Channel and Interface

Kun-Chih (Jimmy) Chen 陳坤志

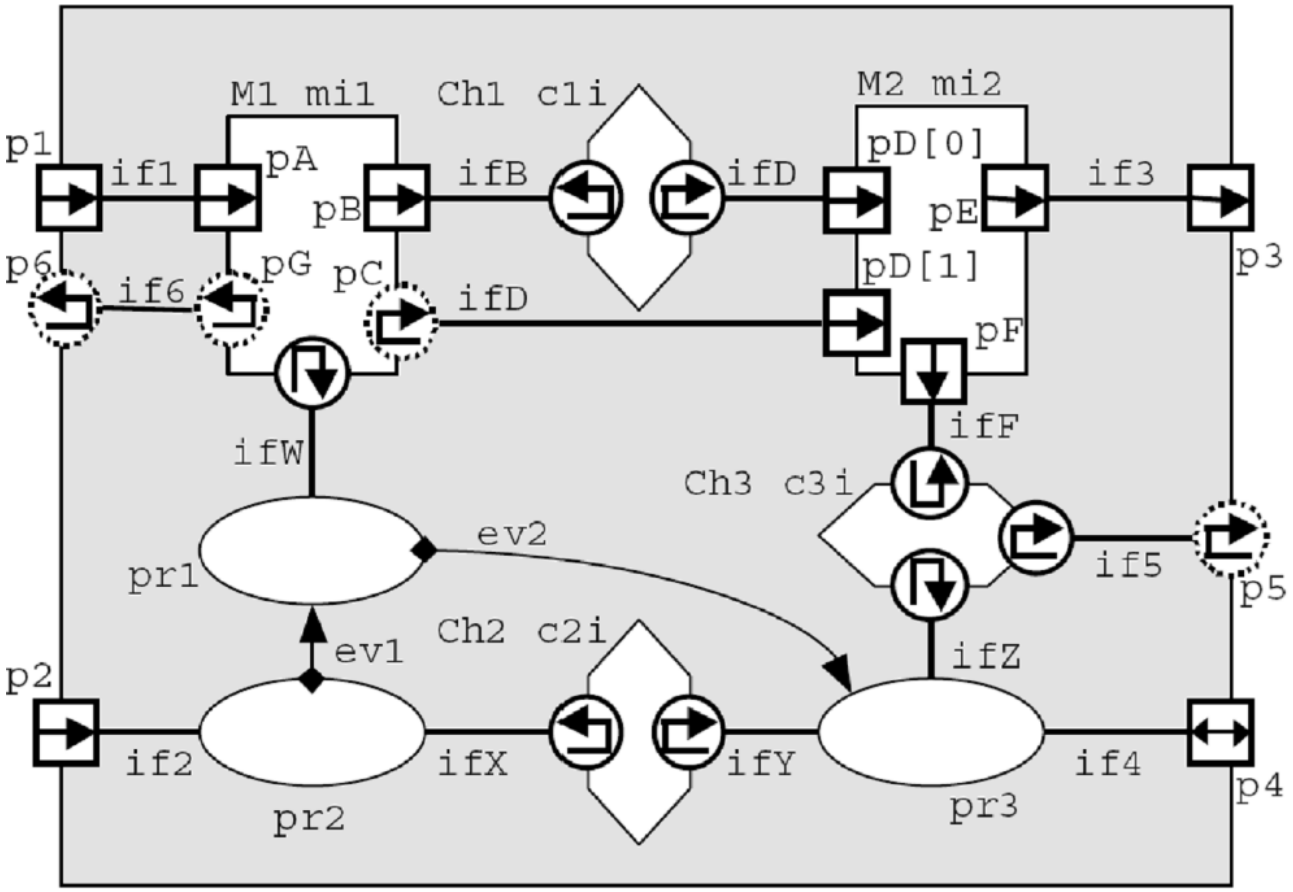
kcchen@nycu.edu.tw

Institute of Electronics,
National Yang Ming Chiao Tung University

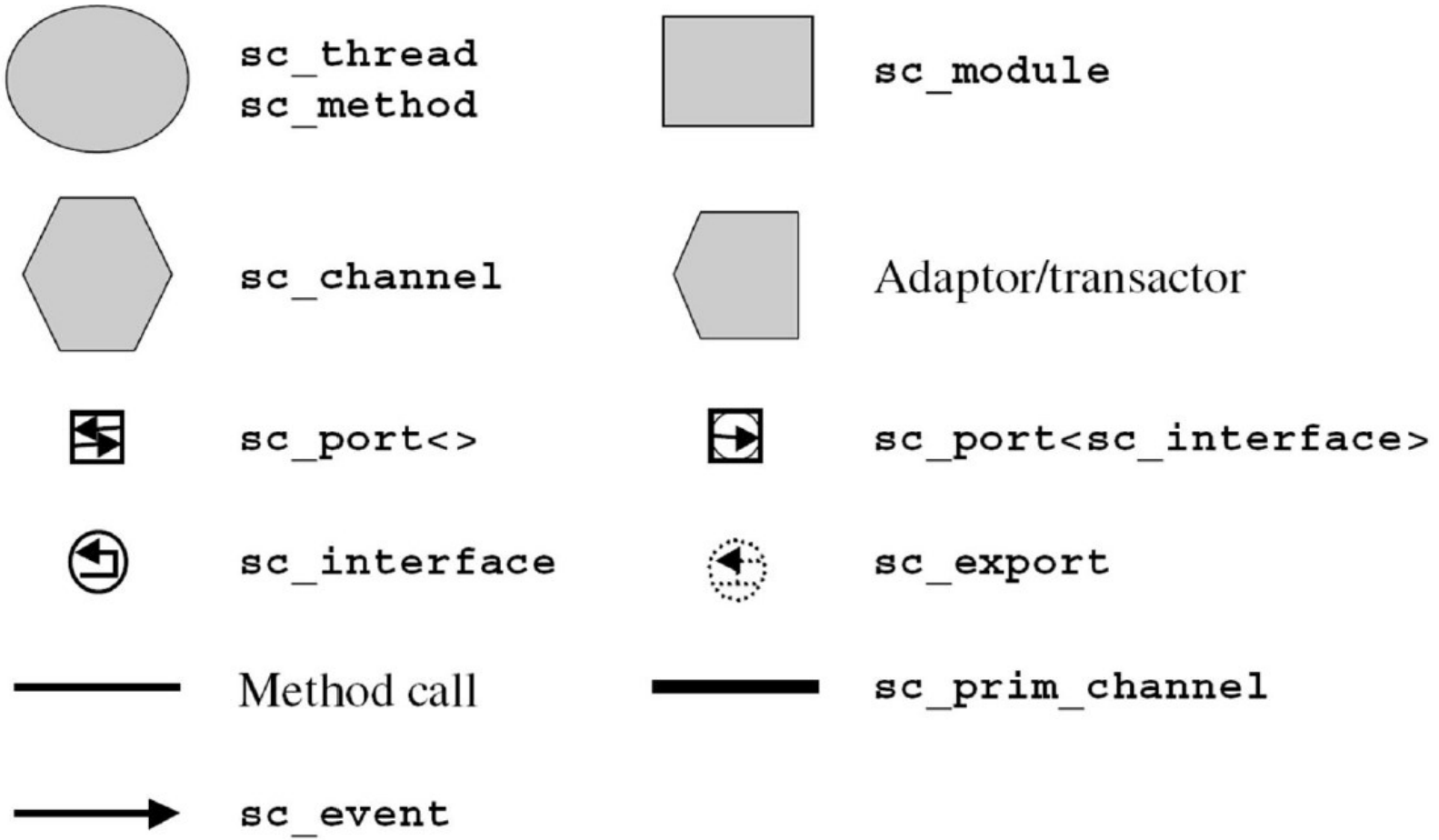
SystemC Communication

- It is necessary to have communication mechanism between each module
- There are three basic “core” SystemC communication mechanism
  - Port
  - Channel
  - Interface
- A SystemC module can connect to another module by mean of a port, specialized for a specific interface, using a channel which implements the methods listed in the interface.

SystemC Communication Mechanism



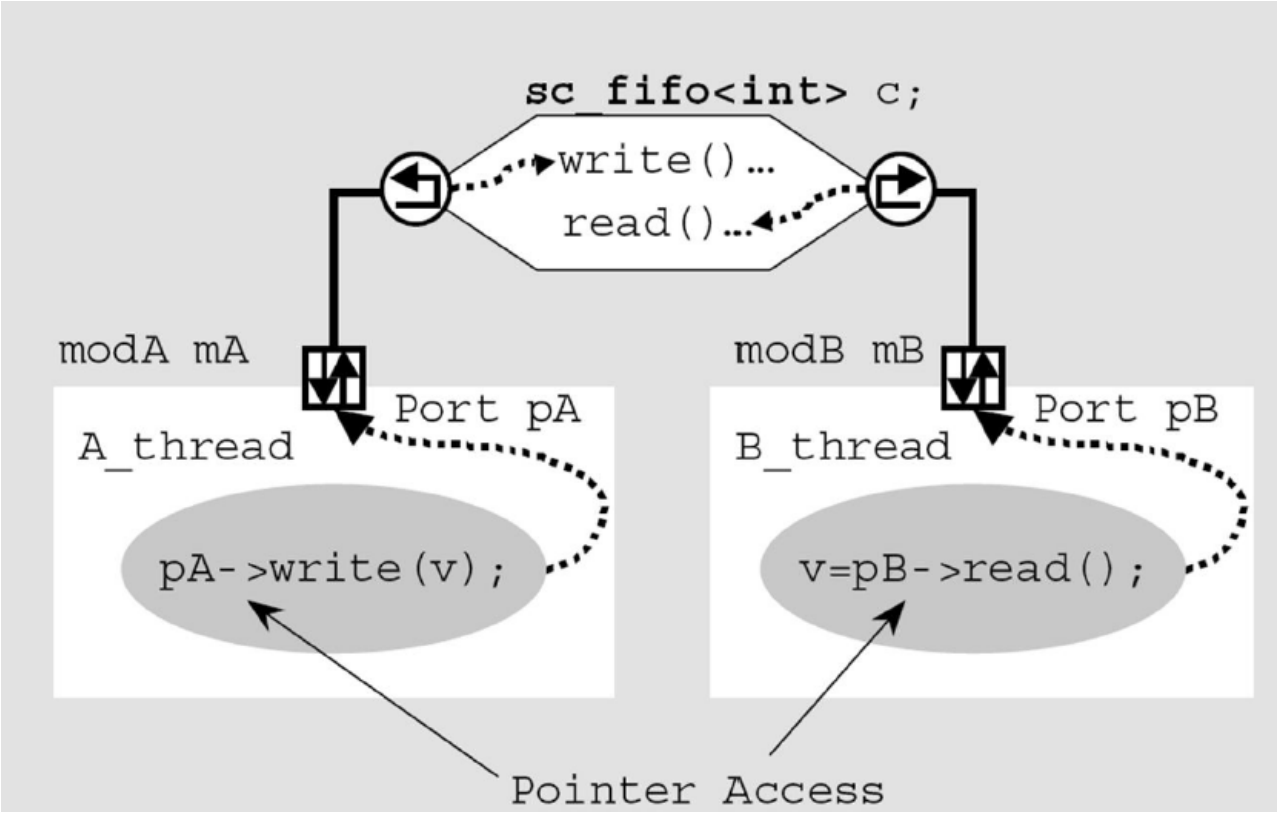
SystemC Communication Mechanism



Port, Channel and Interface

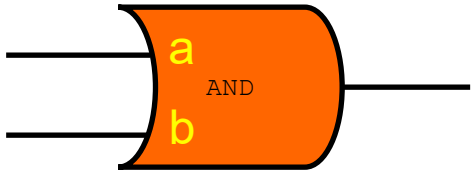
- **Port**
  - A pointer to a channel outside the module
- **Interface**
  - Interface is an abstract class
  - No implementation or data
- **Channel**
  - Implements all the methods of the inherited interface class

Port, Channel and Interface



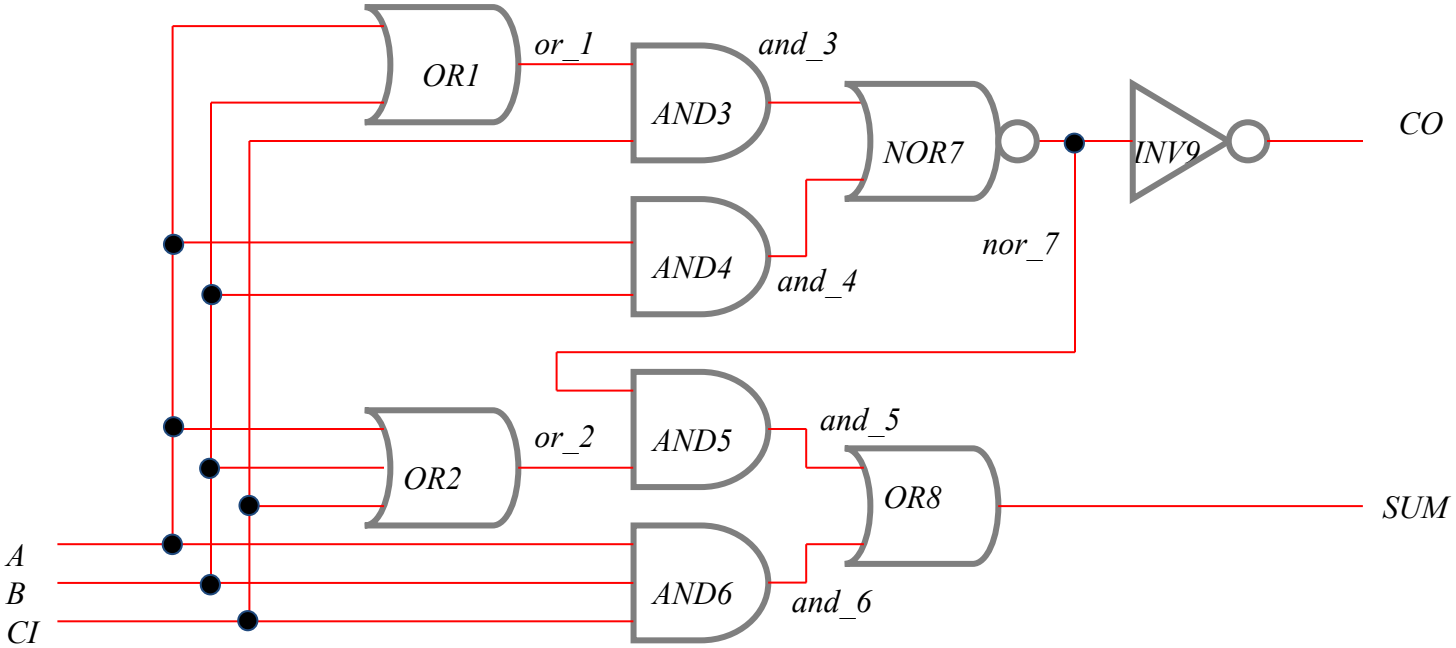
- For safety and ease of use
- A port is a pointer to a channel outside the module

Recap:
A 2-input and-gate class in SystemC




```
#include <systemc.h>

SC_MODULE(AND2)
{
  sc_in<bool> a;  // input pin a
  sc_in<bool> b;  // input pin b

  sc_out<bool> o; // output pin o

  SC_CTOR(AND2)    // the ctor
    {
      SC_METHOD(and_process);
      sensitive << a << b;
    }

  void and_process() {
    o.write( a.read() && b.read() );
  }
};
```

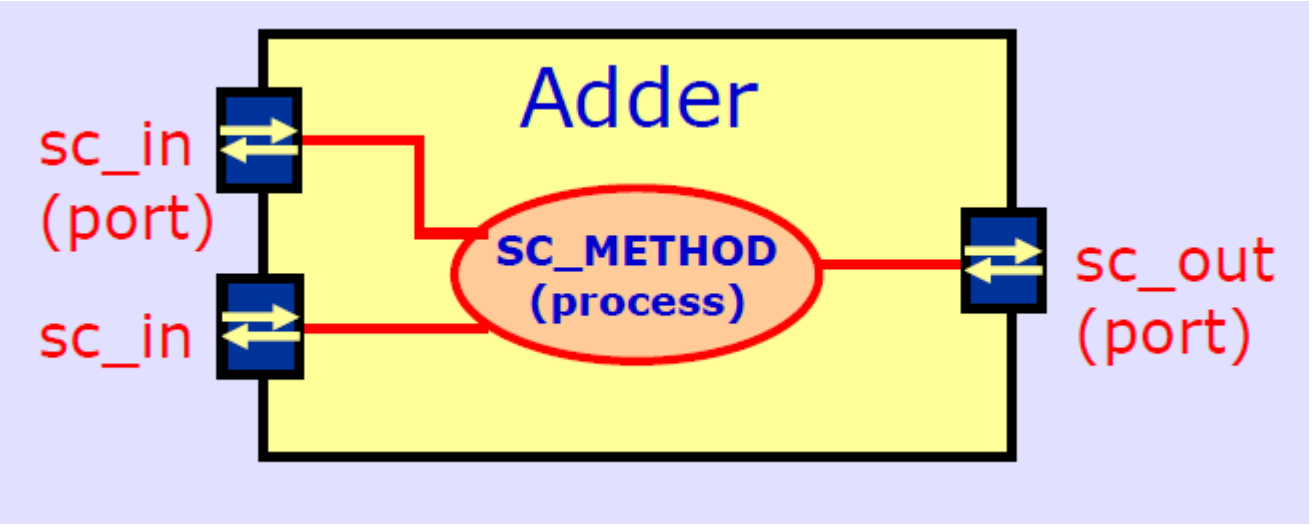
Recap
Connecting pins of gates to signals




```
// 3: Connect the gates to the signal nets
or1.a(A); or1.b(B); or1.o(or_1);
or2.a(A); or2.b(B); or2.c(CI); or2.o(or_2);
and3.a(or_1); and3.b(CI); and3.o(and_3);
and4.a(A); and4.b(B); and4.o(and_4);
and5.a(nor_7); and5.b(or_2); and5.o(and_5);
and6.a(A); and6.b(B); and6.c(CI); and6.o(and_6);
nor7.a(and_3); nor7.b(and_4); nor7.o(nor_7);
or8.a(and_5); or8.b(and_6); or8.o(SUM);
inv9.a(nor_7); inv9.o(CO);
// … continued next page
```

Ports

- Ports are used by modules as a gateway to and from the outside world.
- Must be declared for specific datatype.
- sc_in<type>
- sc_out<type>
- sc_inout<type>
- sc_fifo_in<type>



Port Access

- Components inside module can interact with outside via ports by calling
  - **read()**
  - **write()**

- **read()** method can be called if one port is an incoming port only.

- Same as **write()** method for an outgoing port.

Port Access

- Portname.read()
- Portname.write(value)

```
SC_MODULE(test) {
    sc_in<sc_logic>  in;
    sc_out<sc_logic> out;
    sc_inout<sc_lv<4> > inout;

    void body () {
        if (in.read() == 1) {
            out.write(1);
            inout.write(rand());
        } else {
            out.write("z");
            inout.write("z");
        }
    }

    SC_CTOR(resolve) {
        SC_METHOD(body);
        sensitive << in;
    }
};
```

Channels

- Ports in general do not directly connect to other ports.
  - They are connected through “channels”.
- Channels help modules to communicate with each other.
- In SystemC, there are two kinds of channels:
  - Primitive channels
  - Hierarchical channels

Channels Classify

- Primitive channels (derived from sc_prim_channel)
  - Atomic, doesn't contain other SystemC structures
  - Not exhibit structure
  - Lack of internal processes
  - Can't access (directly) other primitive channels

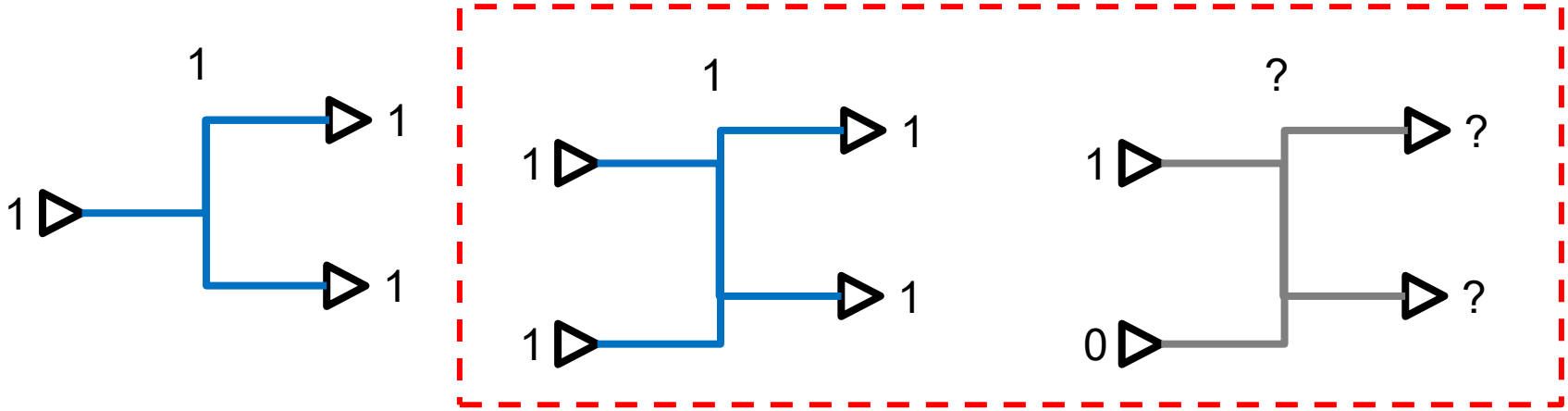
- Hierarchical channels (derived from sc_channel)
  - Construct by module (e.g., BUS)
  - Can have structure
  - Can contain other modules and processes
  - Can access (directly) other channels by using operators,
    ex: fifo1.read() or fifo1.write()

Primitive Channels

- As their name imply, have restrictions in regard to their construction.
- A primitive channel is not allowed to contain SystemC structures for instance threads, methods, other channels etc.
- The SystemC library defines a number of versatile primitive channels :
  - sc_signal<T>
  - sc_signal_resolved
  - sc_signal_rv<W>
  - sc_buffer<T>
  - sc_fifo<T>
  - sc_mutex
  - sc_semaphore

sc_signal<T>

- The simplest channel in SystemC
- Can only connect to one sc_out or sc_inout port (outside modules)
  - Otherwise, multi-driven would occur
- Can be accessed with read()/write() method



sc_signal<T>

```
sc_signal<datatype> signame1,signame2,...;

signame1.write(new_value);
value = signame2.read();
```

Syntax of sc_signal

```
int count;
string message_temp;
sc_signal<int> count_sig;
sc_signal<string> message_sig;

cout << "Initialize during 1st delta cycle" << endl;
count_sig.write(10);
message_sig.write("Hello");
count = 11;
message_temp = "Whoa";
cout << "count is " << count << " "
<< "count_sig is " << count_sig << endl
<< "message_temp is '" << message_temp << "' "
<< "message_sig is '" << message_sig << "'"
<< endl << "Waiting" << endl << endl;
wait(SC_ZERO_TIME);
```

```
cout << "2nd delta cycle" << endl;
count = 20;
count_sig.write(count);
cout << "count is " << count << ", "
<< "count_sig is " << count_sig << endl
<< "message_temp is '" << message_temp << "', "
<< "message_sig is '" << message_sig << "'"
<<endl << "Waiting" << endl << endl;
wait(SC_ZERO_TIME);
```

sc_signal<T>

- count_sig remain unchanged until a delta-cycle has occurred.
- On the other hand, the non-signal values, **count** and **message_temp**, get immediately updated

```
Initialize during 1st delta cycle
count is 11, count_sig is 0
message_temp is 'Whoa', message_sig is ‘’ “
Waiting
2nd delta cycle
count is 20, count_sig is 10
message_temp is 'Whoa', message_sig is 'Hello'
Waiting
```

Result

sc_signal<T>

- SystemC has overloaded the assignment and copy operators
  - It is dangerous to use assignment operator

```
signame = newvalue;// implicit .write() dangerous
varname = signame;// implicit .read() mild danger
```

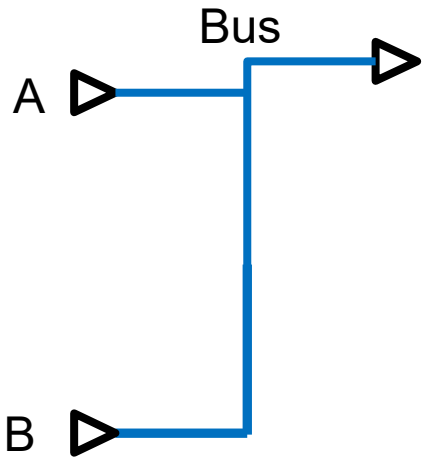
- Assuming that ***r*** is an **sc_signal<*int*>** and x=3, y=4, r=0

```
// Convert rectangular to polar coordinates
r = x;
if ( r != 0 && r != 1 ) r = r * r;
if ( y != 0 ) r = r + y*y;
cout << "Radius is " << sqrt(r) << endl;
```

Radius is 0

sc_signal_resolved

- There are times it is appropriate to have multiple input to an output.
- Have the possibility of high impedance (i.e., Z) and contention (i.e., X)



A\B	‘0’	‘1’	‘X’	‘Z’
‘0’	‘0’	‘X’	‘X’	‘0’
‘1’	‘X’	‘1’	‘X’	‘1’
‘X’	‘X’	‘X’	‘X’	‘X’
‘Z’	‘0’	‘1’	‘X’	‘Z’

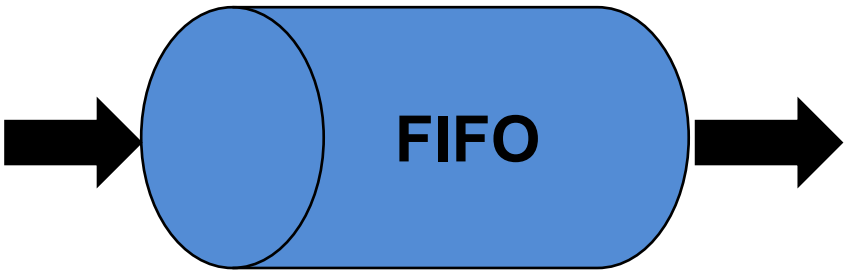
Resolution functionality for **sc_signal_resolved**

sc_signal_resolved

```
SC_MODULE(RESOLVED_SIGNAL) {
  sc_signal_resolved rv;
  vector<sc_logic> levels;
  SC_CTOR(RESOLVED_SIGNAL) : levels(vector<sc_logic>{sc_logic_0, sc_logic_1, sc_logic_Z,
sc_logic_X}){
    SC_THREAD(writer1);
    SC_THREAD(writer2);
    SC_THREAD(consumer);
  }
  void writer1() {
    int idx = 0;
    while (true) {
      rv.write(levels[idx++%4]);
      wait(1, SC_SEC);
    }
  }
  void writer2() {
    int idx = 0;
    while (true) {
      rv.write(levels[(idx++/4)%4]);
      wait(1, SC_SEC);
    }
  }
  void consumer() {
    wait(1, SC_SEC);
    int idx = 0;
    while (true) {
      std::cout << " " << rv.read() << " |";
      if (++idx % 4 == 0) { std::cout << std::endl; }
      wait(1, SC_SEC);
    }
  }
};
```

sc_fifo

- Probably the most popular channel for modeling at the architectural level
- FIFO is short for First-in first-out
  - A common data structure used to manage data flow
  - FIFO is one of the simplest structures to manage
- By default, an **sc_fifo**<T> has a depth of 16



sc_fifo

```
#include <systemc.h>

SC_MODULE(example_fifo) {
    void example_fifo::m_drain_packets(void);
    void example_fifo::t_source1(void);
    void example_fifo::t_source2(void);
    // Constructor
    SC_CTOR(example_fifo) {
        SC_METHOD(m_drain_packets);
        SC_THREAD(t_source1);
        SC_THREAD(t_source2);
        // Size the packet_fifo to 5 ints.
        sc_fifo<int> packet_fifo (5);
    }

    // Declare the FIFO
    sc_fifo<int> packet_fifo;
};
```

```
int sc_main(int argc, char* argv[]) {
    example_fifo ex_fifo ("ex_fifo0");
    sc_start(15, SC_NS);
    return 0;
}
```

sc_fifo

```
void example_fifo::t_source1(void) {
    int val = 1000;
    for (;;) {
        wait(3, SC_NS);
        val++;
        packet_fifo.write(val);
        cout << sc_time_stamp() << ": t_thread1(): Wrote " << val << endl;
    }
}
void example_fifo::t_source2(void) {
    int val = 2000;
    for (;;) {
        wait(5, SC_NS);
        val++;
        packet_fifo.write(val);
        cout << sc_time_stamp() << ": t_thread2(): Wrote " << val << endl;
    }
}

void example_fifo::m_drain_packets(void) {
    int val;
    if (packet_fifo.nb_read(val)) {
        cout << sc_time_stamp() << ": m_drain_packets(): Received " << val
             <<endl;
    }
    else{
        cout << sc_time_stamp() << ": m_drain_packets(): FIFO empty." << endl;
    }
    // Check back in 2ns
    next_trigger(2, SC_NS);
}
```

read() vs. nb_read()

sc_fifo

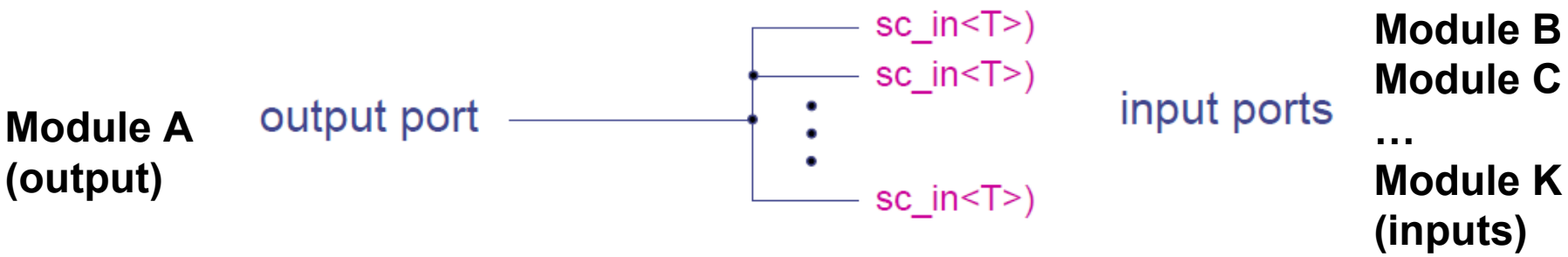
```
0 s: m_drain_packets(): FIFO empty.
2 ns: m_drain_packets(): FIFO empty.
3 ns: t_thread1(): Wrote 1001
4 ns: m_drain_packets(): Received 1001
5 ns: t_thread2(): Wrote 2001
6 ns: m_drain_packets(): Received 2001
6 ns: t_thread1(): Wrote 1002
8 ns: m_drain_packets(): Received 1002
9 ns: t_thread1(): Wrote 1003
10 ns: m_drain_packets(): Received 1003
10 ns: t_thread2(): Wrote 2002
12 ns: m_drain_packets(): Received 2002
12 ns: t_thread1(): Wrote 1004
14 ns: m_drain_packets(): Received 1004
15 ns: t_thread1(): Wrote 1005
15 ns: t_thread2(): Wrote 2003
```

Predefined method in sc_fifo

- **write()**
  - This method write the values passed as an argument into the fifo. If fifo is full, write() function waits till fifo slot is available
- **nb_write()**
  - This method is same as write(), only difference is, when fifo is full nb_write() does not wait till fifo slot is avaiable. Rather it returns false.
- **read()**
  - This method returns the least recent written data in fifo. If fifo is empty, then read() function waits till data is available in fifo.
- **nb_read()**
  - This method is same as read(), only difference is, when fifo is empty, nb_read() does not wait till fifo has some data. Rather it returns false.
- **num_available()**
  - This method returns the numbers of data values available in fifo in current delta time.
- **num_free()**
  - This method returns the number of free slots available in fifo in current delta time.

Channel Design Rule

sc_signal<T>	At most one output (sc_out<T>) or bi-directional port (sc_inout<T>) can connect
	Arbitrary number of input port (sc_in<T>) can connect
sc_fifo<T>	At most one input port (sc_in<T>) can connect
	At most one output port (sc_out<T>) can connect
	No bi-directional ports



sc_buffer<T>

- sc_buffer<T> is similar to sc_signal<T>
- sc_signal<T> first check the new data whether is the same as old data
  - Update the data when the new data is different from the old data
- sc_buffer<T> update the data regardless of whether the data is the same as the old data

sc_buffer

- Receiver and Sender

```
SC_MODULE(Transmitter) {

    sc_out<char> out;

    void transmit() {
        wait(1, SC_NS);
        out.write('x');

        wait(1, SC_NS);
        out.write('x');

        wait(1, SC_NS);
        out.write('y');
    };

    SC_CTOR (Transmitter){
     SC_THREAD(transmit);
    }


};
```

```
SC_MODULE(Receiver){

    sc_in<char> in;

    void receive(){
        cout << sc_time_stamp() << ": " << name() << " received "
             << in.read() << endl;
    }

    SC_CTOR(Receiver){
        SC_METHOD(receive);
        sensitive << in;
        dont_initialize();
    }

};
```

sc_buffer

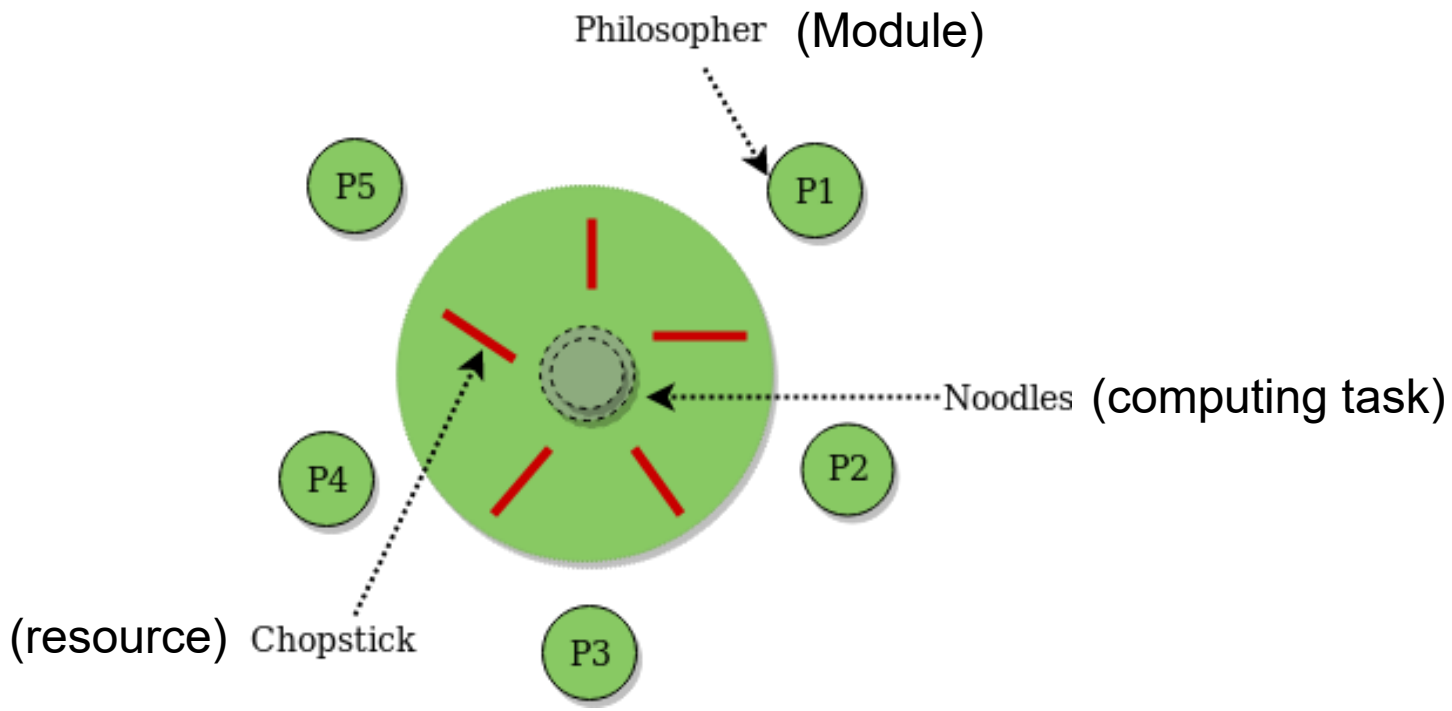
- Signal_receiver did not detect 2ns sent characters

```
int sc_main(int argc, char* argv[])
{
    sc_signal<char> signal;
    sc_buffer<char> buffer;

    Transmitter signal_transmitter("signal_transmitter");
    Receiver signal_receiver("signal_receiver");
    Transmitter buffer_transmitter("buffer_transmitter");
    Receiver buffer_receiver("buffer_receiver");

    signal_transmitter.out(signal);
    signal_receiver.in(signal);

    buffer_transmitter.out(buffer);
    buffer_receiver.in(buffer);

    sc_start();

    return 0;
}
```

1 ns: signal_receiver received x
1 ns: buffer_receiver received x
2 ns: buffer_receiver received x
3 ns: signal_receiver received y
3 ns: buffer_receiver received y

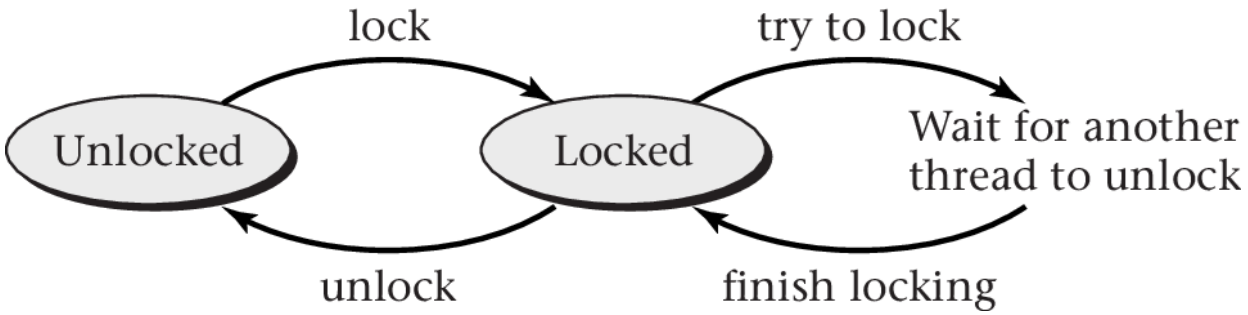
How to describe the limited hardware resource?

- Similar to the **Dining Philosopher Problem**
  - The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him.



sc_mutex

- Mutex is short for mutually exclusive text
  - Multiple program threads share a common resource
- Any process that needs the resource must lock the mutex to prevent other processes from using the shared resource
- Member functions
  - lock(): lock the mutex (wait until unlocked if in use)
  - trylock(): non-blocking, return true if success, else false
  - unlock(): free previously locked mutex



sc_mutex

- SystemC provide the mutex function via sc_mutex channel

```
sc_mutex NAME;

NAME.lock(); //Lock the mutex
             //wait until unlocked if in use
int NAME.trylock() // Non-blocking, return success

NAME.unlock(); //Free a previously locked mutex
```

```
class car : public sc_module {
    sc_mutex drivers_seat;
    public:
    void drive_thread(void);
    …
};
void car::drive_thread(void) {
    drivers_seat.lock(); // sim driver acquires seat
    start();
    … // operate vehicle
    stop();
    drivers_seat.unlock(); // sim driver leaves
    // vehicle
    …
}
```

sc_semaphore

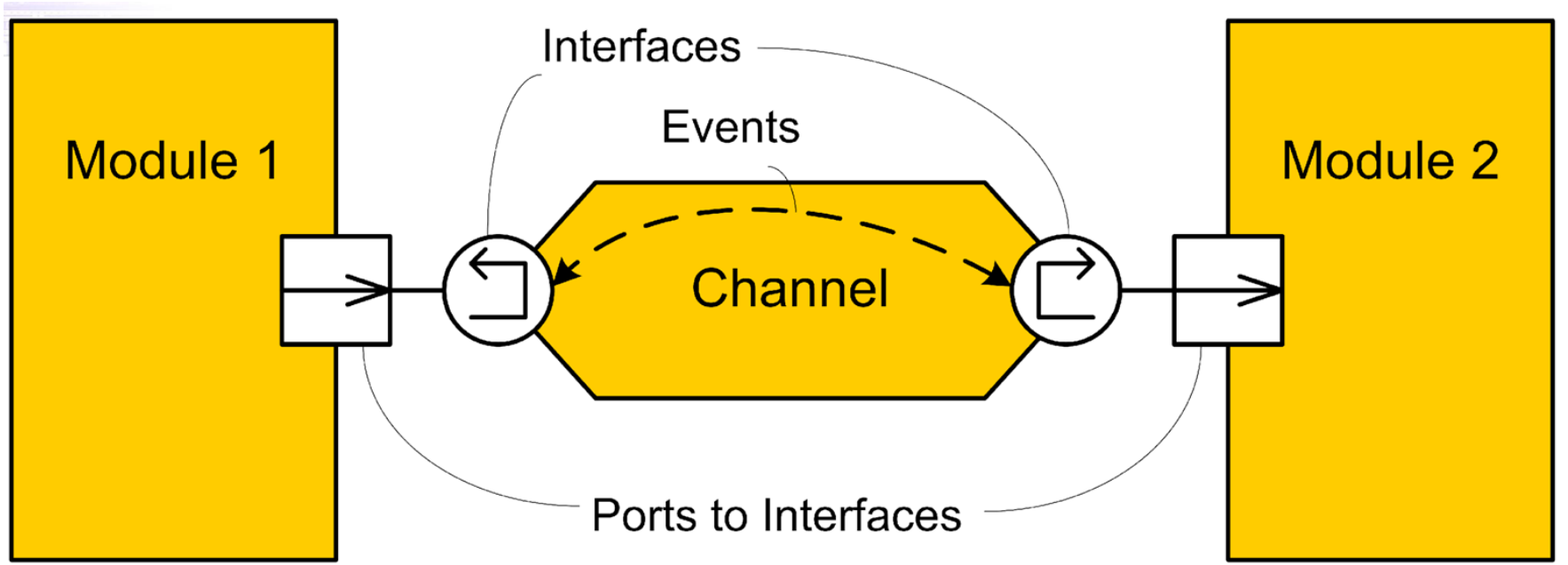
- For some resources, you may want to model more than one copy or owner
  - To manage this type of resource, SystemC provides the **sc_semaphore**
- When creating an **sc_semphore** object, it is necessary to specify how many are available
- An **sc_semaphore** access consists of waiting for an available resource and then posting notice when finished with the resource

sc_semaphore

```
sc_semaphore NAME(COUNT);
NAME.wait(); // Lock one semaphore
// Wait until available if in use
int NAME.trywait() // Non-blocking, return success
int NAME.get_value() // Returns available semaphores
NAME.post(); // Free one previously locked
```

```
SC_MODULE(gas_station) {
    sc_semaphore pump(12);
    void customer1_thread {
    for(;;) {
        // wait till tank empty
        …
        // find an available gas pump
        pump.wait();
        // fill tank & pay
    }
};
```

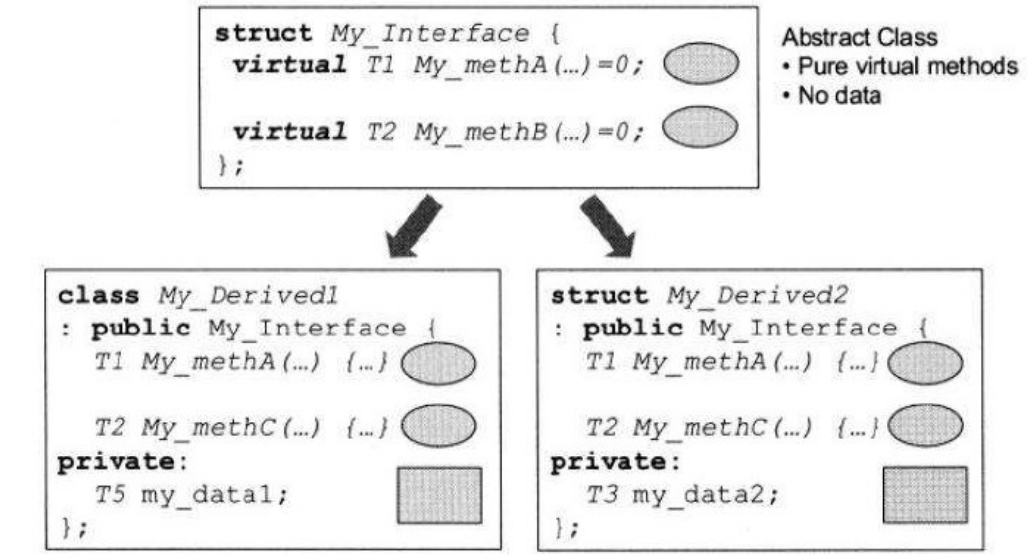
What is Interface and Channel



Interface

- Define methods that channels must implement

- All interfaces must be derived from base class sc_inferface
  - Ports connect to channels through interfaces
  - Ports only can see the channels' method defined in interface which connect to
  - sc_fifo_in_if
  - sc_fifo_out_if
  - sc_mutex_if
  - sc_semaphore_if
  - sc_signal_in_if
  - sc_signal_out_if



Interface Derivation

- Define read/write interface by deriving from read interface and write interface.

```
template <class T>
class sc_read_write_if: public sc_read_if<T>, public sc_write_if<T>
{

};
```

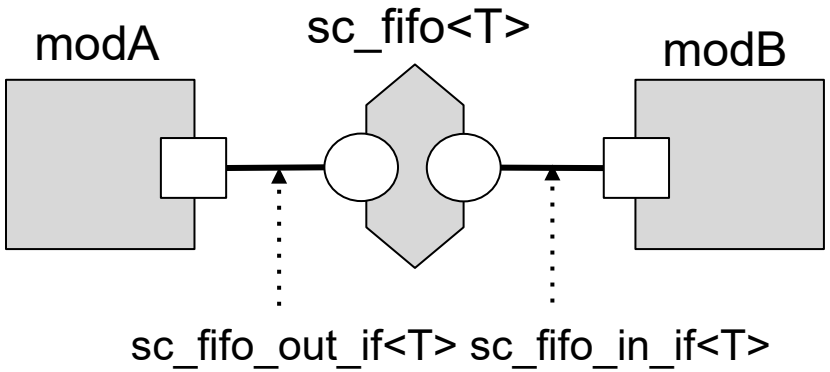
- Interfaces for sc_fifo channel
  - **sc_fifo_in_if**: read(), nb_read(), … methods
    - read(): if not empty, read; if empty, wait until available and then read
    - nb_read(): returns false, when fifo is empty (does not wait); otherwise reads and returns true
  - **sc_fifo_out_if**: write(), nb_write(), … methods

Interface & Channel

- A SystemC interface is an abstract class that inherits from sc_interface
- Provides only pure virtual declarations of methods referenced by SystemC channels and ports
- No implementations or data are provided in SystemC interface
- A SystemC channel is a class that implements one or more SystemC interface classes and inherits from either sc_channel or sc_prim_channel
- A channel implements all the methods of the inherited interface classes

Interface & Channel

- We can implement modules independent of the implementation details of the communication channels



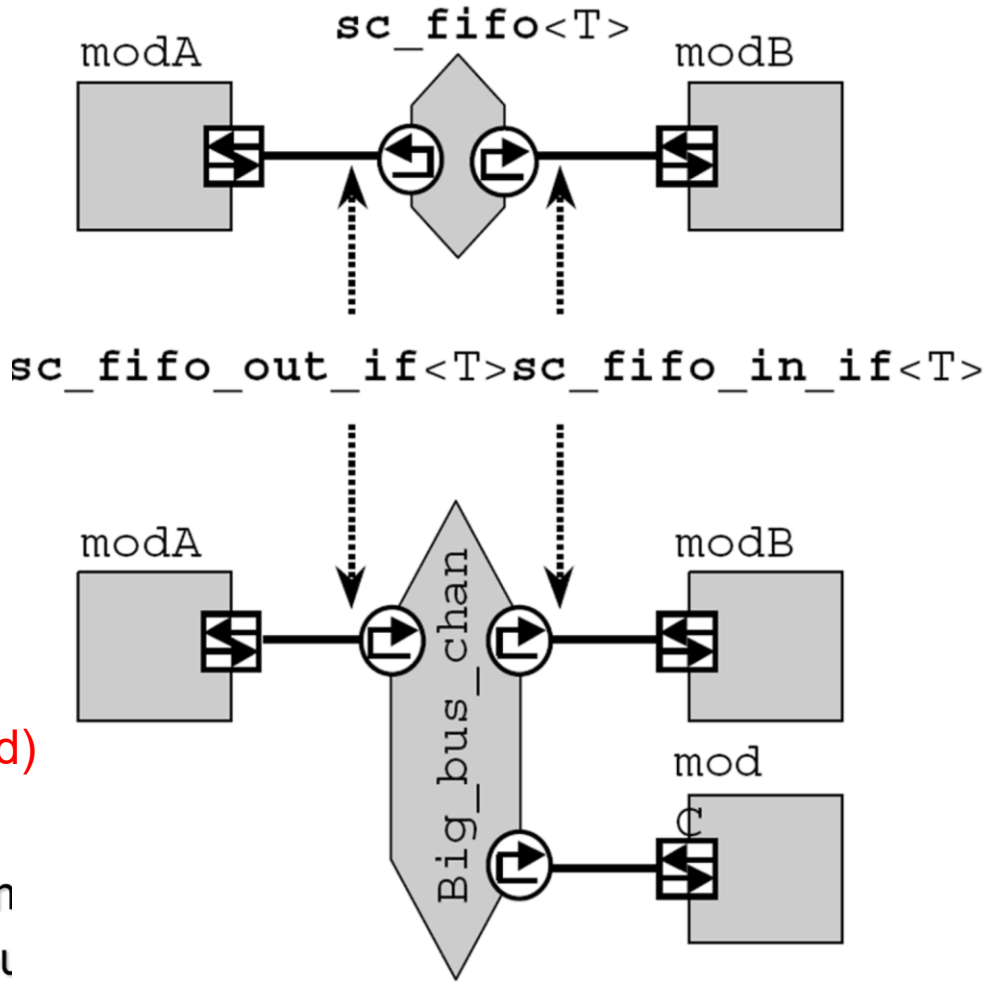
- With no change to the definition of modA or modB, we can swap out the FIFO for a different channel

- **Note:**
  - sc_fifio_in_if<T> is used for “reading” from the FIFO
  - sc_fifio_out_if<T> is used for “writing” from the FIFO

FIFO Interface

- **sc_fifo_in_if**<T> and **sc_fifo_out_**if<T> are provided for the **sc_fifo**<T> channel
- **sc_fifo_out_if**<T>
  - Provides all the methods for output from a module into an **sc_fifo**<*T*>
  - The module pushes data onto the FIFO using **write()**or **nb_write()**
- **sc_fifo_in_if**<T>
  - Provides all the methods for input from a module into an **sc_fifo**<*T*>
  - The module pulls data from the FIFO using **read()**or **nb_read()**

FIFO Interface



sc_port *<interface> portname*;

Example:

space (old)

```
SC_MODULE(stereo_amp) {
    sc_port<sc_fifo_in_if<int> > soun
    sc_port<sc_fifo_out_if<int> > sou
    …
};
```

sc_signal interfaces

- This definition lets a module read the value of an output signal directly rather than being forced to keep a local copy
- Similar to **sc_fifo**<*T*>, **sc_signal_in_if**<*T*>, **sc_signal_out_if**<*T*>, and **sc_signal_inout_if**<*T*>, are provided for the **sc_signal**<*T*>channel
- Use the **sc_signal_inout_if**<*T*> when **sc_signal**<*T*> is needed
- The **sc_signal_in_if**<*T*> interface provides access to the results through **sc_signal**<*T*>::read()
- The **sc_signal_out_if**<*T*> interface provides access to the results through **sc_signal**<*T*>::write()

Example of port interconnection setup (1/2)

```
//FILE: Rgb2YCrCb.h
SC_MODULE(Rgb2YCrCb) {
  sc_port<sc_fifo_in_if<RGB_frame> >     rgb_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > ycrcb_po;
};
```

```
//FILE: YCRCB_Mixer.h
SC_MODULE(YCRCB_Mixer) {
  sc_port<sc_fifo_in_if<float> >         K_pi;
  sc_port<sc_fifo_in_if<YCRCB_frame> >   a_pi, b_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > y_po;
};
```

Example of port interconnection setup (2/2)

```
//FILE: VIDEO_Mixer.h
SC_MODULE(VIDEO_Mixer) {
  // ports
  sc_port<sc_fifo_in_if<YCRCB_frame> >  dvd_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > video_po;
  sc_port<sc_fifo_in_if<MIXER_ctrl> >   control;
  sc_port<sc_fifo_out_if<MIXER_state> > status;
  // local channels
  sc_fifo<float>        K;
  sc_fifo<RGB_frame>    rgb_graphics;
  sc_fifo<YCRCB_frame>  ycrcb_graphics;
  // local modules
  Rgb2YCrCb   Rgb2YCrCb_i;
  YCRCB_Mixer YCRCB_Mixer_i;
  // constructor
  VIDEO_Mixer(sc_module_name nm);
  void Mixer_thread();
};
```

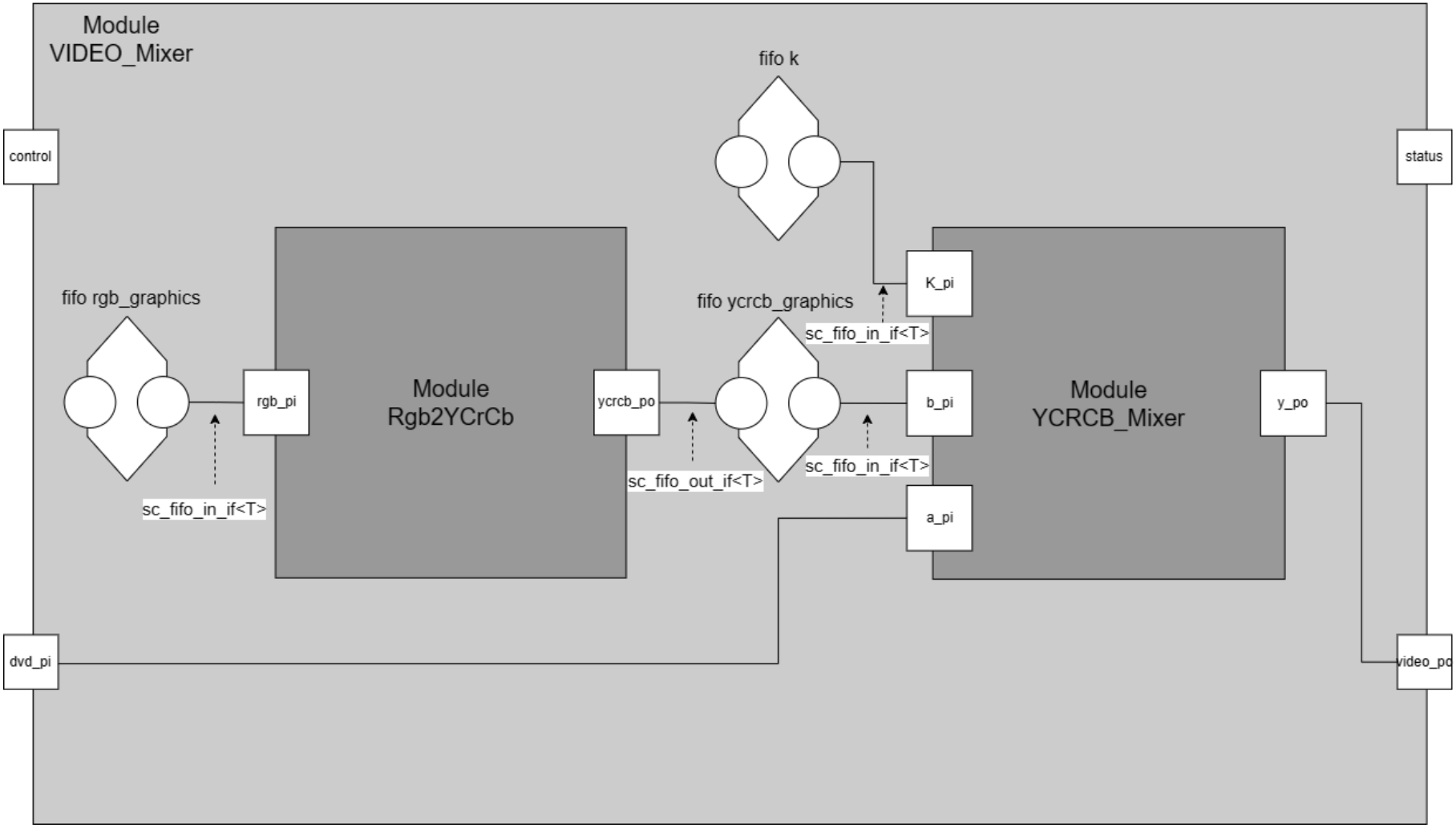
Example of port interconnection (1/3)

```
SC_HAS_PROCESS(VIDEO_Mixer);
VIDEO_Mixer::VIDEO_Mixer(sc_module_name nm)
: sc_module(nm)
, Rgb2YCrCb_i("Rgb2YCrCb_i")
, YCRCB_Mixer_i("YCRCB_Mixer_i")
{
  // Connect
  Rgb2YCrCb_i.rgb_pi(rgb_graphics);
  Rgb2YCrCb_i.ycrcb_po(ycrcb_graphics);
  YCRCB_Mixer_i.K_pi(K);
  YCRCB_Mixer_i.a_pi(dvd_pi);
  YCRCB_Mixer_i.b_pi(ycrcb_graphics);
  YCRCB_Mixer_i.y_po(video_po);
}
```

Example of port interconnection (2/3)

```
SC_HAS_PROCESS(VIDEO_Mixer);
VIDEO_Mixer::VIDEO_Mixer(sc_module_name nm)
 : sc_module(nm)
 {
   // Instantiate
   Rgb2YCrCb_iptr   = new Rgb2YCrCb(
                            "Rgb2YCrCb_i"
                          );
   YCRCB_Mixer_iptr = new YCRCB_Mixer(
                            "YCRCB_Mixer_i"
                          );
   // Connect
   (*Rgb2YCrCb_iptr)( rgb_graphics
                     ,ycrcb_graphics
                    );
   (*YCRCB_Mixer_iptr)( K
                       ,dvd_pi
                       ,ycrcb_graphics
                       ,video_po
                      );
 }
```

Example of port interconnection (3/3): Block Diagram



Static Sensitivity when used with ports

- Sensitivity to events:
  - an SC_METHOD statically sensitive to the **data_written_event**
  - monitor an **sc_signal** for any change in the data using the **value_changed_event**
    - all are defined in C++ kernel
- Ports are pointers initialized during the elaboration stage
- They are not defined when the sensitive method needs to know about them

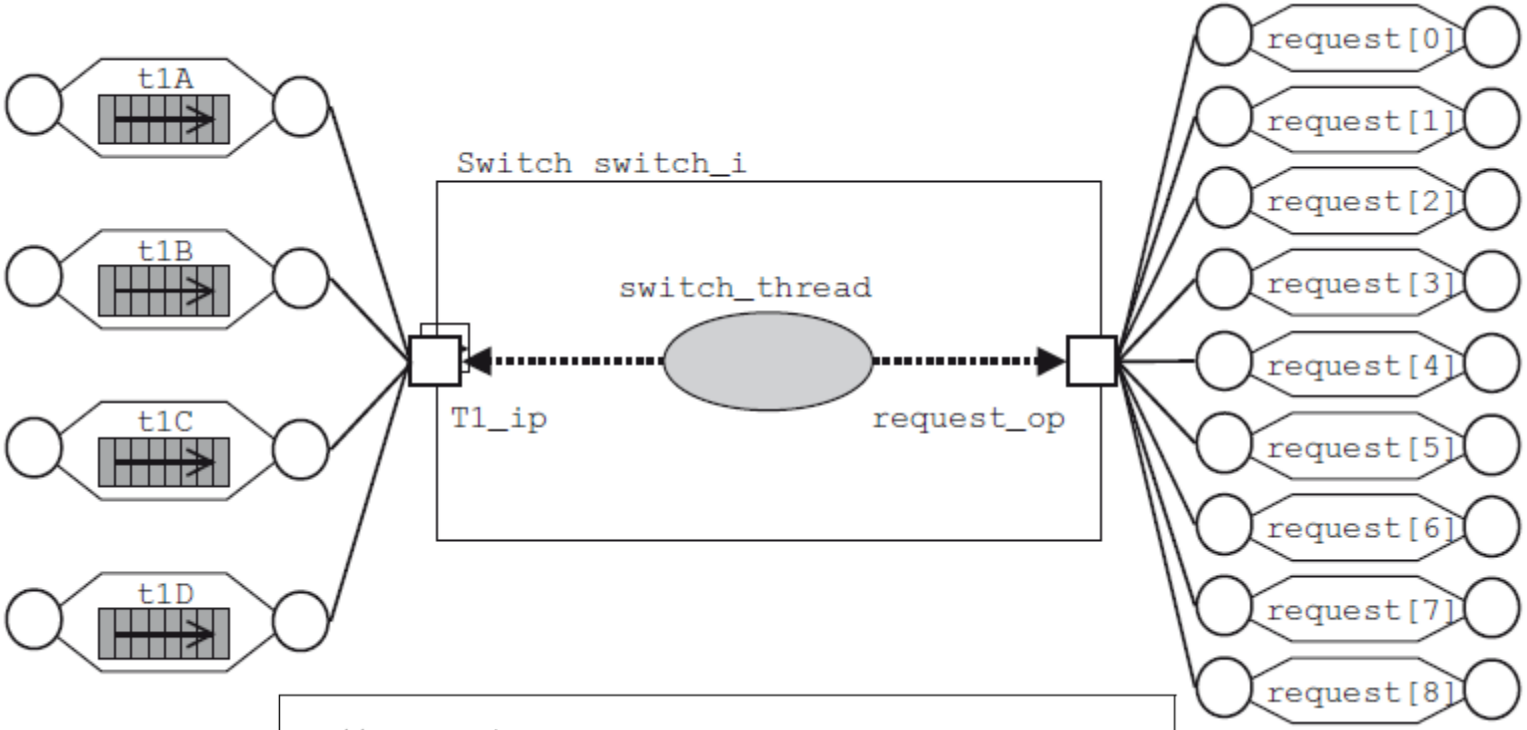
Port Array and Port Policy

- The sc_port<T> provides additional template
  - Array size parameter
  - Port policy parameter
- The array size parameter allows the creation of a number of identical ports
  - This construct is referred to as a multi-port or port array
- The port policy specifies whether zero, one, or all ports must be connected
  - SC_ONE_OR_MORE_BOUND (default)
  - SC_ZERO_OR_MORE_BOUND
  - SC_ALL_BOUND

```
sc_port<interface, N , POL > portname;
//N=0..MAX Default N=1
//POL is of type sc_port_policy
//POL defaults to SC_ONE_OR_MORE_BOUND
```

Port Array and Port Policy

- Four distinct channels connected to four ports T1_ip[0…3] on the left, and on the right, nine separate channel connect to nine ports request[0…8]




```
//FILE: Switch.h
SC_MODULE(Switch) {
  sc_port<sc_fifo_in_if<int>,  4> T1_ip;
  sc_port<sc_signal_out_if<bool>,9> request_op;
  ...
};
```

Port Array and Port Policy

```
//FILE: Board.h
#include "Switch.h"
SC_MODULE(Board) {
Switch switch_i;
sc_fifo<int> t1A, t1B, t1C, t1D;
sc_signa<l bool> request[9];
    SC_CTOR(Board): switch_i("switch_i"){
        // Connect 4 T1 channels to the switch
        switch_i.T1_ip(t1A);
        switch_i.T1_ip(t1B);
        switch_i.T1_ip(t1C);
        switch_i.T1_ip(t1D);
        // Connect 9 request channels to the
        // switch request output ports
        for (unsigned i=0;i!=9;i++) {
            switch_i.request_op(request[i]);
        }//endfor
    ...
    }//end constructor
...
};
```

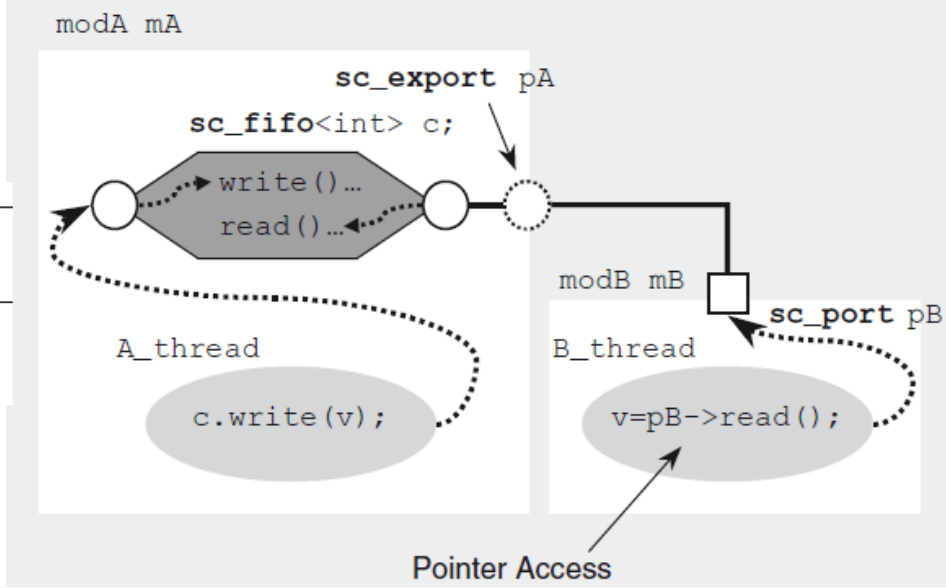
```
//FILE: Switch.cpp
void Switch::switch_thread() {
    // Initialize requests
    for (unsigned i=0;i!=request_op.size();i++){
        request_op[i]->write(true);
    }//endfor
    // Startup after first port is activated
    wait(T1_ip[0]->data_written_event()
    |T1_ip[1]->data_written_event()
    |T1_ip[2]->data_written_event()
    |T1_ip[3]->data_written_event()
    );
    while(true) {
    for (unsigned i=0;i!=T1_ip.size();i++) {
        // Process each port...
        int value = T1_ip[i]->read();
        }//endfor
    }//endwhile
}//end Switch::switch_thread
```

sc_export

- A new type in SystemC 2.1
- There is a second type of port called the **sc_export**<*T*>
- The idea of an **sc_export**<*T*> is to move the channel inside the defining module
  - Hiding some of the connectivity details and using the port externally as though it were a channel

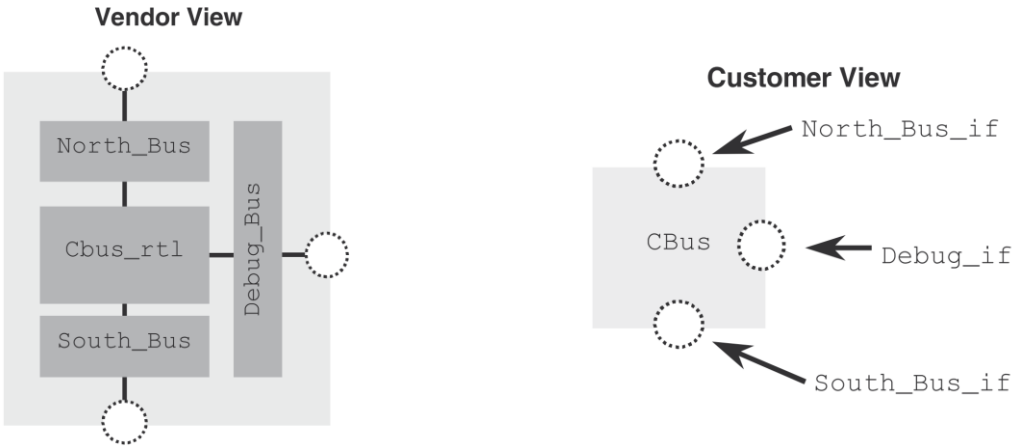
```
sc_export<interface> portname;
```

Syntax of **sc_export** declaration



Why sc_export <T>

- For an IP provider, it may be desirable to export only specific channels and keep everything else private
  - **sc_export <T>** allows control over the interface



- Another reason for using **sc_export**<*T*> is to provide multiple interfaces at the top level

Example 1: sc_export

```
SC_MODULE(clock_gen) {
    sc_export<sc_signal<bool>> clock_xp;
    sc_signal<bool> oscillator;
    SC_CTOR(clock_gen) {
        SC_METHOD(clock_method);
        clock_xp(oscillator); // connect sc_signal
        // channel
        // to export clock_xp
        oscillator.write(false);
    }
    void clock_method() {
    oscillator.write(!oscillator.read());
    next_trigger(10,SC_NS);
}
```

```
#include "clock_gen.h"
…
clock_gen clock_gen_i("clock_gen_i");
collision_detector cd_i("cd_i");
// Connect clock
cd_i.clock(clock_gen_i.clock_xp);
```

sc_port

- A SystemC port is a class templated with and inheriting from a SystemC interface

sc_port<interface> portname;

- SystemC ports are always defined within the module class definition.
- Ports allow access of channels across module boundaries

```
SC_MODULE(stereo_amp) {
    sc_port<sc_fifo_in_if<int> > soundin_p;
    sc_port<sc_fifo_out_if<int> > soundout_p;
    …
};
```

Port Connection Mechanics

- There are two syntaxes for connecting ports
  - by name
  - by position

```
mod_inst.portname(channel_instance); // Named
mod_instance(channel_instance,…); // Positional
```

- The problem with positional connectivity is that of keeping the ordering correct.
  - Using a positional notation can quickly lead to debug problems.

Port Connection Mechanics

- Although slightly more code than the positional notation, the named port syntax is more robust

```
// Instantiate
Rgb2YCrCb_iptr = new Rgb2YCrCb(
    "Rgb2YCrCb_i"
);
YCRCB_Mixer_iptr = new YCRCB_Mixer(
    "YCRCB_Mixer_i"
);
// Connect
(*Rgb2YCrCb_iptr)( rgb_graphics
    ,ycrcb_graphics
);
(*YCRCB_Mixer_iptr)( K
    ,dvd_pi
    ,ycrcb_graphics
    ,video_po
);
```

By Position

```
// Connect
Rgb2YCrCb_i.rgb_pi(rgb_graphics);
Rgb2YCrCb_i.ycrcb_po(ycrcb_graphics);
YCRCB_Mixer_i.K_pi(K);
YCRCB_Mixer_i.a_pi(dvd_pi);
YCRCB_Mixer_i.b_pi(ycrcb_graphics);
YCRCB_Mixer_i.y_po(video_po);
```

By Name

Accessing Ports From Within a Process

- The sc_port overload the c++ operator->()

```
portname->method(optional_args);
```

syntax

- Example

```
void VIDEO_Mixer::Mixer_thread() {
    …
    switch (control->read()) {
    case MOVIE: K.write(0.0f); break;
    case MENU: K.write(1.0f); break;
    case FADE: K.write(0.5f); break;
    default: status->write(ERROR); break;
    }
    …
}
```

Ways to Interconnect

From	To	Method
Port	Sub-module	Direct connect via `sc_port`
Process	Port	Direct access by process
Sub-module	Sub-module	Local channel connection
Process	Sub-module	Local channel connection –or- via `sc_export` –or- interface implemented by sub-module[31]
Process	Process	Events or local channel
Port	Local channel	Direct connect via `sc_export`

SystemC Communication Mechanism