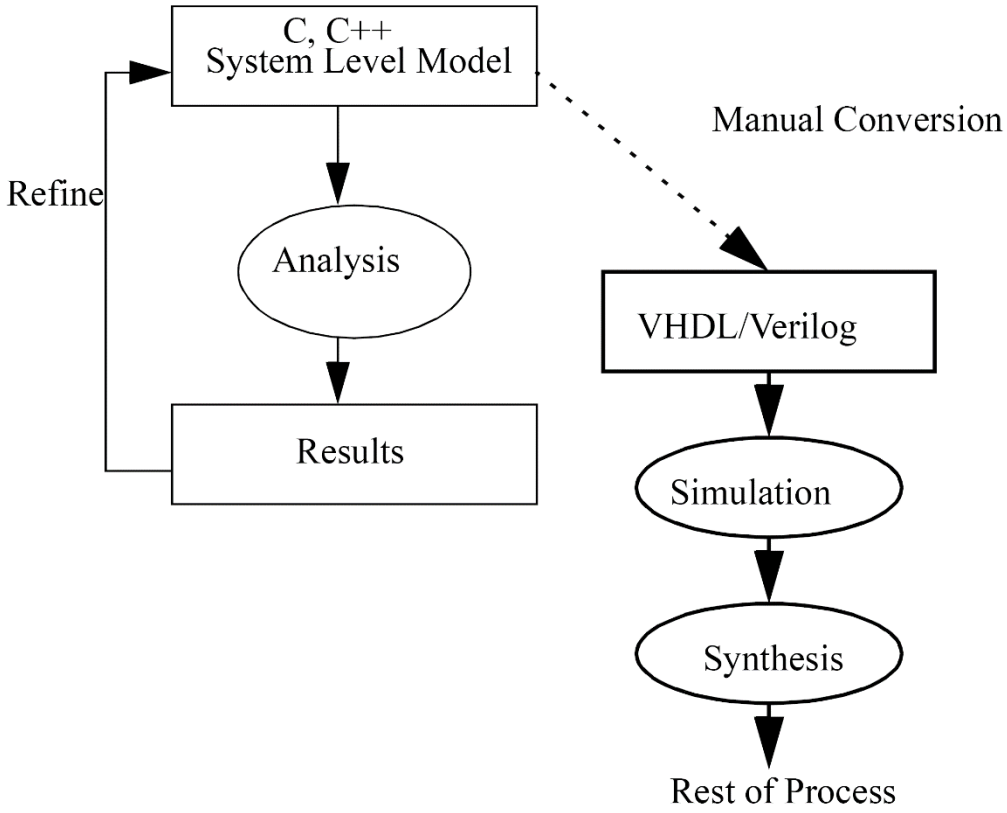
SystemC Features

- **Multiple abstraction levels:**
  - **SystemC supports untimed models at different levels of abstraction,**
    - **ranging from high-level functional models to detailed clock cycle accurate RTL models.**
- **Communication protocols:**
  - **SystemC provides multi-level communication semantics that enable you to describe SoC and system I/O protocols with different levels for abstraction.**
- **Waveform tracing:**
  - **SystemC supports tracing of waveforms in VCD, WIF, and ISDB formats.**

Current System Design Methodology

1. **A system engineer writes a C or C++ model of the system to verify the concepts and algorithms at the system level.**

2. **The parts of the C/C++ model to be implemented in hardware are manually converted to a VHDL or Verilog description for actual hardware implementation.**

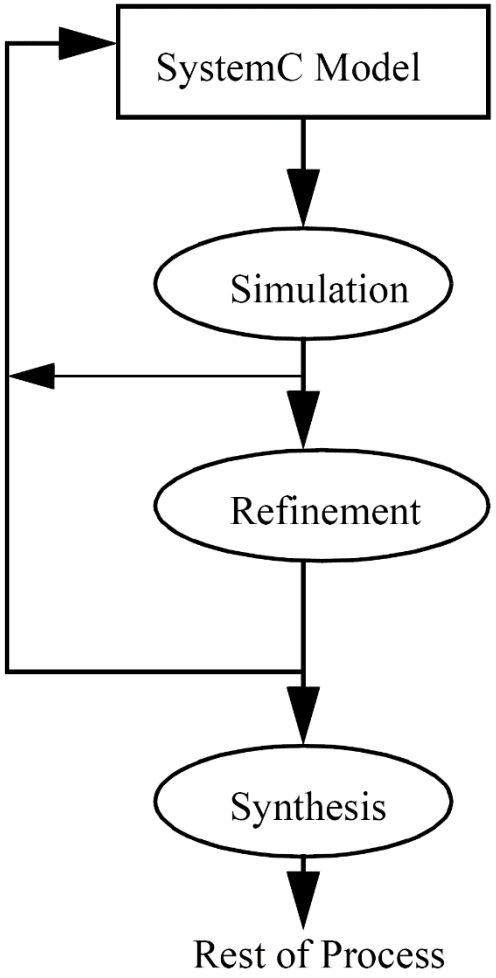


Problems with the Current Methodology

- **Manual Conversion from C to HDL Creates Errors**
  - **The designers:**
  1. creates the C model,
  2. verifies that the C model works as expected, and
  3. translates the design manually into an HDL.
  - ➢ **This is very tedious and error prone.**
- **Disconnect Between System Model and HDL Model**
  - ➢ After conversion, the HDL model becomes the focus of development. The C model quickly becomes out of date as changes are made.
    - Typically changes are made only to the HDL model and not implemented in the C model.
- **Multiple System Tests**
  - ➢ Tests that are created to validate the C model functionality typically cannot be run against the HDL model without conversion.
    - Not only does the designer have to convert the C model to HDL, but the test suite has to be converted to the HDL environment as well.

SystemC System Design Methodology: Advantages

- **Refinement Methodology:**
  - The design is not converted from a C level description to an HDL in one large effort.
  - The design is slowly refined in small sections to add the necessary hardware and timing constructs to produce a good design.
    - Using this refinement methodology, the designer can more easily implement design changes and detect bugs during refinement.



SystemC System Design Methodology: Advantages

- **Written in a Single Language**
  - ➢ the designer does not have to be an expert in multiple languages.
    - SystemC allows modeling from the system level to RTL.
- **Higher Productivity**
  - ➢ because the designer can model at a higher level.
    - → smaller code, that is easier to write and
    - → simulates faster than traditional modeling environments
- **Testbenches can be reused from the system level model to the RTL model**
  - ➢ Saving conversion time.
  - ➢ gives the designer a higher confidence that the system level and the RTL model implement the same functionality.

Why Not Just Use C/C++?

- ***Concurrency***.
  - How do we invoke this counter when the clock changes from 0 to 1 so that it acts like a *hardware* counter?

```
int counter (int clk)
{
    static int countval = 0;


    if (clk == 1) {
       if (countval == 255)
            countval = 0;
       else
            countval = countval + 1;
    }

    return countval;

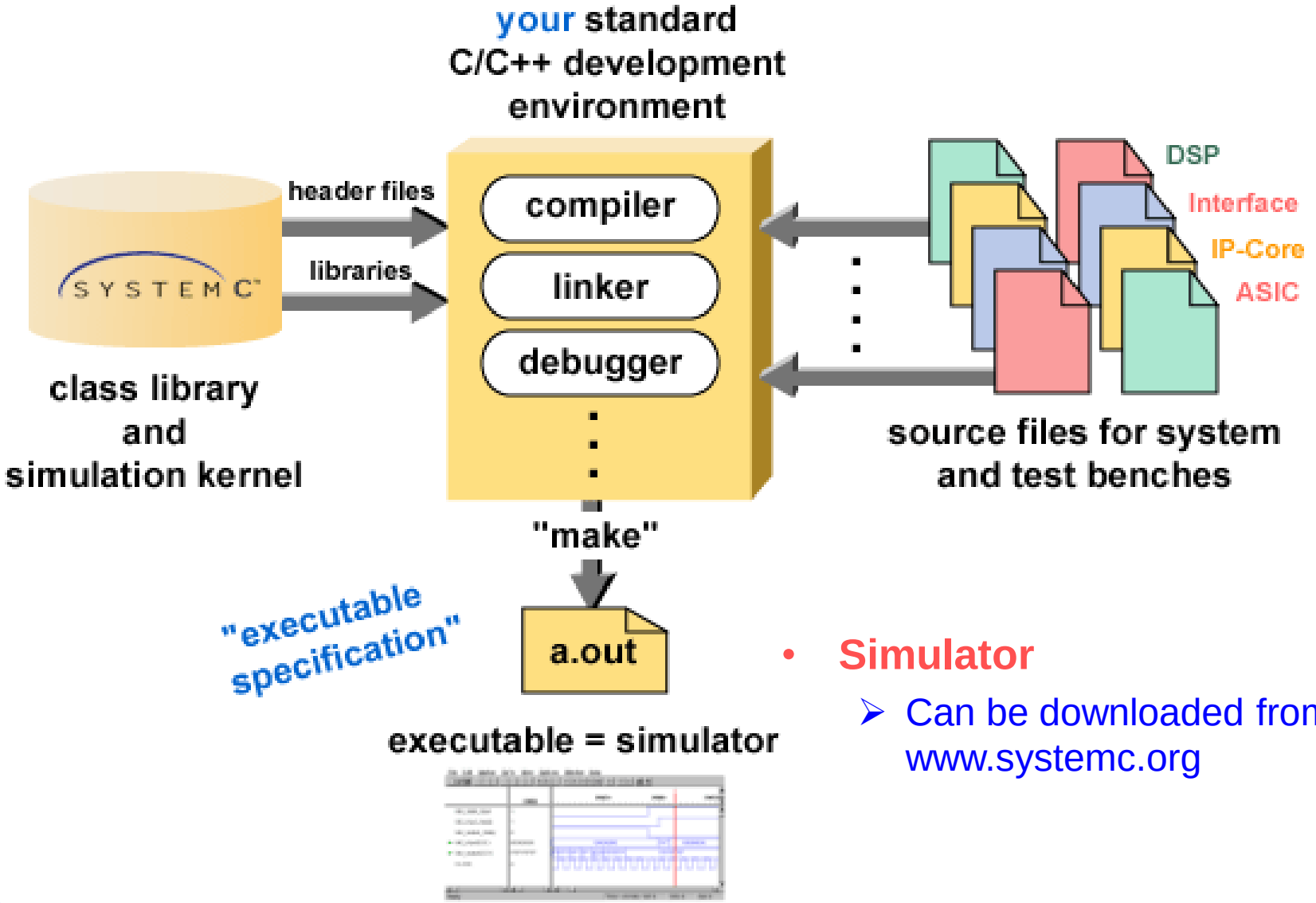
}
```

SystemC Adds to C++

- **The SystemC Class Library:**
  - provides the necessary constructs to model system architectures, including hardware timing, concurrency, and reactive behaviors that are missing in standard C++.

- **The C++ object-oriented programming language:**
  - provides the ability to extend the language through classes, without adding new syntactic constructs.
  - SystemC provides these necessary classes and allows designers to continue to use the familiar C++ language and development tools.

SystemC Development Environment



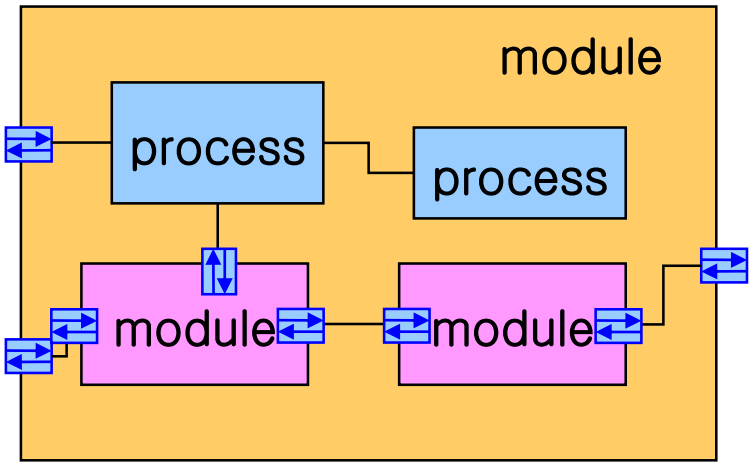
- **Simulator**
  - Can be downloaded from www.systemc.org

Modules and Hierarchy

- **Modules**
  - the basic building block within SystemC to partition a design.
    - Modules allow designers to hide internal data representation and algorithms from other modules.
- **Declaration**
  - Using the macro SC_MODULE
    - **SC_MODULE(***modulename***) {**
  - Using typical C++ struct or class declaration:
    - **struct** *modulename* **: sc_module {**

Modules and Hierarchy

- **Elements**:
  - ports,
  - local signals,
  - local data,
  - other modules,
  - processes, and
  - constructors.



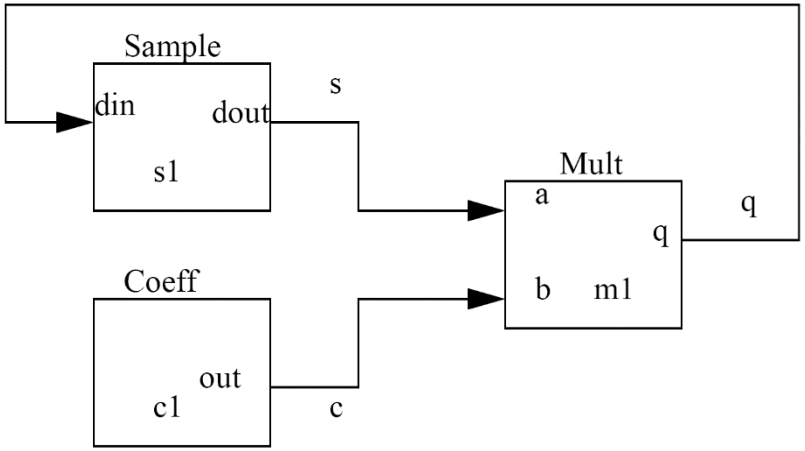
Ports

```
SC_MODULE(fifo) {
   sc_in<bool> load;
   sc_in<bool> read;
   sc_inout<int> data;
   sc_out<bool> full;
   sc_out<bool> empty;
//rest of module not shown
}
```

Signals

```
sc_signal<type > q, s, c;
```

- **Positional Connection**
- **Named Connection**



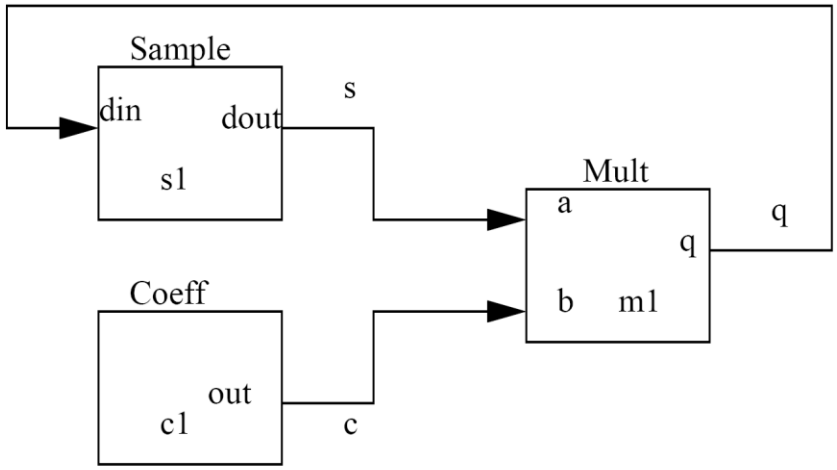
Named Connection

- `Instancename.portname(signalname);`

```
SC_MODULE(filter) {
   sample *s1;
   coeff *c1;
   mult *m1;

   sc_signal<sc_uint<32> > q, s, c;

   SC_CTOR(filter) {
     s1 = new sample ("s1");
     s1->din(q);
     s1->dout(s);
     c1 = new coeff ("c1");
     c1->out(c);
     m1 = new mult ("m1");
     m1->a(s);
     m1->b(c);
     m1->q(q);
   }
}
```



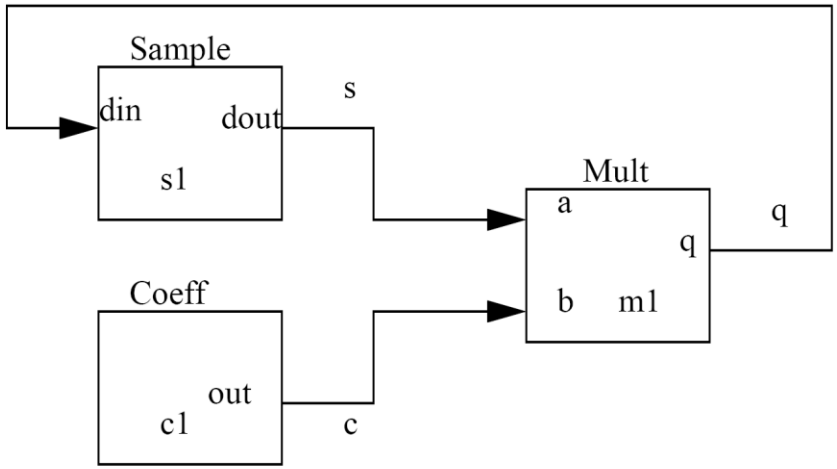
Positional Connection

- `Instancename (sig1, sig2, …);`

```
SC_MODULE(filter) {
   sample *s1;
   coeff *c1;
   mult *m1;

   sc_signal<sc_uint<32> > q, s, c;

   SC_CTOR(filter) {
     s1 = new sample ("s1");
     (*s1)(q,s);
     c1 = new coeff ("c1");
     (*c1)(c);
     m1 = new mult ("m1");
     (*m1)(s,c,q);
   }
}
```



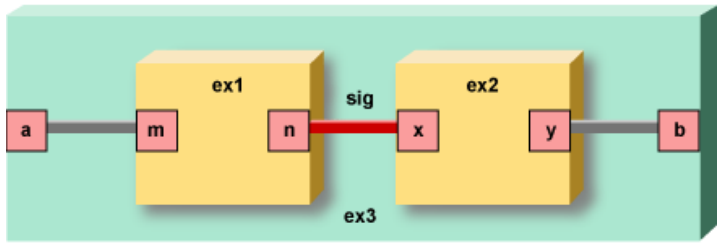
Instantiation: Another Style

```
// file ex1.h
SC_MODULE(ex1) {
    sc_port<sc_fifo_in_if<int> > m;
    sc_port<sc_fifo_out_if<int> > n;

    SC_CTOR(ex1) {
    }
// Rest of the module body is not shown
};
```

```
// file ex2.h
SC_MODULE(ex2) {
    sc_port<sc_fifo_in_if<int> > x;
    sc_port<sc_fifo_out_if<int> > y;

    SC_CTOR(ex2) {
    }
// Rest of the module body is not shown
};
```




```
SC_MODULE(ex3){
    sc_port<sc_fifo_in_if<int> >a;
    sc_port<sc_fifo_out_if<int> > b;
    sc_fifo<int> sig1;
// Instances of ex1 and ex2
    ex1 ex1_instance;
    ex2 ex2_instance;
// Module Constructor
    SC_CTOR(ex3):
          ex1_instance("ex1_instance"),//init'n
          ex2_instance("ex2_instance") //init'n
    {
          // Named connection for ex1
          ex1_instance.m(a);
          ex1_instance.n(sig1);
          // Positional connection for ex2
          ex2_instance(sig1, b);
// Rest of constructor body not shown
    }
};
```

`ex1_instance("ex1_instance"):`
- passes the instance label to the constructors of the instance.

sc_main()

- The top level is a special function called `sc_main`.
  - It is in a file named main.cpp or main.cc (standard practice - not a requirement).
- `sc_main()` is called by SystemC and is the entry point for your code.
- The execution of `sc_main()` until the `sc_start()` function is called (described later) is considered to be the elaboration time of SystemC.

- **Syntax:**

```
int sc_main (int argc, char *argv [ ] ) {
// body of function
return 0 ;
}
```

Instantiation in `sc_main()`

- **Two Steps:**
  - Declaration and Initialization.

```
module_name instance_name ("string_name") ;
```

    - It is recommended to keep the string_name the same as the instance_name.
  - Module Instantialtion (Port Binding).
    - **Named Connection:**

```
instance_name.port_name(channel_or_port);
```

    - **Positional Connection:**

```
instance_name(channel_or_port, channel_or_port, . . .);
```

Instantiation in `sc_main()`: Examples

```
int sc_main(int argc, char *argv[ ])
// Create fifos with a depth of 10
    sc_fifo<int> s1(10);
    sc_fifo<int> s2(10);
    sc_fifo<int> s3(10);

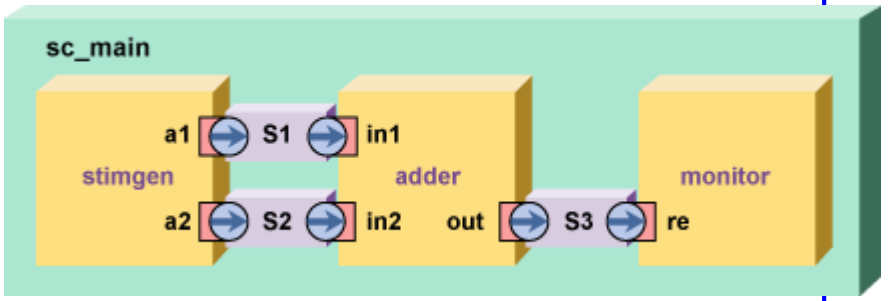
    // Module instantiations
    // Stimulus Generator
    stimgen stim("stim");
    stim(s1, s2);

    // Adder
    adder add("add");
    add(s1, s2, s3);

    // Response Monitor
    monitor mon ("mon");
    mon.re(s3);

    sc_start();

    return 0;
}
```



sc_start()

- At the bottom of the `sc_main()` function and before the return statement is the `sc_start()` function.
- Execution of this statement marks the end of elaboration and the start of simulation.
- `sc_start(arg)` has an optional argument:
  - specifies the number of time units to simulate.
  - If it is a null argument the simulation will run forever.

Processes

- Processes
  - **The real work of the modules are performed in processes.**
  - **Processes are functions that**
    - **are called whenever signals these processes are "sensitive to" change value.**
    - **executed sequentially.**
    - **may contain calls to a function named `wait()` that will halt execution of the process.**
- **Types of Processes:**
  - method processes,
  - thread processes.

Sensitivity

```
SC_MODULE(my_module) {
    sc_event c, d;
    void proc_1();
    void proc_2();
    void proc_3();
    SC_CTOR(my_module) {
       SC_THREAD(proc_1);
        sensitive << c << d; //proc_1 sensitive to c & d
       SC_THREAD(proc_2); // no static sensitivity
       SC_THREAD(proc_3);
        sensitive << d ; // proc_3 sensitive to d
    }
    // rest of module not shown
};
```

Sensitivity

- the pattern for multiple processes:
  - declaration
  - sensitivity list
  - declaration
  - sensitivity list
  - ….

```
SC_MODULE(dff) {
  …
  void doit();
  SC_CTOR(dff) {
    SC_METHOD(doit);
    sensitive_pos << clock << sig1 << sig2;
  }
};
```

```
SC_MODULE(dff) {
  …
  void doit();
  SC_CTOR(dff) {
    SC_METHOD(doit);
    sensitive_pos (clock);
    sensitive_pos (sig1);
    sensitive_pos (sig2);

  }
};
```

Sensitivity

```
SC_MODULE(dff) {
  …
  void doit();
  SC_CTOR(dff) {
    SC_METHOD(doit);
    sensitive_pos << clock << sig1 << sig2;
  }
};
```

```
SC_MODULE(dff) {
  …
  void doit();
  SC_CTOR(dff) {
    SC_METHOD(doit);
    sensitive_pos (clock);
    sensitive_pos (sig1);
    sensitive_pos (sig2);

  }
};
```

Processes (Example)

```
// dff.h
#include "systemc.h"

SC_MODULE(dff) {
  sc_in<bool> din;
  sc_in<bool> clock;
  sc_out<bool> dout;
  void doit();
  SC_CTOR(dff) {
    SC_METHOD(doit);
    sensitive_pos << clock;
  }
};

// dff.cc
#include "dff.h"
void dff::doit() {
  dout = din;
}
```

- **SC_CTOR:**
  - constructor,
    - creates and initializes an instance of a module.
- **SC_METHOD,**
  - Makes the member function **`doit()`** a **process** (sensitive to positive edge of clock)
- **"systemc.h" :**
  - interface to the SystemC library,
  - must be included in any file that contains references to SystemC functions.

Constructor (Example)

```
// ram.h
#include "systemc.h"
SC_MODULE(ram) {
  sc_in<int> addr;
  sc_in<int> datain;
  sc_in<bool> rwb;
  sc_out<int> dout;
  int memdata[64]; // local memory storage
  int i;
  void ramread();
  void ramwrite();
  SC_CTOR(ram){
    SC_METHOD(ramread);
    sensitive << addr << rwb;
    SC_METHOD(ramwrite);
    sensitive << addr << datain << rwb;
    for (i=0; i++; i<64) {
      memdata[i] = 0;
    }
  }
};
// rest of module not shown
```

Constructor (Example)

- When a RAM module is instantiated
  1. the constructor is called,
  2. data is allocated for the module,
  3. the two processes are registered with the SystemC kernel,
  4. finally the for loop is executed
     - initialize all the memory locations of the newly created ram module.

Processes

- **Parallelism in hardware:**
  - SystemC has the concept of processes to model the parallel activities of a system.
- **Three types:**
  - Methods
  - Threads
  - (Clocked Threads):
    - A special case of thread (about to be obsolete).

Processes: Basics

- **SC_METHOD:**
  - simply a member function of an **SC_MODULE** class where time does not pass between the invocation and return of the function.
  - the simulator kernel repeatedly calls it.
- **SC_THREAD:**
  - is only invoked *once.*
  - has the option to *suspend* itself and potentially allow time to pass before continuing.
- **SC_CTHREAD:**
  - a thread process that has the requirement of being sensitive to a clock.
- **Processes are not hierarchical:**
  - no process will call another process directly.
    - Processes can call methods and functions that are not processes.
- **Sensitivity lists:**
  - Processes cause other processes to execute by assigning new values to signals in the sensitivity list of the other process.

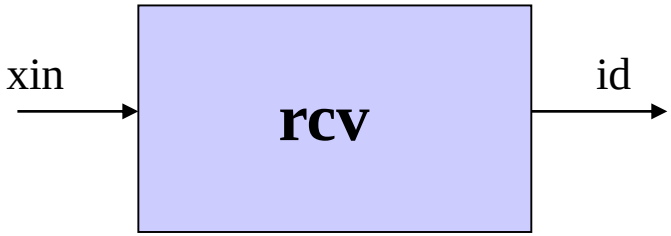
Method Process

- When events occur on signals of the sensitivity list, the process executes.
- The method executes and returns control back to the simulation kernel.

```
// rcv.h
#include "systemc.h"
#include "frame.h"
SC_MODULE(rcv) {
  sc_in<frame_type> xin;
  sc_out<int> id;

  void extract_id();

  SC_CTOR(rcv) {
    SC_METHOD(extract_id);
    sensitive(xin);
  }
};`
```



Method Process

- When input xin changes, method extract_id is invoked which assigns a value to port id.

```
// rcv.cc

#include "rcv.h"
#include "frame.h"

void rcv::extract_id() {
  frame_type frame;
  frame = xin;
  if(frame.type == 1) {
    id = frame.ida;
  } else {
    id = frame.idb;
  }
}
```

Method Process

- **When a method process is invoked, it executes until it returns.**
  - Users are strongly recommended not to write infinite loops within a method process as control will never be returned back to the simulator.
- **Local variables are redefined each time the process is invoked.**
  - If you wish to save the state information then it must be done in member variables
- **Cannot use wait() in a METHOD process**
  - Any attempt to wait a method process results in a runtime error.
- **A Method process is run during initialization phase.**
  - If you want not to run at that time, use `dont_initialize();` after defining sensitivity list.

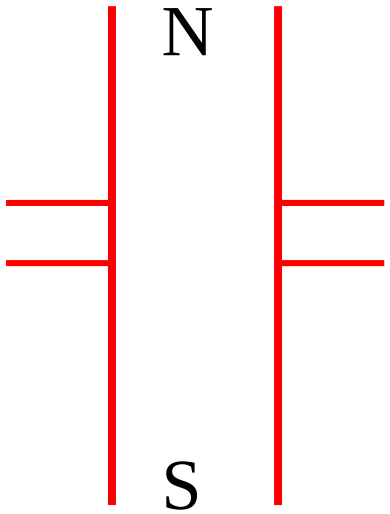
Thread Process

- Thread Processes are started once and only once by the simulator.
  - Once it is returned, it is gone forever
  - → **SC_THREAD**s typically contain an infinite loop containing at least one **wait.**
- Thread Process can be suspended and reactivated.
- The Thread Process can contain wait() functions
  - Wait() suspends process execution until an event occurs on one of the signals the process is sensitive to.
- An event will reactivate the thread process from the statement the process was last suspended.
  - The process will continue to execute until the next wait().

Thread Process: Traffic Light Controller

- **Highway**
  - Normally has a green light.
- **Sensor:**
  - A car on the East-West side road triggers the sensor
    - The highway light: green => yellow => red,
    - Side road light: red => green.
- **SystemC Model:**
  - Uses two different time delays:
    - green to yellow delay >= yellow to red delay

    (to represent the way that a real traffic light works).



Thread Process: Example

```
// traff.h
#include "systemc.h"

SC_MODULE(traff) {

  // input ports
  sc_in<bool> roadsensor;
  sc_in<bool> clock;

  // output ports
  sc_out<bool> NSred;
  sc_out<bool> NSyellow;
  sc_out<bool> NSgreen;
  sc_out<bool> EWred;
  sc_out<bool> EWyellow;
  sc_out<bool> EWgreen;
  void control_lights();
  int i;
```

```
  // Constructor
  SC_CTOR(traff) {
    SC_THREAD(control_lights); // Thread
      sensitive << roadsensor;
      sensitive_pos << clock;
  }
};
```

Thread Process: Example

```
// traff.cc
#include "traff.h"
void traff::control_lights() {
  NSred = false;
  NSyellow = false;
  NSgreen = true;
  EWred = true;
  EWyellow = false;
  EWgreen = false;
  while (true) {
    while (roadsensor == false)
        wait();
  NSgreen = false; // road sensor triggered
  NSyellow = true; // set NS to yellow
  NSred = false;
  for (i=0; i<5; i++)
    wait();
  NSgreen = false; // yellow interval over
  NSyellow = false; // set NS to red
  NSred = true; // set EW to green
  EWgreen = true;
  EWyellow = false;
  EWred = false;
  for (i= 0; i<50; i++)
    wait();
```

```
    NSgreen = false; // times up for EW green
    NSyellow = false; // set EW to yellow
    NSred = true;
    EWgreen = false;
    EWyellow = true;
    EWred = false;
    for (i=0; i<5; i++) // times up for EW yellow
      wait();
    NSgreen = true; // set EW to red
    NSyellow = false; // set NS to green
    NSred = false;
    EWgreen = false;
    EWyellow = false;
    EWred = true;
    for (i=0; i<50; i++) // wait one more long
      wait(); // interval before allowing
                              // a sensor again
    }
}
```

Thread Process

- Thread Process is the most general process and can be used to model nearly anything.
  - An SC_METHOD process to model this same design would require more typing and be more difficult to understand and maintain.
- The simulation of SC_THREAD processes is slower than SC_METHOD processes.
  - If simulation speed is a current goal, limit the SC_THREAD processes.

Clocked Thread Process

- **A special case of a Thread Process.**
  - only triggered on one edge of one clock.
  - to create implicit state machines
    - An implicit state machine is one where the states of the system are not explicitly defined. Instead the states are described by sets of statements with wait() function calls between them.
    - This design creation style is simple and easy to understand.
- **Differences:**
  - has the clock that triggers the process.
    - `SC_CTHREAD(xfer, clock.pos());`
  - does not have a separate sensitivity list
    - It is only sensitive to clock edge.
- **A Point:**
  - Signals assigned new values by an SC_CTHREAD process will be not be available until after the next clock edge.

Dynamic Sensitivity

- Dynamic sensitivity lists are created during simulation time.
- When a wait(args) is executed, the thread process suspends.
- The Process is re-invoked based on the dynamic sensitivity list, which is determined by the arguments in the wait(args) function.

sc_event

- **Event**
  - Something that happens at a specific point in time.
  - Has no value or duration
- **Sc_event:**
  - A class to model an event
    - Can be triggered and caught.
- **important (the source of a few coding errors):**
  - Events have no duration → you must be watching to catch it
    - If an event occurs, and no processes are waiting to catch it, the event goes unnoticed.

sc_event

- **you can perform only two actions with an sc_event:**
  - wait for it
    - `wait(ev1)`
    - ```
      SC_THREAD(my_thread_proc);
      sensitive << ev_1; // or sensitive(ev_1)
      ```
  - cause it to occur
    - `notify(ev1)`
- **Common misunderstanding:**
  - **`if (event1) do_something`**
    - **Events have no value**
    - **You can test a Boolean that is set by the process that caused an event;**
    - **However, it is problematic to clear it properly.**
      ```

Dynamic Sensitivity

- **Given:**

```
sc_event e1,e2,e3; // events
sc_time t(200, SC_NS); // variable t of type sc_time
```

- **wait for an event in a list of events:**

```
wait(e1);
wait(e1 | e2 | e3); // wait on e1, e2 or e3
```

- **wait for an event in a list of events:**

```
wait( e1 & e2 & e3); // wait on e1, e2 and e3
```

  - **The events do not need to occur at the same time. (Events are "collected" until all three have occurred).**

- **wait for specific amount of time:**

```
wait(200, SC_NS); // wait for 200 ns
wait(t); // wait for 200 ns
```

Dynamic Sensitivity

- **wait for events with timeout:**

```
// wait on e1,e2, or e3,timeout after 200 ns
wait(200, SC_NS, e1 | e2 | e3);
// wait on e1, e2, or e3, timeout after 200 ns
wait(t, e1 | e2 | e3);
// wait on e1,e2, and e3,timeout after 200 ns
wait(200, SC_NS, e1 & e2 & e3);
```

- **`timed_out()` boolean function:**

```
wait(t_MAX_DELAY, ack_event | bus_error_event);
if (timed_out()) break; // path for a time out
```

- **wait for one delta cycle:**

```
wait( 0, SC_NS ); // wait one delta cycle
wait( SC_ZERO_TIME ); // wait one delta cycle
```

Method Process: Dynamic Sensitivity
`next_trigger()`

- Method processes may have a dynamic sensitivity list - but it is different than for a thread.
- A method process (and only a method process) may call the function **`next_trigger()`** to create dynamic sensitivity.
- The function **`next_trigger`** sets the dynamic sensitivity of the method process instance from which it is called for the **very next** occasion on which that process instance is triggered, and for that occasion only.
- Arguments are the same as `wait()`.

next_trigger(): Examples

- **Given:**

```
sc_event e1,e2,e3; // event
sc_time t(200, SC_NS); // variable t of type sc_time
```

- **trigger on an event in a list of events:**

```
next_trigger(e1);
next_trigger(e1 | e2 | e3); // trigger on e1, e2 or e3
```

- **trigger on all events in a list:**

```
next_trigger( e1 & e2 & e3); // trigger on e1, e2 and e3
```

- **trigger after a specific amount of time:**

```
next_trigger(200, SC_NS); // trigger 200 ns later
next_trigger(t); // trigger 200 ns later
```

Method Process: Dynamic Sensitivity

- If `next_trigger` is called more than once during a single execution of a particular method process, the last call prevails. (The earlier calls are cancelled).
- If an invocation of a method process does not call `next_trigger()`, then the static sensitivity list will be restored.
- if no argument is specified, then the static sensitivity list is used.
- Calling `next_Trigger()` does not suspend the current method process.

wait_until()

- **In a thread process, wait_until() methods can be used to control the execution of the process.**
  - **`wait_until(roadsensor.delayed() == true);`**
  - halts execution of the process until the new value of `roadsensor` is true.
  - `delayed()` method is required to get the correct value of the object (so that the signal may be resampled at every clock cycle).

➢A more complex example:

```
wait_until(clk.delayed() == true && rst.delayed() == false);
```

notify()

- **To trigger an event.**
- **Syntax:**
  - `event_name.notify(args);`
  - `event_name.notify_delayed(args);`
  - `notify(args,event_name);`
- **Immediate Notification**:
  - causes processes which are sensitive to the event to be made ready to run in the current evaluate phase of the ***current*** **delta-cycle**.
- **Delayed Notification:**
  - causes processes which are sensitive to the event to be made ready to run in the evaluate phase of the ***next*** **delta-cycle**.
- **Timed Notification:**
  - causes processes which are sensitive to the event to be made ready to run at a ***specified time*** in the future.

notify(): Examples

- **Given:**

```
sc_event my_event; // event
sc_time t_zero (0,SC_NS); // variable t_zero of type sc_time
sc_time t(10, SC_MS); // variable t of type sc_time
```

- **Immediate:**

```
my_event.notify(); // current delta cycle
notify(my_event); // current delta cycle
```

- **Delayed:**

```
my_event.notify_delayed(); // next delta cycle
my_event.notify(t_zero); // next delta cycle
notify(t_zero, my_event); // next delta cycle
```

- **Timed:**

```
my_event.notify(t); // 10 ms delay
notify(t, my_event); // 10 ms delay
my_event.notify_delayed(t); // 10 ms delay
```

cancel()

- cancels pending notifications for an event.
  - It is supported for delayed and timed notifications.
  - not supported for immediate notifications.

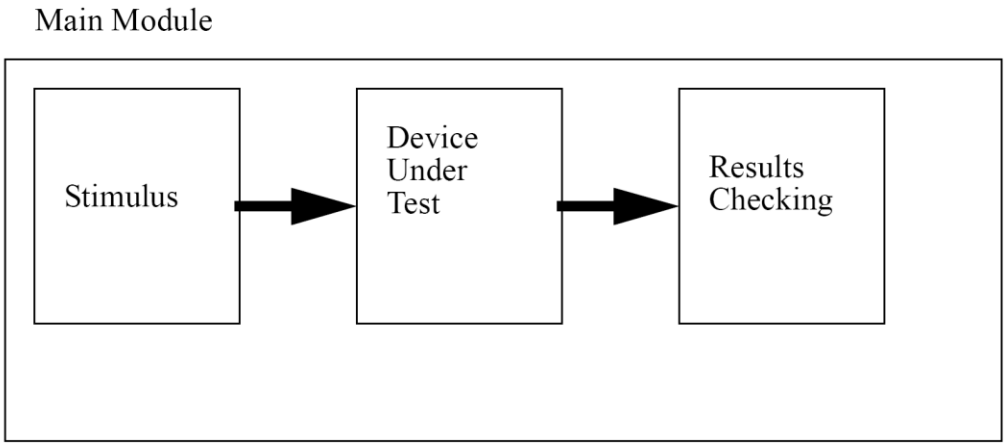
- **Given:**

```
sc_event a, b, c;  // event
sc_time t_zero (0,SC_NS);  // variable t_zero of type sc_time
sc_time t(10, SC_MS);  // variable t of type sc_time
...
a.notify();  // current delta cycle
notify(t_zero, b);  // next delta cycle
notify(t, c);  // 10 ms delay
```

- **Cancel of Event Notification:**

```
a.cancel();  // Error! Can't cancel immediate notification
b.cancel();  // cancel notification on event b
c.cancel();  // cancel notification on event c
```

Testbenches



- Testbenches:
  - **The stimulus module**
    - **can be implemented by reading stimulus from a file,**
    - **or as an SC_THREAD process,**
    - **or an SC_CTHREAD process.**
  - **The same is true of the results checking module.**
  - **Some designers combine the stimulus and results checking modules into one module**

Example: Counter

```
// count.h
#include "systemc.h"
SC_MODULE(count) {
   sc_in<bool> load;
   sc_in<int> din; // input port
   sc_in<bool> clock; // input port
   sc_out<int> dout; // output port
   int count_val; // internal data storage
   void count_up();
   SC_CTOR(count) {
     SC_METHOD(count_up); // Method process
     sensitive_pos << clock;
   }
};

// count.cc
#include "count.h"
void count::count_up() {
   if (load) {
   count_val = din;
   } else {
      count_val = count_val + 1; // could also
                                 //write count_val++
   }
   dout = count_val;
}
```

Example: Stimulus Generator (.h)

```
// count_stim.h
#include "systemc.h"
SC_MODULE(count_stim) {
   sc_out<bool> load;
   sc_out<int> din; // input port
   sc_in<bool> clock; // input port
   sc_in<int> dout;
   void stimgen();
   SC_CTOR(count_stim) {
       SC_THREAD(stimgen);
       sensitive_pos (clock);
   }
};
```

- **Clock:**
  - Clock will be generated from a clock object located in the **`sc_main`** routine discussed later.

Example: Stimulus Generator (.cc)

```
// count_stim.cc
#include "count_stim.h"
void count_stim::stimgen() {
   while (true) {
      load = true; // load 0
      din = 0;
      wait(); // count up, value = 1
      load = false;
      wait(); // count up, value = 2
      wait(); // count up, value = 3
      wait(); // count up, value = 4
      wait(); // count up, value = 5
      wait(); // count up, value = 6
      wait(); // count up, value = 7
   }
}
```

- wait():
  - Suspends the process until it is resumed by an event on a signal in the sensitivity list.

Example: Result Checker

- Options:
  - A separate module could be used to check that the counter values were correct, or
  - each of the wait statements could have a result checking statement like:

```
wait(); // count up, value = 2
if (dout != 2) {
   printf("counter failed at value 2");
}
```

watching

- SC_CTHREAD processes, just like SC_THREAD processes, typically have infinite loops.
- A designer typically wants some way to initialize the behavior of the loop or jump out of the loop when a condition occurs.

- **watching construct:**
  - monitors a specified condition.
  - When this condition occurs, control is transferred from the current execution point to the beginning of the process (where the occurrence of the watched condition can be handled).
  - allows the designer to reset a design, or jump out of a loop, without having to check the reset condition at each wait statement.
  - Watching expressions are tested at every active edge of the execution of the process.
    - Therefore these signals are tested at the wait() or wait_until() calls in the infinite loop.
  - only available for SC_CTHREAD processes.
    - **SC_THREAD** functionality may need to be augmented by the extra mechanisms of watching before eliminating SC_CTHREAD.

Watching: Example

- generate data output values that increase in value whenever a new clock edge is detected.
- If the designer wants the value of data to start again from 0, the watching expression needs to reset the design.

```
// datagen.h
#include "systemc.h"

SC_MODULE(data_gen) {
  sc_in_clk clk;
  sc_inout<int> data;
  sc_in<bool> reset;
  void gen_data();
  SC_CTOR(data_gen){
    SC_CTHREAD(gen_data,
    clk.pos());
    watching(reset.delayed() ==
    true);
  }
};
```

```
// datagen.cc
#include "datagen.h"

void gen_data() {
  if (reset == true) {
    data = 0;
  }
  while (true) {
    data = data + 1;
    wait();
    data = data + 2;
    wait();
    data = data + 4;
    wait();
  }
}
```

Watching

- **General Usage:**

```
void data_gen::gen_data () {
  // variable declarations
  // watching code
  if (reset == true) {
    data = 0;
  }
  // infinite loop
  while (true) {
    // Normal process function
  }
}
```

Multiple Watches

- Multiple watches can be added to a process.
- Test which watch expression triggered the exit from the loop. Then perform the appropriate watch action based on the expression that triggered the exit.

Local Waching

- **Global Watching:**
  - cannot be disabled.
- **Local Watching:**
  - allows you to specify exactly which section of the process is watching which signals, and where the event handlers are located.

```
W_BEGIN
  // put the watching declarations here
  watching(...);
  watching(...);
W_DO
  // This is where the process functionality goes
  ...
W_ESCAPE
  // This is where the handlers for the watched events go
  if (..) {
    ...
  }
W_END
```

Local Watches

- **Some points:**
  - All of the events in the declaration block have the **same priority**.
  - If a different priority is needed then local watching blocks will need to be **nested**.
  - Local watching only works in **SC_CTHREAD** processes.
  - The signals in the watching expressions are sampled only on the **active edges** of the process. In an SC_CTHREAD process this means only when the clock that the process is sensitive to changes.
  - Globally watched events have **higher priority than locally** watched events.

Ports and Signals

- **sc_in<porttype> // input port of type porttype**
- **sc_out<porttype> // output port of type porttype**
- **sc_inout<porttype> // inout port of type porttype**

- **Reading/Writing Ports/Signals:**
  - use the read() and write() methods
  - or the assignment operator.

```
sc_in<int> in_data;
sc_out<int> out_data;
int a;
a = in_data.read();
out_data.write(10);
```

```
a = in_data;
out_data = 10;
```

- **Arrays of Ports/Signals:**

```
sc_in<sc_logic> a[32];
```

  - creates an array of ports named a[0] to a[31] of type sc_logic.

```
sc_signal<sc_logic> i[16];
```

Resolved Logic Vectors

- **To create a resolved logic vector port:**

```
sc_in_rv<n> x;  //  input resolved logic vector n bits wide
sc_out_rv<n> y; //  output resolved logic vector n bits wide
sc_inout_rv<n> z;  //  inout resolved logic vector n bits wide
```

- **To create a resolved logic vector signal:**

```
sc_signal_rv<n> sig3;  //  resolved logic vector signal n bits wide
```

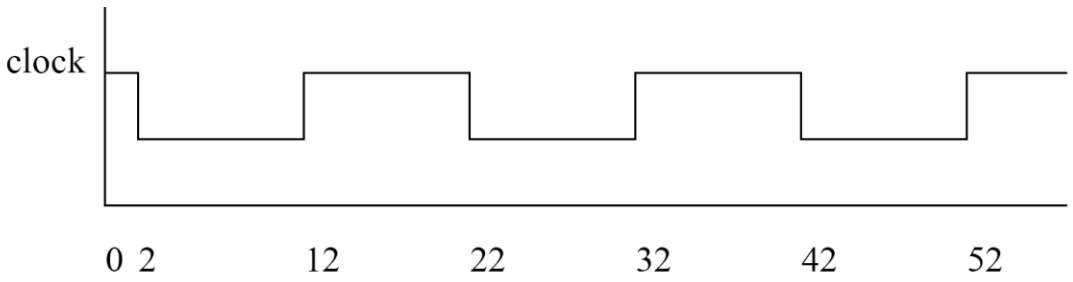
	0	1	Z	X
0	0	X	0	X
1	X	1	1	X
Z	0	1	Z	X
X	X	X	X	X

Clock Signal

- **To create a clock object:**

```
sc_clock clock1("clock1", 20, SC_NS, 0.5, 2, true);
```

- **Parameters:**
  - clock object name is “clock1”,
  - Period of 20 time units,
  - Duty cycle of 50%,
  - the first edge will occur at 2 time units,
  - the first value will be true.
- **Default values:**
  - duty cycle: 0.5,
  - first edge: 0,
  - first value: true.



Clock Objects

- **Member functions:**
  - `name()` : returns clock name
  - `period()` : returns clock period in ‘sc_time’ class
  - `duty_cycle()` : returns clock duty cycle
  - `signal()` : returns current Boolean value of the clock

Clock Signal

- **Typically clocks are created at the top level of the design in the testbench and passed down through the module hierarchy to the rest of the design.**
  - The entire design (or areas of the design) is synchronized by the same clock.

```
int sc_main(int argc, char*argv[]) {
  sc_signal<int> val;
  sc_signal<sc_logic> load;
  sc_signal<sc_logic> reset;
  sc_signal<int> result;
  sc_clock ck1("ck1", 20, SC_NS, 0.5, 0, true);
  filter f1("filter"); // instantiates a filter object
  f1.clk(ck1.signal()); // signal() method provides signal of clock object.
  f1.val(val);
  f1.load(load);
  f1.reset(reset);
  f1.out(result);
  // rest of sc_main not shown
}
```

- **For SC_CTHREAD processes the clock object is directly mapped to the clock input of the process and the signal() method is not required.**

Data Types

- **C++ Data Types:**
  - long
  - int
  - char
  - short
  - float
  - Double
    - → This means that existing algorithms can be brought directly into a SystemC design with little or no conversion.
- **SystemsC Data Types:**
  - sc_bit – 2 value single bit type
  - sc_logic – 4 value single bit type (‘0’, ‘1’, ’X’, ‘Z’)
  - sc_int – 1 to 64 bit signed integer type
    - - can be ranged
  - sc_uint – 1 to 64 bit unsigned integer type
  - sc_bigint – arbitrary sized signed integer type
  - sc_biguint – arbitrary sized unsigned integer type
  - sc_bv – arbitrary sized 2 value vector type
  - sc_lv – arbitrary sized 4 value vector type
  - sc_fixed - templated signed fixed point type
  - sc_ufixed - templated unsigned fixed point type
  - sc_fix – untemplated signed fixed point type
  - sc_ufix - untemplated unsigned fixed point type

sc_bit and sc_logic Operators

Bitwise	&(and)	|(or)	^(xor)	~(not)
Assignment	=	&=	|=	^=
Equality	==	!=		

- **Comparison:**

```
sc_bit x;
sc_logic y,z;
x == y; // sc_bit and sc_logic
y != z; // sc_logic and sc_logic
y == '1' // sc_logic and character literal
```

sc_bit and sc_logic Operators

- **Assignment:**
  - Conversion can be done by assignment
    - If ‘Z’ or ‘X’ are assigned to sc_bit, the result is undefined and a warning is generated.

```
sc_bit x;
sc_logic y;
x = y; // sc_logic to sc_bit
y = x; // sc_bit to sc_logic
```

Fixed Precision Integers

- **C++ int type:**
  - Machine dependent but usually maximum 32 bits.
- **sc_int, sc_uint:**
  - 64 bit word

- **Simulation Speed:**
  - C++ built-in data types: Fastest.
  - sc_int, sc_uint: Second Fastest.
  - sc_bigint, sc_biguint: Slowest.

```
sc_int<64> x;
sc_uint<48> y;
```

Fixed Precision Integer Operators

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

```
sc_uint<4> myrange;
sc_uint<32> myint;

myrange = myint.range(7,4);
```

```
sc_uint<4> inta;
sc_uint<4> intb;
sc_uint<8> intc;

intc = (inta, intb);
```

Fixed Precision Integer Operators

```
sc_int<5> a;
a = 13; // a gets 01101, a[4] = 0, a[3] = 1, ..., a[0] = 1
bool b;
b = a[4]; // b gets 0
c = a.range(3, 1); // c gets 110 - interpreted as -2
```

Arbitrary Precision Integers

```
sc_biguint<128> b1;
sc_biguint<64> b2;
sc_biguint<150> b3;

b3 = b1*b2;
```

- **Operators:**
  - The same as sc_int and sc_uint
- **Conversion:**
  - sc_biguint, sc_bigint, sc_int, sc_uint, and C++ integer types can all be mixed together in expressions.

Bit Vector Operators

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

- **Reduction:**
  - to find out if databus is all 0's

```
sc_bv<64> databus;
sc_logic result;
result = databus.or_reduce();
```

- **Part Selection and Concatenation:**

```
sc_bv<16> data16;
sc_bv<32> data32;
data32.range(15,0) = data16;
data16 = (data32.range(7,0), data32.range(23,16));
(data16.range(3,0),data16.range(15,12)) = data32.range(7,0);
```

Bit Vectors and Integers

- sc_bv types cannot have arithmetic performed directly on them.
  - First assign sc_bv objects to the appropriate SystemC integer.
  - Perform the arithmetic operation on the integer type.
  - Then copy the results back to the sc_bv type.
- The = operator is overloaded to allow assignment of a sc_bv type to a SystemC integer and vice versa.

Logic Vectors: sc_lv

- **Operators:**
  - The operations that can be performed on an sc_lv object are exactly the same as those for an sc_bv object.
  - The only difference is the speed of the simulation.

- **to_string() method:**
  - To print a human readable character string of the value from an sc_lv object

```
sc_lv<32> bus2;
cout << "bus = " << bus2.to_string();
```

Fixed Point Types

- When designers model at a high level, floating point numbers are useful to model arithmetic operations.
- In hardware floating point data types are typically converted or built as fixed point data types to minimize the amount of hardware needed to implement the functionality.

  - **`sc_fixed`**
  - **`sc_ufixed`**
  - **`sc_fix`**
  - **`sc_ufix`**

Fixed Point Types

- **sc_fixed and sc_ufixed:**
  - uses static arguments to specify the functionality of the type
    - Static arguments must be known at compile time.
- **sc_fix and sc_ufix:**
  - can use argument types that are nonstatic.
    - Nonstatic arguments can be variables. Types sc_fix and sc_ufix can use variables to determine word length, integer word length, etc. while types sc_fixed and sc_ufixed are setup at compile time and do not change.

Time

- **sc_time:**
  - Time units are enumerated:
    - SC_FS,
    - SC_PS,
    - SC_NS,
    - SC_US,
    - SC_MS,
    - SC_SEC

```
sc_time t(20, SC_NS);
//var t of type sc_time with value of 20ns
```

Time

- **Time Functions:**
  - To set time resolution:
    - **`sc_set_time_resolution(10, SC_PS)`**
      - Any time value smaller than this is rounded off.
      - default; 1 psec.
      - e.g.

```
sc_set_time_resolution(10, SC_PS);
sc_time t2(3.1416, SC_NS);
```

→ t2 gets 3140 psec.

  - To control simulation:
    - **`sc_start()`**
    - **`sc_stop()`**
  - To report time information:
    - **`sc_time_stamp()`**
      Returns an sc_time object with the current simulation time.
    - **`sc_simulation_time()`**
      Returns a value of type double with the current simulation time in the current default time unit.

Debugging

Text-based

cout

- SystemC provides overloaded stream insertion operators for the built-in data types, so you can just use statements such as

```
cout << mydata << endl;
```

- and it will work, even for SystemC data types.

Example (exor monitor)

```
#include "systemc.h"
#include
SC_MODULE(mon)
{
    sc_in<bool> A,B,F;
    sc_in<bool> Clk;

  void monitor()
  {
    cout << setw(10) << "Time";
    cout << setw(2) << "A" ;
    cout << setw(2) << "B";
    cout << setw(2) << "F" << endl;
    while (true)
    {
      cout << setw(10) << sc_time_stamp();
      cout << setw(2) << A.read();
      cout << setw(2) << B.read();
      cout << setw(2) << F.read() << endl;
      wait();    // wait for 1 clock cycle
    }
  }

  SC_CTOR(mon)
  {
    SC_CTHREAD(monitor, Clk.pos());
  }
};
```

- `read()` allows reading the ports.
- `cout` works fine here, as the `read()` method returns values of type `bool`, which is a built in C++ data type; but it would have worked fine for SystemC data types such as `sc_logic, sc_int`, and so on.
- Note the use of `sc_time_stamp()` to print out the current simulation time.

Cout for tracing components construction

- **Another use for cout is to find out how your design is built-up before simulation starts.**
  - Will the NAND gates of the previous chapter be constructed before the EXOR gate? They should be, but a simple way to prove it is to add a line inside each constructor printing out a message. E.g.

```
SC_CTOR(nand2)
{
   cout << "Constructing nand2" << endl;
```

Cout for tracing components construction

- **the corresponding output:**

```
Constructing stim
Constructing nand2
Constructing nand2
Constructing nand2
Constructing nand2
Constructing exor2
Constructing mon
```

- **disadvantage:**
  - it does not show which instance is referred to.
  - This can be overcome by using the name() method:

```
cout << "Constructing nand2 " << name() << endl;
```

Cout for tracing components construction

- **the corresponding output:**

```
Constructing stim
Constructing nand2 exor2.N1
Constructing nand2 exor2.N2
Constructing nand2 exor2.N3
Constructing nand2 exor2.N4
Constructing exor2
Constructing mon
```

Debugging

Graphical

Signal Tracing

- SystemC has a series of trace functions which create a VCD (**Value Change Dump**) file.
- VCD Files are text files in a known format that have been around since the time of plotters.
- There are many programs available that can read VCD files and Display them.
  - The graphical system in the Aldec tools for example can display VCD files.

Creating a Trace File

- **First step: create the trace file:**
  - The trace file is usually created at the top level after all modules and signals have been instantiated.
  - For tracing waveforms using the VCD format, the trace file is created by calling the sc_create_vcd_trace_file() function with the name of the file as an argument.
  - This function returns a pointer to a data structure that is used during tracing. For example,

```
sc_trace_file* my_trace_file;
my_trace_file= sc_create_vcd_trace_file("my_trace");
```

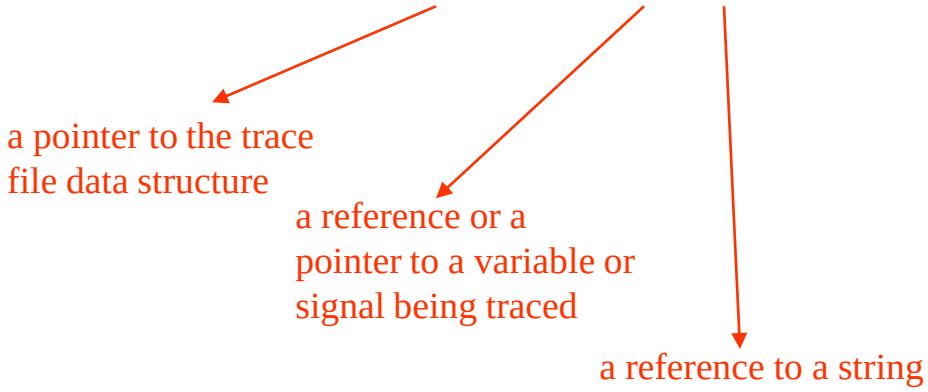
  - ➢ creates the VCD file named my_trace.vcd (the .vcd extension is automatically added).
  - ➢ A pointer to the trace file data structure is returned. You need to store this pointer so it can be used in calls to the tracing routines.

Creating a Trace File

- SystemC provides tracing functions for scalar variables and signals.

```
sc_signal<int> a;
float b;

sc_trace(trace_file, a, "MyA");
sc_trace(trace_file, b, "B");
```



Example

```
#include "counter1.h"

int sc_main(int argc, char* argv[]) {

sc_signal<bool> Rst;
sc_signal<sc_int<8> > cval;

sc_clock CLOCK("clock", 20);

counter1 U1 ("count1");
U1.Clk(CLOCK);
U1.Reset(Rst);
U1.Ctr_Val(cval);

// trace file

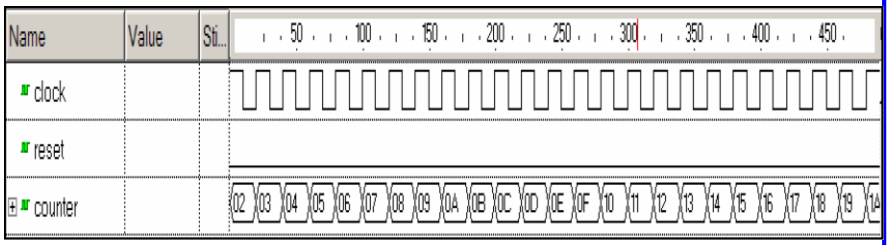
sc_trace_file *tf = sc_create_vcd_trace_file
("simplex");
// External Signals
sc_trace(tf, CLOCK.signal(), "clock");
sc_trace(tf, Rst, "reset");
sc_trace(tf, cval, "counter");

sc_start(500);

sc_close_vcd_trace_file(tf);

return (0);

}
```



Example 2 (exor)

```
#include "systemc.h"
#include "stim.h"
#include "exor2.h"
#include "mon.h"

int sc_main(int argc, char* argv[])
{
  sc_signal ASig, BSig, FSig;
  sc_clock TestClk("TestClock", 10, SC_NS,0.5, 1,
     SC_NS);

  ... instance of stim

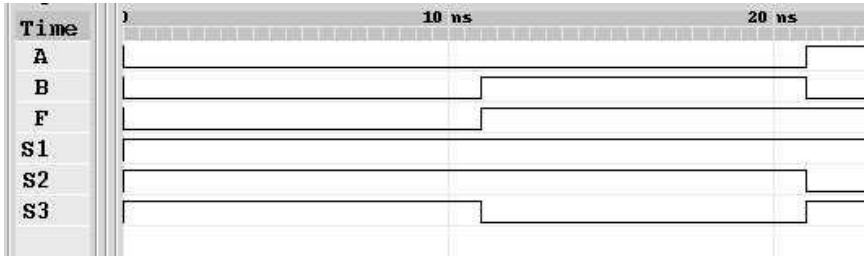
  exor2 DUT("exor2");
  DUT.A(ASig);
  DUT.B(BSig);
  DUT.F(FSig);

  ... instance of mon

  sc_trace_file* Tf;
  Tf = sc_create_vcd_trace_file("traces");
  ((vcd_trace_file*)Tf)->sc_set_vcd_time_unit(-9);
  sc_trace(Tf, ASig  , "A" );
  sc_trace(Tf, BSig  , "B" );
  sc_trace(Tf, FSig  , "F" );
  sc_trace(Tf, DUT.S1, "S1");
  sc_trace(Tf, DUT.S2, "S2");
  sc_trace(Tf, DUT.S3, "S3");

  sc_start();  // run forever
  sc_close_vcd_trace_file(Tf);
  return 0;
}
```

- Optionally specify the trace time unit.
- Possible to trace hierarchically (DUT.S1)



The sequence of operations:

- Declare the trace file
- Create the trace file
- (Optionally specify the trace time unit)
- Register signals or variables for tracing
- Run the simulation
- Close the trace file

Interfaces and Channels

- **To be merged with My first SystemC Presentation.ppt**

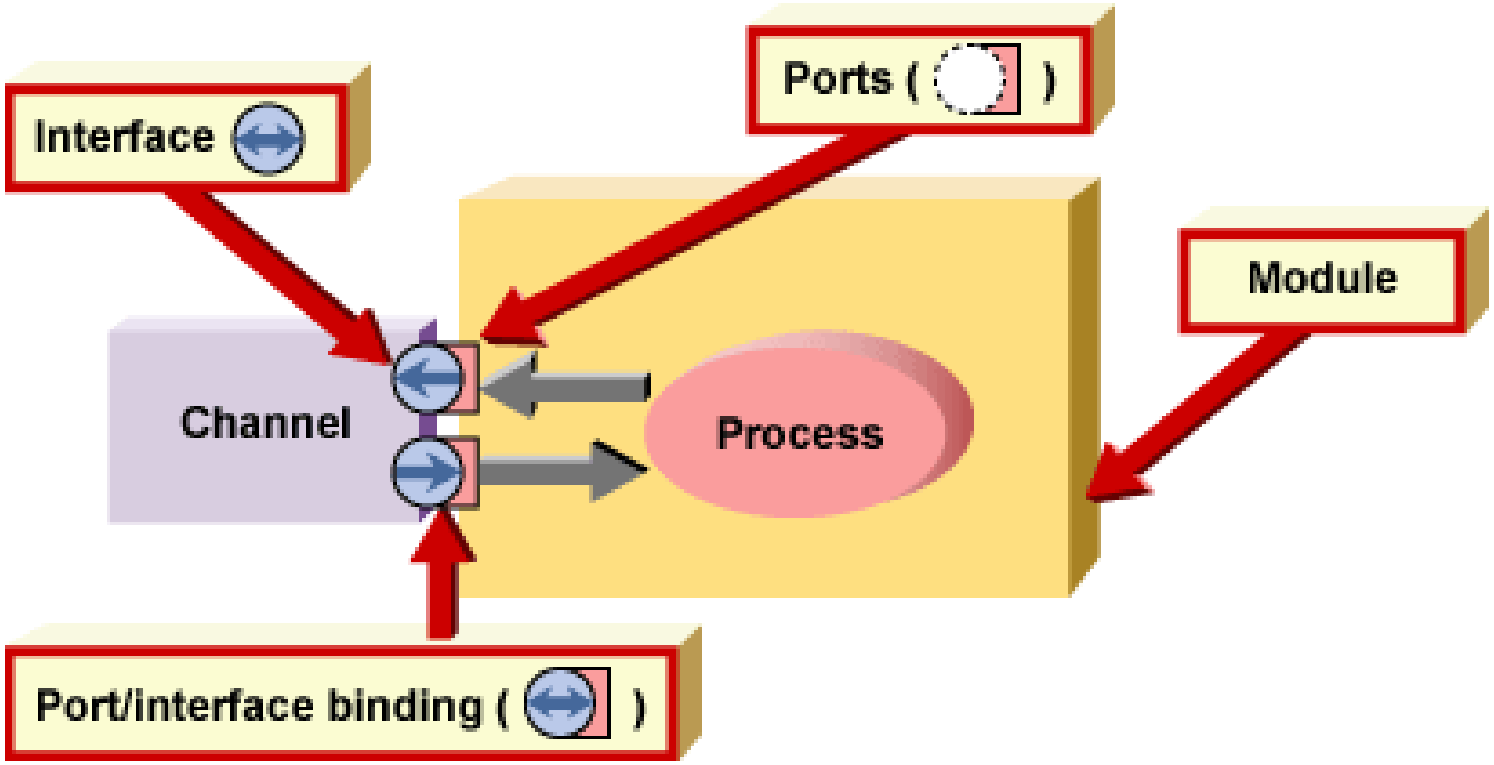
Interface and Channel

- **Interface:**
  - Defines a set of access methods.
  - It is purely functional and does not provide the implementation of the methods.
    - It only provides the method's signature.
  - Interfaces are bound to ports
    - They define what can be done through a particular port.
  - The implementation of interfaces is done inside a channel.
- **Channel:**
  - implements an interface
    - It must implement all of its defined methods.

Interface

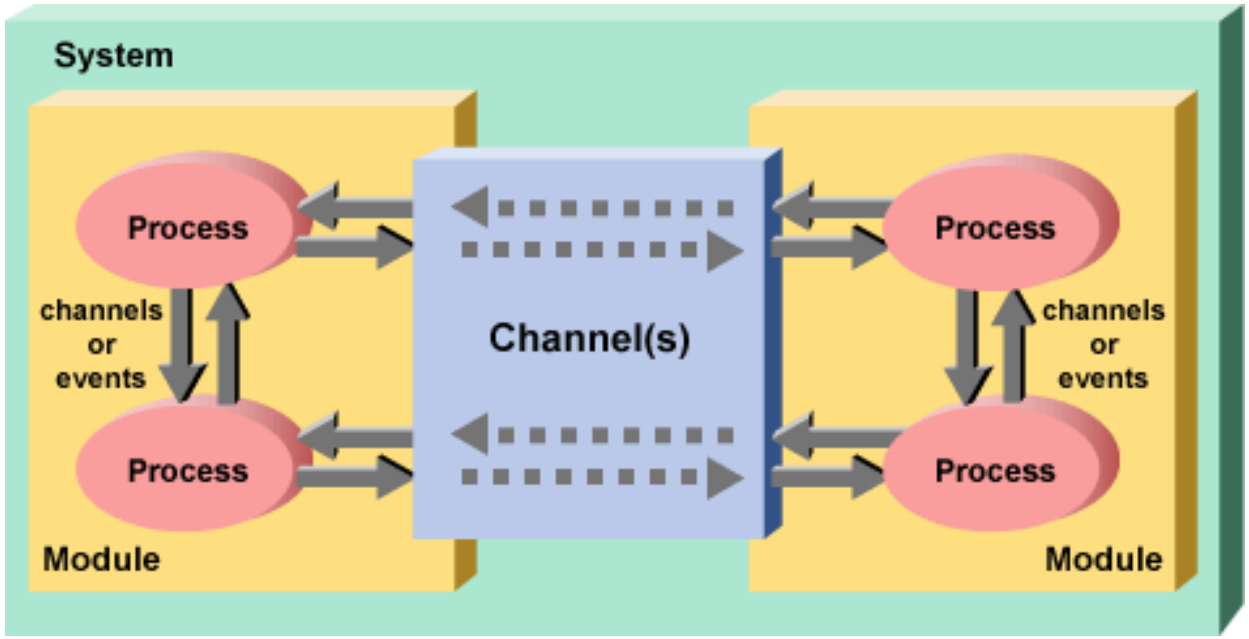
- **A port:**
  - is "bound" to an interface.
  - It is the object through which modules, and hence its processes, can access the methods of a channel.
  - A process accesses the channel by applying the interface methods to a port.
  - To connect a channel to a port, the channel must implement the interface the port is bound to.
    - This allows for refinement of channels independent of ports. Ports may be bound to multiple channels.

Interface

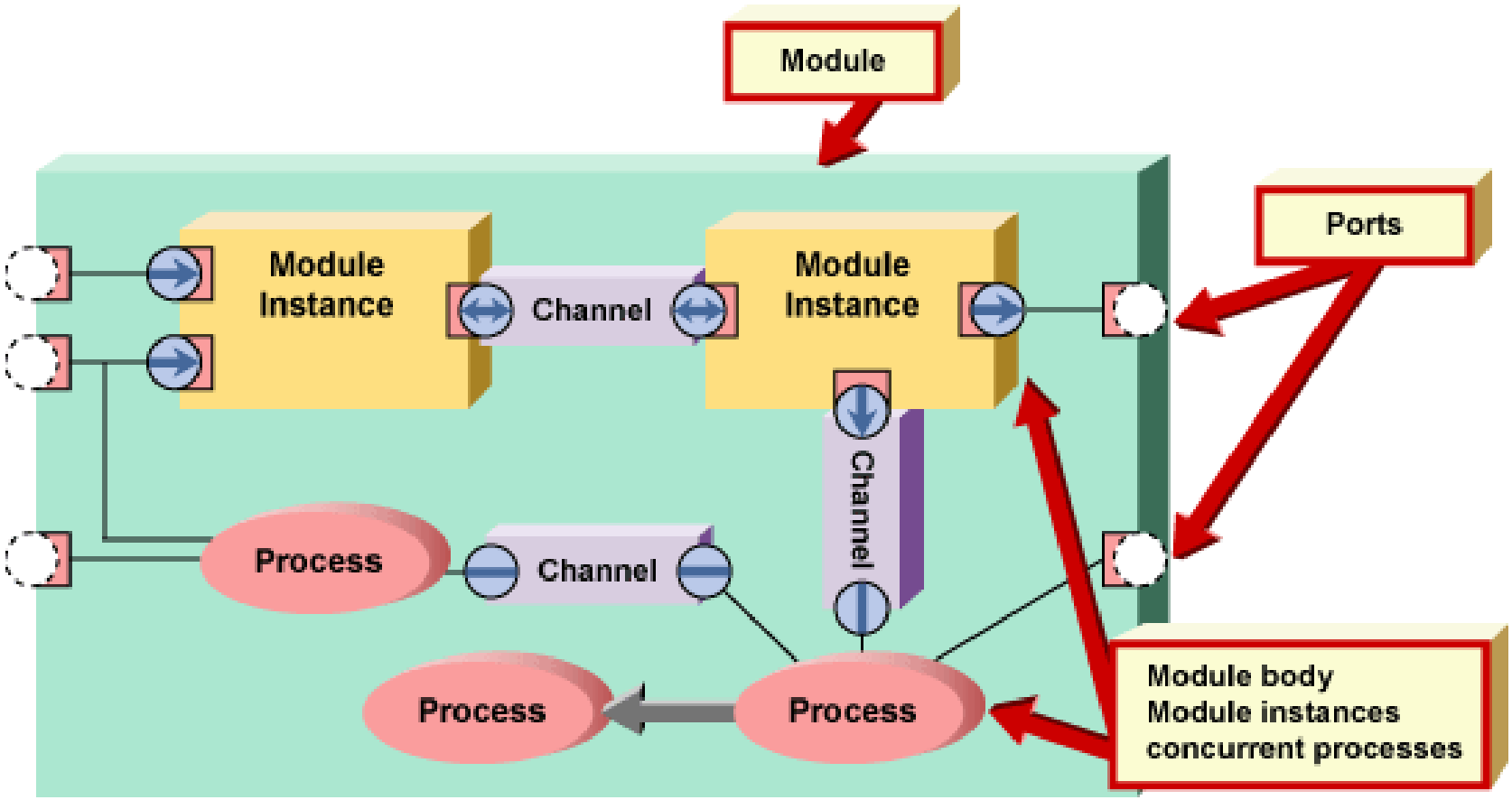

Interface
Ports ( )
Module
Channel
Process
Port/interface binding ( )

Channels

- **Channels:**
  - are used for communication between processes inside of modules and between modules.
  - Inside of a module a process may directly access a channel.
  - If a channel is connected to a port of a module, the process accesses the channel through the port.



Channels


Module
Module Instance
Channel
Module Instance
Ports
Channel
Process
Channel
Process
Process
Module body
Module instances
concurrent processes

Channels

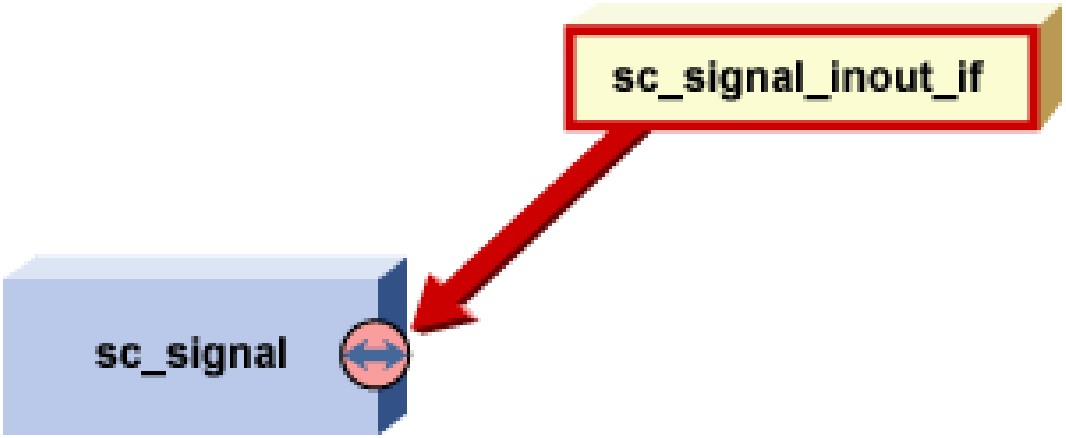
- **Two types:**
  - Primitive
  - Hierarchical
- **Primitive Channels:**
  - have no visible structure and no processes
  - cannot directly access other primitive channels.
    - sc_signal
    - sc_signal_rv
    - sc_fifo
    - sc_mutex
    - sc_semaphore
    - sc_buffer
- **Hierarchical Channels:**
  - are modules,
  - may contain processes, other modules etc.
  - may directly access other hierarchical channels.

Channels: Usage

- **Use Primitive Channels:**
  - when you need to use the request-update semantics,
  - when channels are atomic and cannot reasonably be chopped into smaller pieces,
  - when speed is absolutely crucial.
    - Using primitive channels can often reduce the number of delta cycles.
  - when it doesn't make any sense trying to build up a channel out of processes and other channels such as a semaphore or a mutex.
- **Use Hierarchical Channels:**
  - when you would want to be able to explore the underlying structure,
  - when channels contain processes or ports,
  - when channels contain other channels.

sc_signal

- **`sc_signal` channel:**
  - are often referred to as signals.
  - implements the `sc_signal_inout_if` interface .
  - are used to describe hardware signals and are typically used in RTL modeling.
  - are used for either point-to-point or multi-point communication.
    - Signals are unresolved; they may only have one writer, but may have multiple readers.



sc_signal

- **`sc_signal` channel:**
  - holds a single data value that is templatized as to the type.
  - C++, SystemC and user defined types are supported.
  - Signals implement the evaluate-update semantics to avoid data order dependencies.
  - The signal holds a `current_value`, `new_value` and `old_value`.
    - During the evaluate phase of the delta-cycle, a write to the signal will cause the value to be written into new_value. Successive writes to the signal will cause new_value to be updated every time (last one wins). During the update phase of the delta-cycle if new_value is different from current_value then current_value is moved to old_value and then new_value is moved to current_value and an event occurs.

sc_signal: Example

```
SC_MODULE (module_name) {
 sc_signal<int> d ;
 sc_signal<char> e ;
 sc_signal<sc_int<10> > f;
// rest of module not shown
} ;
```

sc_signal: Methods

- These methods return a reference to an event and may be used in static and dynamic sensitivity lists.
- `value_changed_event ()`
  - Returns a reference to an event. The event is "triggered" when the current signal value is different from the old signal value on a write
- `default_event()`
  - Same as `value_changed_event ()`
- `negedge_event()(<bool>, <sc_logic> only)`
  - Returns a reference to an event. The event is "triggered" upon a true to false change on a signal of type bool, or a non-'0' to '0' change on type sc_logic.
- `posedge_event()(<bool>, <sc_logic> only)`

sc_signal Methods

- `kind()`
  - Returns the string "`sc_signal`"
- `read()`
  - Returns a reference to `current_value`.
- `write( const T& val );`
  - If val is not equal to `current_value` then schedules an update with `val` as `new_value`.
- `get_data_ref()`
  - Returns a reference to the `current_value` (for tracing).
- `get_new_value()`
  - Returns a reference to `new_value`.
- `event()`
  - Returns a `bool`. true if an event occurred in previous delta-cycle.
- `negedge()  (<bool>, <sc_logic> only)`
  - Returns a `bool`. true if a non-false-to-false transition occurred in the previous delta-cycle.
- `posedge()  (<bool>, sc_logic only)`
  - Returns a `bool`. true if a non-true-to-true transition occurred in the previous delta-cycle.

sc_signal Methods: Examples

```
// declarations
sc_signal<int> sig_a; // channel used inside module
int a;

// use
a = sig_a.read(); // read local channel
sig_a.write(5); // write local channel
```

sc_signal Operators

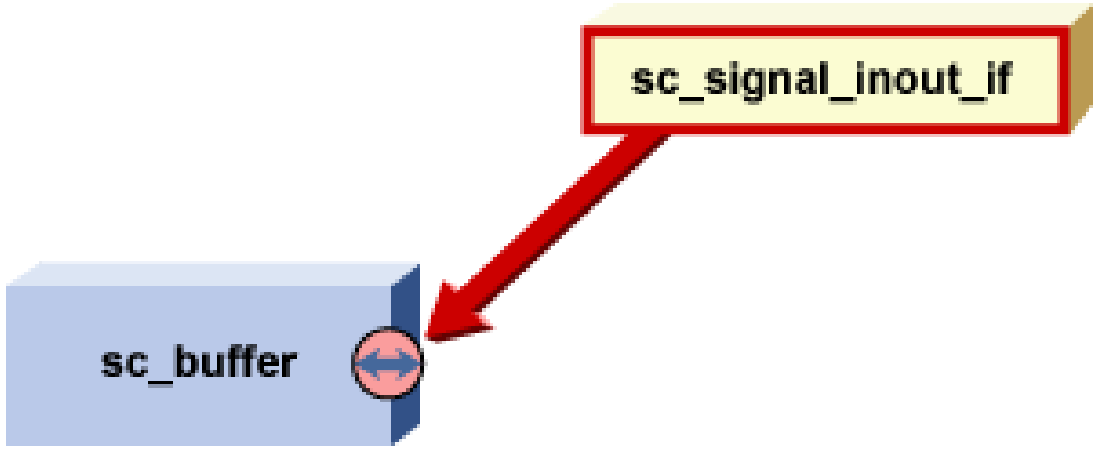
- operator ==()
- operator =()

```
// declarations
sc_signal<int> m, n, count;
int i;

// inside of process
count = 8; // same as count->(8);
m = i; // same as m.write(i);
i = n; // same as i = n.read();
if( m == n) { . . . } // compare channels
```

sc_buffer

- behaves exactly like an `sc_signal` channel except on a write, the value is always updated and the `value_changed_event` occurs, even when `new_value` = `current_value`.



sc_buffer: Methods and Operators

- The same as `sc_signal`.
  - Except `kind()` which returns "`sc_buffer`".

sc_signal_rv

- acts like an sc_signal<sc_lv<w> >.
- has a resolution function to allow for multiple writers.
- You do not specify the type -- it is sc_logic and is "built in". You only specify the width.
- sc_signal_rv channels are primarily used for describing hardware buses.

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

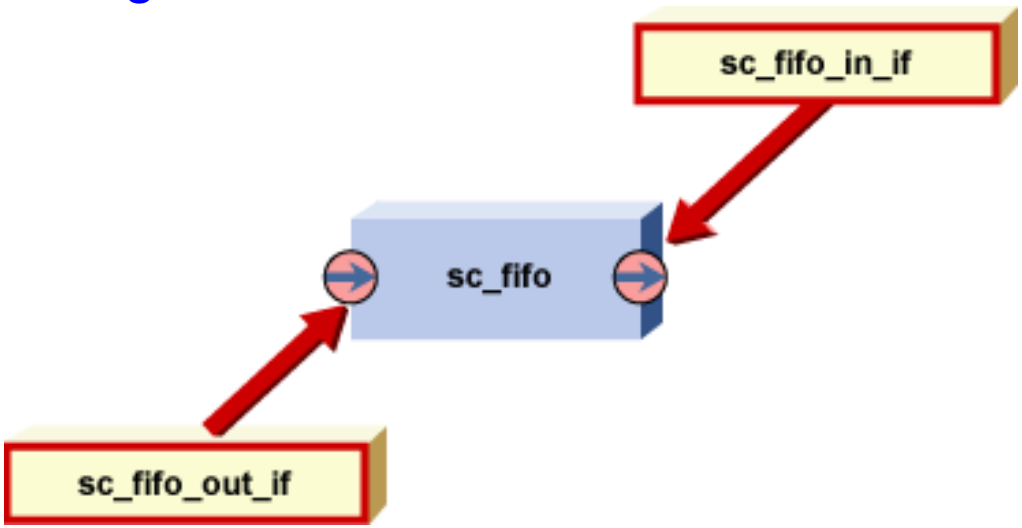
```
SC_MODULE (module_name) {
  sc_signal_rv<8> d ;
  sc_signal_rv<44> e ;
  sc_signal_rv<16 > f;
// rest of module not shown
} ;
```

sc_signal_rv: Methods and Operators

- Same as for an `sc_signal<sc_lv<w> >`

sc_fifo

- implements the `sc_fifo_in_if` and the `sc_fifo_out_if` interface.
- It is by default 16 entries deep.
  - The depth may be defined by the user during elaboration.
- It is a point-to-point connection and may only be connected to one input and one output port when connecting between modules.



sc_fifo Definition: Examples

```
SC_MODULE (module_name) {
// channels
sc_fifo<int> d; // type int, depth of 16
sc_fifo<char> e ; // type char,depth of 16
sc_fifo<sc_int<10> > f; // type sc_int<10>, depth of 16
// rest of module
} ;
```

sc_fifo Methods: Write to FIFO

- `write(value)`
  - Blocking write: Writes value into the FIFO and blocks until there is space available in the FIFO.
- `nb_write(value)`
  - Non-blocking write: Completes regardless of space available in FIFO.
  - Returns a `bool true` if the write to the FIFO was successful.
  - Writes value into the FIFO if successful.
  - Does not overwrite if no space available.
- `num_free()`
  - Returns the number of free elements in the FIFO.
- `data_read_event()`
  - Returns a reference to an event that occurs when a read access occurs.

sc_fifo Methods: Read from FIFO

- `read()`
  - Blocking read: Returns value from the FIFO.
  - Blocks if FIFO is empty until data is written.
- `read(variable)`
  - Blocking read: The value placed in variable.
  - Blocks if FIFO is empty until data is written.
  - Returns void.
- `nb_read(variable)`
  - Non-blocking read: The value read is placed in variable.
  - Returns bool true if successful read of the FIFO.
- `num_available()`
  - Returns the number (type int) of elements available in the FIFO, i.e. the number of data elements written to the FIFO but not yet read.
- `data_written_event()`
  - Returns a reference to an event that occurs when a write access occurs.

sc_fifo Methods: Examples

```
// declarations
// fifo channel of depth 16 inside a module
sc_fifo<int> fifo_a;
int a;
a = fifo_a.read();
fifo_a.write(a);
if(fifo_a.num_available() > 3)
a = fifo_a.read();
```

sc_fifo Operators

- `operator T ()`
  - Returns a value from the FIFO. If the FIFO is empty it suspends until an element is written on the FIFO.
- `operator = (val)`
  - Inserts val into the FIFO. If the FIFO is full it suspends until an element is read from the FIFO.

```
// declarations
sc_fifo<int> fifo_a;
int a;

// example use of: operator T ()
a = fifo_a; // equivalent to a = fifo_a.read();

// example use of: operator = (val)
fifo_a = a; // equivalent to fifo_a.write(a);
```

sc_mutex

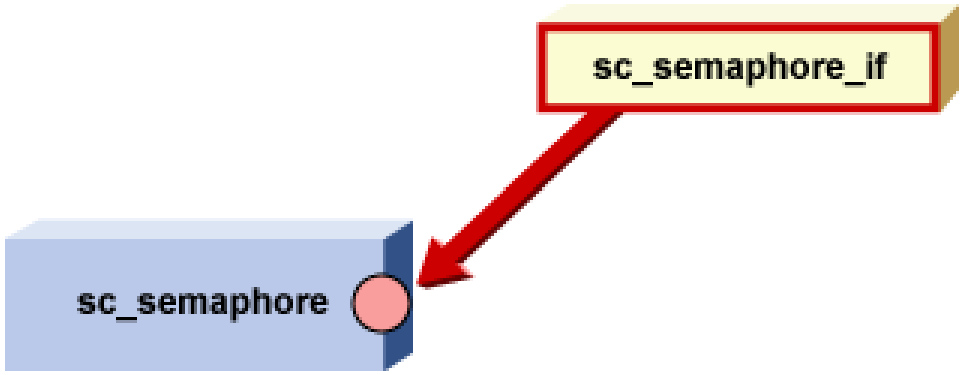
- implements the `sc_mutex_if` interface.
- is used for safe access to a shared resource, to avoid race conditions etc.
- Each process can attempt to lock the sc_mutex. Once the process locks the channel it then has access to the shared resource.
- After the process is finished with the shared resource it unlocks the channel so that other processes may access the shared resource.
  - The channel warns if multiple requests are issued during the same delta cycle.
- Only the process that locked the channel may unlock it.
  - If the channel is locked then dynamic sensitivity is used to suspend and un-suspend a requesting process.

sc_mutex: Methods

- `Kind()`
  - Returns the string "sc_mutex"
- `lock()`
  - If the mutex is not locked then locks mutex else suspends the calling process. Returns 0.
- `trylock()`
  - If the mutex is not locked then locks mutex and returns 0, else returns -1.
- `unlock()`
  - If mutex was locked by calling process then unlocks mutex, triggers any processes suspended while attempting to lock the mutex and returns 0, else returns -1.

sc_semaphore

- implements the `sc_semaphore_if` interface.
- similar to the `sc_mutex` channel but allows for limited concurrent access.
- It is created with a mandatory positive parameter of type int. The parameter is assigned to an internal "count" variable. The parameter represents the maximum number of concurrent "users" of the channel.
  - The internal count will decrement on each `wait()` or successful `trywait()`.
  - The internal count is incremented on each `post()`.
  - The channel is available as long as internal count > 0.



sc_semaphore: Methods

- `Kind()`
  - Returns the string "sc_semaphore"
- `wait()`
  - If the semaphore is available then locks semaphore decreasing by 1 the number of concurrent accesses available else suspends the calling process. Returns 0.
- `trywait()`
  - If the semaphore is available then locks semaphore, decreases by 1 the `semaphore_value` and returns 0, else returns -1.
- `post()`
  - Unlocks semaphore and increases by 1 the `semaphore_value`.
- `get_value()`
  - Returns the value of the internal count.

Ports

- **Syntax:**
  - `sc_port<interface_type, N> port_name, port_name,... ;`
  - **N is the number of channels that can be connected to the port.**

```
SC_MODULE(my_module) {
   // port in_p with 2 channels connected
   sc_port<sc_signal_in_if<int>,2> in_p;
   sc_port<sc_signal_inout_if<int> > out_p;

// body of module
};
```

Ports

- To access a channel method through an `sc_port` object the `->` operator is used.
  - The port doesn't know anything about the specific interface/channel to which it is bound.
  - The " -> " returns an interface pointer to the channel, which allows access to the interface methods of the channel.

```
SC_MODULE(my_module) {
   sc_port<sc_signal_in_if<int>,2> in_p;
   sc_port<sc_signal_inout_if<int> > out_p;
   int a,b;
// body of module
};
…
a = in_p->read(); //read from the 1st channel on in_p
b = in_p[1]->read(); // read from the 2nd channel on in_p
out_p->write(b); // write to out_p
```

Examples of FIFO Ports

```
SC_MODULE(my_module) {
   // "read" fifo channel port
   sc_port<sc_fifo_in_if<int> > p1;
   // "write" fifo channel port
   sc_port<sc_fifo_out_if<int> > p2;

   int j;
// body of module
};
…
//get an entry from fifo channel put value in j
j = p1->read();
//get an entry from fifo channel put value in j
p1->read(j);
p2->write(j); // put value from j on fifo channel
j = p1->num_available(); // get number of entries on fifo
j = p2->num_free(); //get number of entries open
```

Channels

- **Channels are used for communication.**
- **They are used in three places:**
  - To connect modules in module-to-module communication where the modules are peers.
  - In process-to-process communication within a module.
  - Between a process of a parent module and the port of a child module.

Internal Channels: Syntax

- Channels are instantiated as data members of the module for process to process communication.

```
SC_MODULE ( module_name) {
   // ports not shown
   channel_type channel_name, channel_name , ... ;
   // rest of module
} ;
```

- If accessing a local channel directly, that is not through a port, you must use the following syntax: **`channel_name.function_name`**
- If accessing a channel through a port, use:

  `port_name->function_name`

Internal Channels: Examples

```
SC_MODULE (module_name) {
   // channels
   sc_fifo<int> d; // fifo channel of type int
   sc_fifo<char> e ; //fifo channel of type char
   //fifo channel of type sc_int<10>
   sc_fifo<sc_int<10> > f;
   char c;
   int i;
// rest of module
} ;
…
f.write(1); // blocking write of fifo
c = e.read(); // blocking read of fifo

// Check to see if more than 2 entries
if(d.num_avail() > 2)
i = d.read();
```

Specialized Ports

- **sc_signal, sc_buffer, sc_signal_rv, and sc_fifo channels**
  - have specialized ports associated with them that are provided for convenience. They may be used instead of using the sc_port syntax for these channels.

```
SC_MODULE (module_name) {
 // ports
 sc_in<int> a ;
 sc_inout<sc_logic> b;
 sc_fifo_in<int> c;
…
} ;
```

The same as:

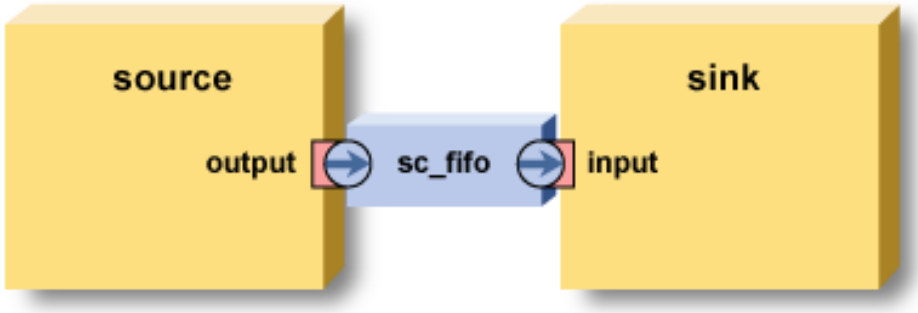
```
SC_MODULE (module_name) {
 // ports
 sc_port<sc_signal_in_if<int>,1> a ;
 sc_port<sc_signal_inout_if<sc_logic>,1> b ;
 sc_port<sc_fifo_in_if<int>,1> c;
 …
} ;
```

Hierarchical Channels

- Hierarchical channels are modules,
- They may contain processes, other modules etc.
- They may directly access other hierarchical channels.

- **Usage:**
  - Use hierarchical channels
    - when you would want to be able to explore the underlying structure,
    - when channels contain processes or ports and
    - when channels contain other channels.

Example: By Primitive Channel




```
// main.cpp
int sc_main(int argc, char **argv) {
  // fifo channel of depth 5
  sc_fifo<int> fifo_chan(5);
  // module instantiations
  source src("src");
  sink snk("snk");
  // connect src & snk
  src.output(fifo_chan);
  snk.input(fifo_chan);
  sc_start();
}
```

Example: By Primitive Channel

```
// file source.h
SC_MODULE(source) {
  // output port
  sc_port<sc_fifo_out_if<int> > output;
  // process
  void src_proc();

  SC_CTOR(source) {
    SC_THREAD(src_proc);
  }
};
```

```
// file source.cpp
void source::src_proc() {
  wait(5, SC_NS); // get off of time = 0
  for (int i = 1; i < 13; i++) {
    output->write(i);
  }
  wait(300, SC_NS); // give time to get to other side
  sc_stop();
}
```

Example: By Primitive Channel

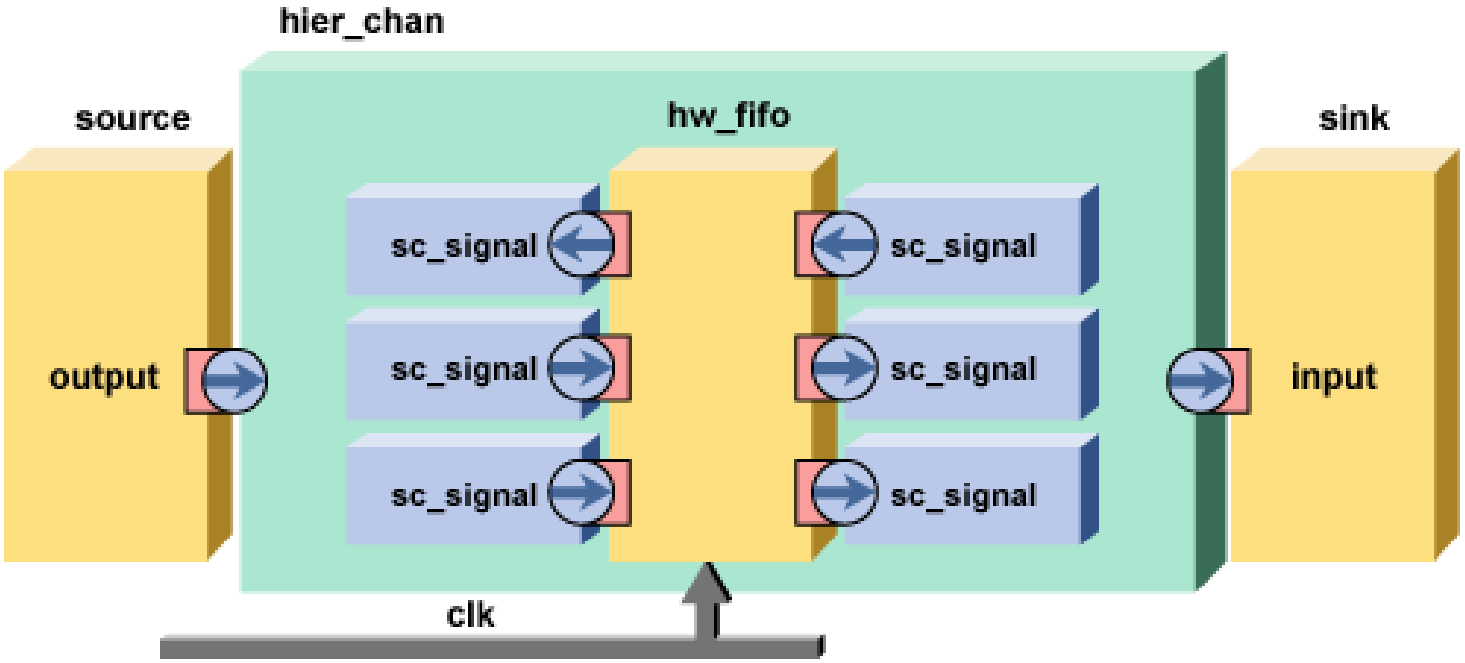
```
// file sink.h
SC_MODULE(sink) {
  // input port
  sc_port<sc_fifo_in_if<int> > input;
  // process
  void sink_proc();

  SC_CTOR(sink) {
    SC_THREAD(sink_proc);
  }
}
```

```
// file sink.cpp
void sink::sink_proc() {
  int val;
  while(true) {
     val = input ->read(); // read value
  }
}
```

Example: By Hierarchical Channel

- This example uses the same source and sink modules as in the previous example. (The code has not changed).
- The channel `hier_chan` has replaced the `sc_fifo` channel.
- The `hier_chan` channel instantiates the module `hw_fifo` which implements a FIFO.

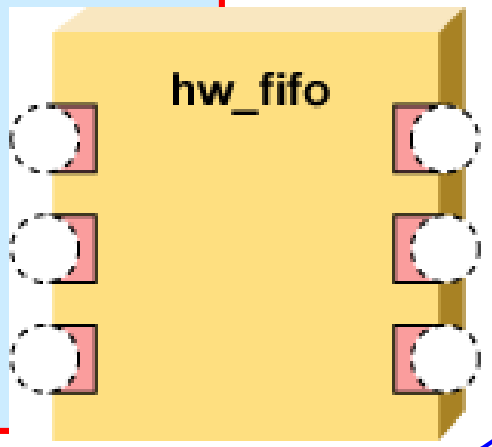


hw_fifo.h

```
// file hw_fifo.h
SC_MODULE( hw_fifo ) {
  // input: can accept at most one sample per clock cycle
  sc_in<int> data_in; // data
  sc_in<bool> data_in_valid; // sender: data is valid
  sc_out<bool> data_in_ready; //FIFO: ready to accept input
  // output: can provide at most one sample per clock cycle
  sc_out<int> data_out; // data
  sc_out<bool> data_out_valid; // FIFO: data is available
  sc_in<bool> data_out_was_read; // receiver: data was read
  sc_in<bool> clock; // the clock
  int head,tail; // head & tail pointers
  int e_cnt; // element counter
  int fifo_mem[DEPTH]; // memory

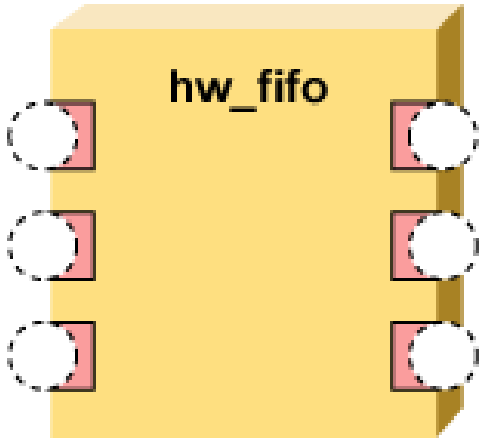
  void w_proc();
  void r_proc();

  SC_CTOR(hw_fifo) {
```



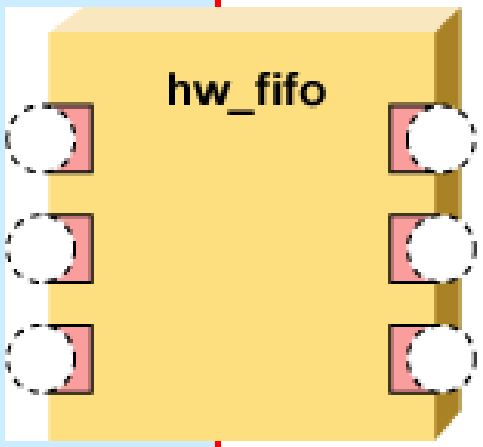
hw_fifo.h (cont.)

```
  SC_CTOR(hw_fifo) {
     head = 0;
     tail = 0;
     e_cnt = 0;
     data_in_ready.initialize(true);
     data_out_valid.initialize(false);
     SC_THREAD(w_proc);
     sensitive_pos << clock;
     SC_THREAD(r_proc);
     sensitive_pos << clock;
  }
};
```



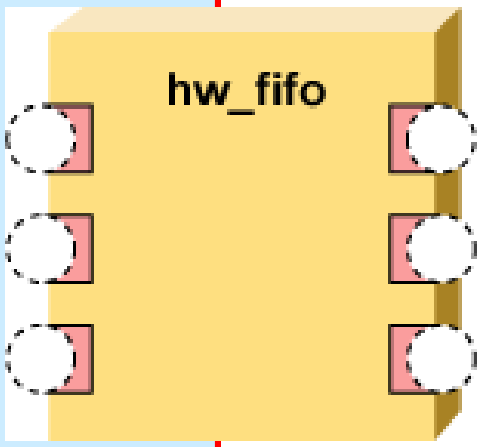
hw_fifo.cpp

```
// file hw_fifo.cpp
void hw_fifo::r_proc() { // read
  while(true) {
     wait();
     wait(0,SC_NS); // so can read every clk
     if(data_out_was_read && e_cnt !=0 ) { // was data read?
        tail++; // increment tail ptr
        if(tail == DEPTH)
        tail = 0; // wrap the pointer
        e_cnt--; // decrement element count
     }
     if (e_cnt == 0) // empty?
        data_out_valid = false; // yes, no data avail
     else {
        data_out = fifo_mem[tail]; // put data out
        data_out_valid = true; // data is avail
     }
  }
}
```



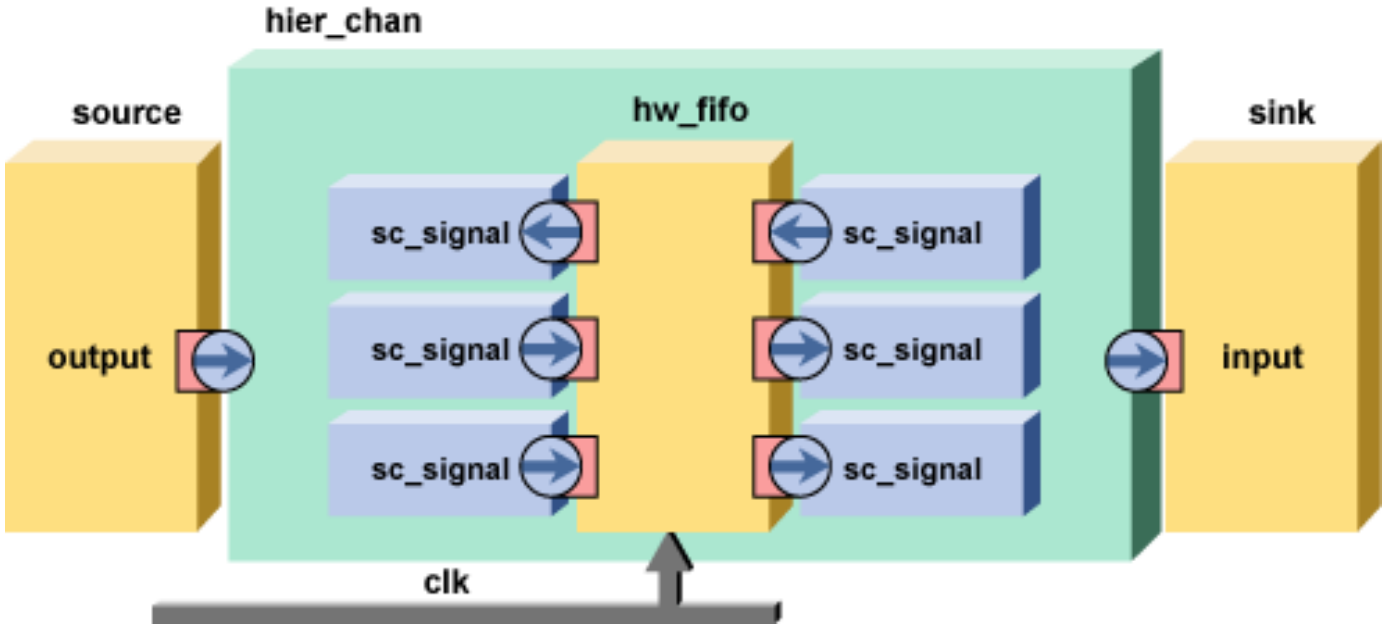
hw_fifo.cpp (cont.)

```
void hw_fifo::w_proc() { // write
  while(true) {
     wait();
     if (data_in_valid && e_cnt != DEPTH) {
        fifo_mem[head++] = data_in; // write data in fifo
        if(head == DEPTH)
        head = 0; // wrap the pointer
        e_cnt++; // increment element count
     }
     if (e_cnt == DEPTH) // full?
        data_in_ready = false; // yes, can't write
     else
        data_in_ready = true; //no, can accept more
  }
}
```



Hierarchical Channel (`hier_chan`)

- The channel inherits from the base class `sc_module`.
- It also inherits the `sc_fifo_in_if` and `sc_fifo_out_if`, meaning it will implement these interfaces. (This is so the channel can connect to the ports on the source and sink modules).



hier_chan.h

```
// hier_chan.h
#include "hw_fifo.h"

class hier_chan :
  public sc_module,
  public sc_fifo_out_if<int>,
  public sc_fifo_in_if<int>
{
  public:
  // channels to connect to ports for hw_fifo
  sc_signal<int> w_data; // data to fifo
  sc_signal<bool> w_valid; // data is valid
  sc_signal<bool> w_ready; // FIFO: ready to accept input
  sc_signal<int> r_data; // data from fifo
  sc_signal<bool> r_valid; // FIFO: data is available
  sc_signal<bool> r_was_read; // receiver: data was read

  // clock
  sc_in<bool> clock; // the clock
...
```

hier_chan.h (cont.)

```
sc_event dummy_event;

//implement methods for sc_fifo_out_if
virtual void write(const int& data_in);
virtual bool nb_write( const int& ) { return true; }
virtual int num_free() const { return 0; }
virtual const sc_event& data_read_event() const
  { return dummy_event; }

// implement methods for sc_fifo_in_if
virtual int read();
virtual void read(int &a) { a = read(); }
virtual bool nb_read(int &) { return true; }
virtual int num_available() const { return 0; }
virtual const sc_event& data_written_event() const
  { return dummy_event; }

hw_fifo fifo; // instantiate fifo
SC_CTOR(hier_chan):
fifo("fifo") // init fifo
  {
  // connect fifo
  fifo.data_in(w_data);
  fifo.data_in_valid(w_valid);
  fifo.data_in_ready(w_ready);
  fifo.data_out(r_data);
  fifo.data_out_valid(r_valid);
  fifo.data_out_was_read(r_was_read);
  fifo.clock(clock);
  }
};
```

hier_chan.cpp

```
// file hier_chan.cpp

void hier_chan::write(const int& data_in){
  while(w_ready.read() == false)
    wait(clock->posedge_event());//wait until can send data
  w_data.write(data_in); // write data
  w_valid.write(true); // set valid
  wait(clock->posedge_event() );//wait until the next clock
  w_valid.write(false); // clear valid
}

int hier_chan::read() {
  int temp;
  // read from source
  while(r_valid.read() == false)
    // wait until data there to read
    wait(clock->posedge_event());
  temp = r_data.read(); // read data
  r_was_read.write(true); // data was read
  wait(clock->posedge_event());// wait until the next clock
  r_was_read.write(false); // clear
  return(temp);
}
```

main.cpp

```
// file main.cpp
int sc_main(int argc, char **argv) {
   // channels
  hier_chan adapt("adapt");

   sc_time t(10, SC_NS);
   sc_clock clock ("clock", t);
   // module instantiations
  source src("src");
  sink snk("snk");

   // connect src & snk
  src.output(adapt);
  snk.input(adapt);

   // connect hier_chan ports
  adapt.clock(clock);

  sc_start();
}
```

References

- System Design with SystemC, QA 76.9 .S88 S9525, 2002 (Main reference and succinct)

- A SystemC Primer, Second Edition (Hardcover), 2004 (2002 PDF exists) by J. Bhasker (RTL SystemC).

- SystemC: From the Ground Up, by David C. Black and Jack Donovan, 2004.