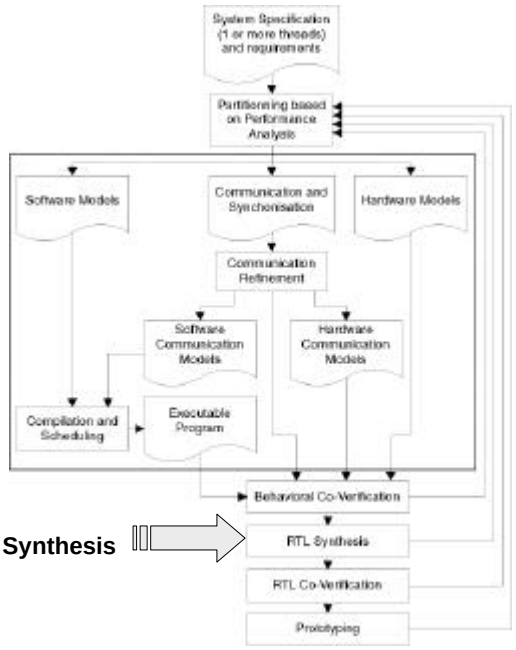
Co-design Methodology and Synthesis



Synthesis

SystemC

Introduction to SystemC

C++

- C++ class library and a methodology ⇒
  - create a cycle-accurate model of software algorithms,
  - hardware architecture, and
  - interfaces of an SoC (System On a Chip)
  - system-level designs.
- Use SystemC and other 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
  - a C++ program with the same behavior as the system when executed.

Modeling Constructs

- Constructs to model system architecture
  - Hardware timing
  - Concurrency
  - Reactive behavior
- Alternatives:
  - Proprietary extensions to the language
    - Not an acceptable solution for the industry
  - Use C++ object-oriented
    - Provides the ability to extend the language through classes
    - No need for adding new syntactic constructs
    - SystemC is a C++ Class Library

Executable Specifications Benefits

- Avoid inconsistency and errors
- Ensure completeness
- Ensure unambiguous interpretation of the specification.
- Validate system functionality before implementation
- Create early performance models of the system and validate system performance.
- Testbench used to test the executable specification
  - Can be refined or used as is to test the implementation
  - Drastically reduce the time for implementation verification

SystemC Highlights

- Supports
  - Hardware-software co-design
  - Description of hardware, software, and interfaces in a C++ environment.
- Modules
  - Hierarchical entity, can have other modules or processes contained in it.
- Processes
  - Describe functionality
  - Contained inside modules
  - Three different process abstractions
- Ports
  - Modules have ports through which they connect to other modules.
  - Single-direction and bidirectional ports.
- Signals
  - Resolved signals have more than one driver (a bus)
  - Unresolved signals can have only one driver.

Highlights (cont'd)

- Rich set of data types
  - fixed precision data types allow for fast simulation,
  - arbitrary precision types can be used for computations with large numbers,
  - fixed-point data types can be used for DSP applications
  - two-valued and four-valued data types.
- Clocks
  - notion of clocks (as special signals)
  - Multiple clocks, with arbitrary phase relationship, supported.
- Ultra light-weight cycle-based simulation kernel $\Rightarrow$ high-speed.
- Multiple abstraction levels
  - high-level functional models
  - detailed clock cycle accurate RTL models
- Iterative refinement of high level models into lower levels of abstraction

Where to Find SystemC ?

- On the web, the official SystemC site:
  `http://www.systemc.org`
- Installed on the system at UdeM:
  `/u/lablasso/systemc-`<version>
  (where “version” is a number (“2.0” for example))
  - The documentation is in the `/doc` subfolder
  - Examples are in the `/examples` subfolder
  - Sources are in the `/src` subfolder
  - The internal structure of SystemC can be found on the lab LASSO web page, inside the “documentation” section:
    `http://www.iro.umontreal.ca/~lablasso`

Linking SystemC Library

Simple Example

(file main.cpp)

```
#include<systemc.h>

int sc_main(int argc, char **argv)
{
  cout << "starting simulation" << endl;
  sc_start(1000);
  cout << "end of simulation" << endl;
  return(0);
}
```


Simple Example of a Makefile

```
SYSTEMCDIR = /u/lablasso/systemc-2.0 ##SystemC installation path
TARGET_ARCH = linux                  ## architecture
CXX = g++
MODULE = systemc_test                ## name of the executable
INCDIR = -I. -I.. -I$(SYSTEMCDIR)/include -I$(SYSTEMCDIR)/src
LIBDIR = -L. -L.. -L$(SYSTEMCDIR)/lib-$(TARGET_ARCH)
LIBS   =  -lsystemc -lm $(EXTRA_LIBS)
all : $(MODULE)                      ## dependencies list
$(MODULE) : main.o
        $(CXX) -o $(MODULE) $(LIBDIR) $(LIBS) main.o
main.o : main.cpp
        $(CXX) $(INCDIR) -c main.cpp
```

Compilation and Execution

```
% make
g++  -I. -I.. -I/u/lablasso/systemc-2.0/include -
I/u/lablasso/systemc-2.0/src  -c main.cpp
g++  -o systemc_test -L. -L.. -L/u/lablasso/systemc-
2.0/lib-linux -lsystemc -lm  main.o
% ./systemc_test
               SystemC 2.0 --- Apr  9 2002 16:05:08
        Copyright (c) 1996-2001 by all Contributors
                    ALL RIGHTS RESERVED


starting simulation
end of simulation
%
```


Modules and Hierarchy

Introduction

- basic building block to partition a design
  - break complex systems into smaller more manageable pieces
  - split complex designs among a number of different designers
- hide internal data representation and algorithms from other modules.
  - This forces designers to use public interfaces to other modules
  - the entire system becomes easier to change and maintain
- Example
  - One can decide to completely change the internal data representation and implementation of a particular module.
  - if external interface and internal function remain the same, the users of the module do not know that the internals were changed. ⇒ allows designers to optimize the design locally
- Syntax

  SC_MODULE(transmit) { //macro
  (is equivalent to…)
  class transmit : sc_module { //class declaration

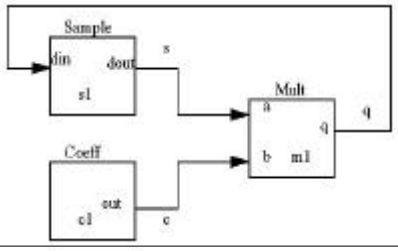
Module Ports

- Pass data to and from the processes of a module
- Mode
  - in, out, inout
- Data type
  - C++ data type
  - SystemC data type
  - user defined type


```
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

- Can be local to a module
- Used to connect ports of lower level modules
- Carry data
- Aren't declared with mode: in, out, or inout.

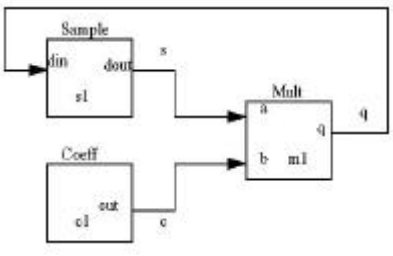


- Lower level modules instantiated:
  Sample, coeff, and mult
- Module ports connected by local signals:
  Q, s, and c.

```
// filter.h
#include "systemc.h"
#include "mult.h"
#include "coeff.h"
#include "sample.h"
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);
      }
}
```


Named Connection

```
#include "systemc.h"
#include "mult.h"
#include "coeff.h"
#include "sample.h"
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);
      }
}
```



Internal Data Storage

```
// 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
      // internal data storage
      int count_val;
      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++
      }
      dout = count_val;
}
```


Processes

- Provide the module functionality
- Look very much like normal C++ methods
- Functions that are registered with the SystemC kernel
- Called whenever “sensitive to” signals change value
- Contained statements are executed sequentially until end occurs, or suspension by a wait call
- Different types of processes including
  - Method processes
  - Thread processes
  - Clocked thread processes

Process 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;
}
```

- Module *dff* contains an SC_METHOD process named *doit*.
- An sc_method process
  - Triggered by events and
  - Executes all of the statements in the method before returning control to the kernel
- Sensitive to positive edge changes on input port clock
- The process runs once when the first event (positive edge on *clock*) is received. It
- Executes the assignment of *din* to *dout*, then returns control to the System kernel
- Another event causes the process to be invoked again, and the assignment statement is executed again

Module Constructor

- Creates and initializes an instance of a module
- Creates the internal data structures
- Initializes them to known values
- Instance name of the module is passed to the constructor at instantiation

  ⇒ helps identify the module when errors occur or when reporting information from the module

Constructor Example

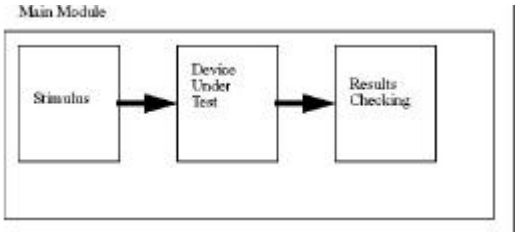
```
// ram.h
#include "systemc.h"
SC_MODULE(ram) {
      sc_in<int> addr;
      sc_in<int> datain;
      sc_in<bool> rwb;
      sc_out<int> dout;
      int memdata[64]; // local memory storage
      int i;
      void ramread();
      void ramwrite();
      SC_CTOR(ram){
            SC_METHOD(ramread);
            sensitive << addr << rwb;
            SC_METHOD(ramwrite)
            sensitive << addr << datain << rwb;
            for (i=0; i++; i<64) {
                  memdata[i] = 0;
            }
      }
};
```

When a RAM module is instantiated:

- the constructor called
- data allocated for the module
- the two processes registered with the kernel.
- the for loop executed ⇒ initialize all the memory locations of the newly created ram module

TestBenches

- Provide stimulus to a design under test (DUT) and check design results
- Can be implemented in a number of ways
- Stimulus generated by one process and results checked by another
- Stimulus embedded in the main program and results checked in another process
- Checking can be embedded in the main program, etc.



```
// 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);
        }
};
```

```
// 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
        }
}
```


Processes

Process Basics

- Some behave like functions, started when called, and return execution when complete
- Others: called only once at the beginning of simulation and either actively executing or suspended waiting for a condition
- Condition: clock edge, a signal expression, combination of both
- No hierarchy ⇒ no process will call another process directly (?)
- Can call methods and functions that are not processes
- Have sensitivity lists: a list of signals that cause the process to be invoked, whenever the value of a signal in this list changes
- May cause other processes to execute by assigning new values to signals in the sensitivity list of the other process
- An event on a signal is a change in the value of the signal
- If a signal has a current value of 1 and a new assignment updates the value to 0
  - An event will occur on the signal
  - All processes sensitive to that signal will be activated

Method Process

```
// 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);
      }
};
```

```
// 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;
      }
}
```

- module *rcv* has input *xin*, output *id,* a single method *extract_id,* sensitive to *xin*
- When invoked, *extract_id* executes and assigns value to *id*
- When terminates, control returned back to scheduler
- When a method process is invoked, it executes until it returns.
- recommended not write infinite loops within a method process otherwise
  - control will never be returned back to the simulator

Thread Processes

- Can be suspended and reactivated
- Can contain wait() functions that suspend process execution until an event occurs on one of the signals the process is sensitive to
- An event reactivates a thread process from the statement the process was last suspended
- The process will continue to execute until the next wait()
- The input signals that cause the process to reactivate are specified by the sensitivity list.
- The sensitivity list specified in the module constructor
  - Same syntax used in Method Processes

// 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 Process
            sensitive << roadsensor;
            sensitive_pos << clock;
      }
};
```

// traff.cc

```
#include "traff.h"
void traff::control_lights() {
      NSgreen = true; //…
      while (true) {
            while (roadsensor == false) wait();
            NSyellow = true; // road sensor triggered, set NS to yellow …
            for (i=0; i<5; i++) wait();
            EWgreen = true; // yellow interval over set EW to green …
            for (i= 0; i<50; i++) wait();
            EWyellow = true; // times up for EW green, set EW to yellow…
            for (i=0; i<5; i++)  wait();
            NSyellow = false; // times up for EW yellow set NS to green
            for (i=0; i<50; i++) wait(); // wait one more long
            // interval before allowing
            // a sensor again
      }
}
```

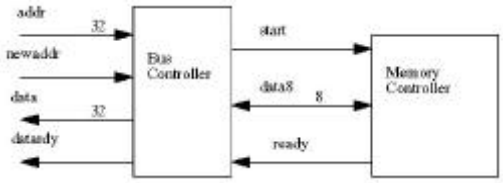
Thread processes implemented as co-routines
Slower than SC_METHOD processes

Clocked Thread Process

- Special case of a thread process
- Help designers describe their design for better synthesis results
- Only triggered on one edge of one clock, which matches the way hardware is typically implemented with synthesis tools
- Can be used to create implicit state machines within design descriptions 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
- An explicit state machine would define the state machine states in a declaration and use a case statement to move from state to state.
- To illustrate the clock thread process, a bus controller example will be presented.
- The example is a bus controller for a microcontroller application. It is a very simple design so that the design can be described easily.
- Let's assume that we have a microcontroller with a 32-bit internal data path
- Only one 8-bit external data path to get data to and from the controller.
- Every address and data transaction will have to be multiplexed out over the 8-bit bus, 8 bits at a time.
- This is a perfect application for an implicit state machine and an SC_CTHREAD process.

// bus.h

```
#include "systemc.h"
SC_MODULE(bus) {
      sc_in_clk clock;
      sc_in<bool> newaddr;
      sc_in<sc_uint<32> > addr;
      sc_in<bool> ready;
      sc_out<sc_uint<32> > data;
      sc_out<bool> start;
      sc_out<bool> datardy;
      sc_inout<sc_uint<8> > data8;
      sc_uint<32> tdata;
      sc_uint<32> taddr;
      void xfer();
      SC_CTOR(bus) {
            SC_CTHREAD(xfer, clock.pos());
            // ready to accept new address
            datardy.initialize(true);
      }
};
```



// bus.cc

```
#include "bus.h"
void bus::xfer() {
  while (true) {
    // wait for a new address to appear
    wait_until( newaddr.delayed() ==
    true);
    // got a new address so process it
    taddr = addr.read();
    // cannot accept new address now
    datardy = false;
    // new addr for memory controller
    start = true;
    data8 = taddr.range(7,0); wait();
    // wait 1 clock between data transfers
    start = false;
    data8 = taddr.range(15,8); wait();
    data8 = taddr.range(23,16); wait();
    data8 = taddr.range(31,24); wait();
    // now wait for ready signal from memory
    // controller
    wait_until(ready.delayed() == true);
    // now transfer memory data to databus
    tdata.range(7,0) = data8.read(); wait();
    tdata.range(15,8) = data8.read(); wait();
    tdata.range(23,16) = data8.read();
    wait();
    tdata.range(31,24) = data8.read();
    data = tdata;
    // data is ready, new addresses ok
    datardy = true;
  }
}
```

SC_CTHREAD particularities

- Sc_cthread ≠ sc_thread
  - Described in a module constructor not only by the name but also the clock edge that triggers the process
  - Does not have a separate sensitivity list
  - Sensitivity list is the specified clock edge only
  - Activated whenever the specified clock edge occurs
  - Ex: process *xfer* will execute on every +ive edge of *clock*
  - Port *datardy* to initialize it to true. In case a port is not yet bound, this is the only
  - signals assigned new values by an SC_CTHREAD process will be not be available until after the next clock edge occurs
  - Ex: a clock is passed to the bus module through port clock. Port clock is an *sc_in_clk* port.
  - *pos*() or *neg*() method of this port is passed to the SC_CTHREAD constructor to specify the clock edge that triggers the process

Wait Until

- In an SC_CTHREAD process *wait_until*() methods can be used to control the execution of the process.
- The *wait_until*() method will halt the execution of the process until a specific event has occurred.
- This specific event is specified by the expression to the wait_until() method.
- Ex: wait_until(roadsensor.delayed() == true);
- halt execution of the process until the new value of *roadsensor* is true.
- The *delayed*() method is required to get the correct value of the object
- compilation error if *delayed*() method is not present.
- Only a Boolean expression is allowed as argument of the wait_until() function and
- only Boolean signal objects can be used in the Boolean expressions.
- Boolean signal objects include clock type sc_clock, signal type sc_signal<bool>, and port types sc_in<bool>, sc_out<bool>, and sc_inout<bool>.
- Ex: wait_until(clock.delayed() == true && reset.delayed() == false);

Watching

- SC_CTHREAD processes, just like SC_THREAD processes, typically have infinite loops that will continuously execute
- A designer typically wants some way to initialize the behavior of the loop or jump out of the loop when a condition occurs
- This is accomplished through the use of the watching construct
- The watching construct will monitor 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
- Only available for SC_CTHREAD processes
- The delayed() function allows the compiler to identify signals that are used in watching expressions
- A lazy evaluation algorithm is used for these signals which dramatically increases simulation performance

Watching Example

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

Global Watching

- Watching expressions are tested at every active edge of the execution of the process
  ⇒ Signals tested at the wait() or wait_until() calls in the infinite loop
- Unexpected consequence of starting again the the process:
  – All variables defined locally within the process lose their value.
  ⇒ Variable value needed to be kept between invocations of the process, declare it in the module, outside of the process
- Multiple watches can be added to a process
- The data type of the watched object must be of type *bool*
- This type of watching is called global watching and cannot be disabled
- To watch different signals at different times, use local watching

Local Watching

- Local watching allows you to specify exactly which section of the process is watching which signals, and where the event handlers are located
- Functionality specified with 4 macros:

```
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
```

About local watching

- All events in the declaration block have the same priority
  - ⇒ If different priority needed then local watching blocks need to be nested
- Works in SC_CTHREAD processes
- Signals in the watching expressions are sampled only on the active edges of the process
- Globally watched events have higher priority than locally watched events

Interrupt Memory to Databus Transfer

```
// watchbus.cc
#include "bus.h"
void bus::xfer() {
  while (true) {
    // wait for a new address to appear
    wait_until( newaddr.delayed() ==
    true);
    // got a new address so process it
    taddr = addr;
    // cannot accept new address now
    datardy = false;
    // new addr for memory controller
    start = true;
    data8 = taddr.range(7,0); wait();
    // wait 1 clock between data transfers
    start = false;
    data8 = taddr.range(15,8); wait();
    data8 = taddr.range(23,16); wait();
    data8 = taddr.range(31,24); wait();
    // wait for ready from memory
    wait_until(ready.delayed() == true);
    W_BEGIN
      watching(reset.delayed());
      // reset will trigger watching
    W_DO
      // the rest of this block is as before
      // transfer memory data to databus
      tdata.range(7,0) = data8.read(); wait();
      tdata.range(15,8) = data8.read(); wait();
      tdata.range(23,16) = data8.read();wait();
      tdata.range(31,24) = data8.read();
      data = tdata;
      // data is ready, new addresses ok
      datardy = true;
    W_ESCAPE
      if (reset) {
        datardy = false;
      }
    W_END
  }
}
```

Ports and Signals

- Ports of a module
  - External interface that pass information to and from a module
  - Trigger actions within the module
- Signals create connections between module ports allowing modules to communicate
- A port can have three different modes of operation
  - An input port transfers data into a module.
  - An output port transfers data out from a module,
  - An inout port transfers data both into and out of a module depending on module operation

Port binding

- A signal connects the port of one module to the port of another
- 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 write operation has finished execution, or has been suspended ⇒
  - all operations within the process will work with the same value of the signal
  - prevents some processes seeing the old value while other processes see the new value during execution.
  - All processes executing during a time step will see the old value of the signal
- signal semantics same as VHDL signal operation and Verilog deferred assignment behavior

Scalar ports and signals

- C++ built in types
  - long
  - int
  - char
  - short
  - float
  - double

- SystemC types
  - sc_int<n>
  - sc_uint<n>
  - sc_bigint<n>
  - sc_biguint<n>
  - sc_bit
  - sc_logic
  - sc_bv<n>
  - sc_lv<n>
  - sc_fixed
  - sc_ufixed
  - sc_fix
  - sc_ufix
  - User defined structs

Ports described using the following syntax
sc_in<porttype> // input port of type porttype
sc_out<porttype> // output port of type porttype
sc_inout<porttype> // inout port of type porttype

R/W Ports and Signals

- For R/W
  - The read() and write() methods
  - The assignment operator
- Using assignment operator: more concise and more like HDL code
- Use the read() and write() methods explicitly
  - To clarify the exact intent
  - If an implicit type conversion needed
  - The type that you are reading or writing is different from the port type

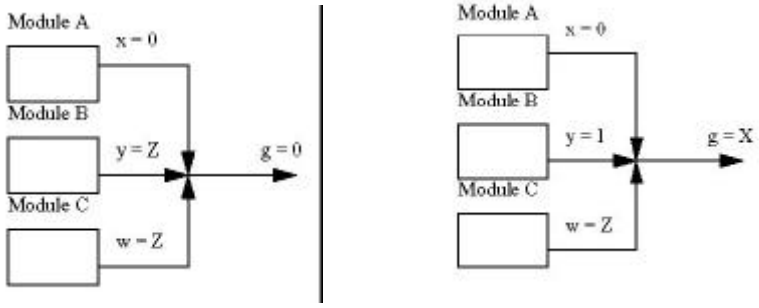
Array Ports and Signals

- Same syntax as C++

```
// create ports a[0] to a[31] of type sc_logic
    sc_in<sc_logic> a[32];
// create signals i[0] to i[15] of type sc_logic
    sc_signal<sc_logic> i[16];
```

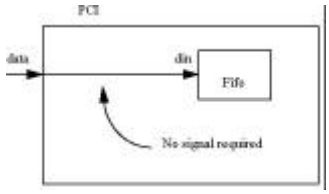

Resolved Logic Vectors

- Bus resolution becomes an issue when more than one driver is driving a signal



Port Binding

- Ports are always bound to at most one signal
- That signal may be a complex signal such as a structure
- Signal binding occurs during module instantiation
- When building a hierarchical design structure, modules are instantiated within other modules to form the hierarchy of the design
- No signal needed when a top level module port is directly bound to a lower level module port during instantiation



Syntax

```
//input resolved logic vector n bit wide
    sc_in_rv<n> x;
// output resolved logic vector n bit wide
    sc_out_rv<n> y;
// inout resolved logic vector n bit wide
    sc_inout_rv<n> z;
// resolved logic vector signal n bit wide
    sc_signal_rv<n> sig3;
```

Signal Binding

```
// statemach.h
#include "systemc.h"
SC_MODULE(state_machine) {
      sc_in<sc_logic> clock;
      sc_in<sc_logic> en;
      sc_out<sc_logic> dir;
      sc_out<sc_logic> status;
      // ... other module statements
}; //-----------------------------------------------
// controller.h
#include "statemach.h"
SC_MODULE(controller) {
      sc_in<sc_logic> clk;
      sc_out<sc_logic> count;
      sc_in<sc_logic> status;
      sc_out<sc_logic> load;
      sc_out<sc_logic> clear
      sc_signal<sc_logic> lstat;
      sc_signal<sc_logic> down;
      state_machine *s1;
```

```
  SC_CTOR(controller) {
    // .... other module statements
    s1 = new state_machine ("s1");
    // special case port to port binding
    s1->clock(clk);
    // port en bound to signal lstat
    s1->en(lstat);
    // port dir bound to signal down
    s1->dir(down);
    // special case port to port binding
    s1->st(status);
    }
};
```


Clocks

- Special objects in systemc.
- Syntax

  sc_clock clock1("clock1", 20, 0.5, 2, true);
  - Create a clock object named *clock1*
  - Period = 20 time units
  - Duty cycle = 50%,
  - First edge at 2 time units
  - First value true.
- Default values

  period = 1, the duty cycle =0.5, the first edge = 0, first value = true
- Typically created at the top level of the design in the testbench
- Passed down through the module hierarchy to the rest of the design

⇒ areas of the design or the entire design to be synchronized by the same clock

Clock in the main

```
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, 0.5, 0, true);
    filter f1("filter");
    f1.clk(ck1.signal());
    f1.val(val);
    f1.load(load);
    f1.reset(reset);
    f1.out(result);
    // rest of sc_main not shown
}
```