
# *Version 1.1*
# *User's Guide*

**Synopsys, Inc.**

**CoWare, Inc.**

**Frontier Design, Inc.**

**Bugs and Suggestions**

Please report bugs and suggestions about this document by going to

http://www.systemc.org/feedback

# *Contents*

CHAPTER 1

# Introduction

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

C or C++ are the language choice for software algorithm and interface specifications because they provide the control and data abstractions necessary to develop compact and efficient system descriptions. Most designers are familiar with these languages and the large number of development tools associated with them.

The SystemC Class Library provides the necessary constructs to model system architecture including hardware timing, concurrency, and reactive behavior that are missing in standard C++. Adding these constructs to C would require proprietary extensions to the language, which is not an acceptable solution for the industry. 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.

If you are familiar with the C++ programming language, you can learn to program with SystemC by understanding the additional semantics introduced by the SystemC classes; no additional syntax has to be learned. If you are one of the many that are more familiar with the C programming language, you need to learn some C++ syntax in addition to the semantics introduced by the classes. The use of C++ has been kept to a minimum in SystemC. If you are familiar with the Verilog and VHDL hardware description languages and the C programming language, learning SystemC will be easy.

This document describes how to use the SystemC Class Library version v1.0 to create an executable specification for your system-level designs.

## *Using Executable Specifications*

There are many benefits to creating an accurate executable specification of your complex system at the beginning of your design flow. These benefits are

- An executable specification avoids inconsistency and errors and helps ensure completeness of the specification. This is because in creating an executable specification, you are essentially creating a program that behaves the same way as the system. The process of creating the program unearths inconsistencies and errors, and the process of testing the program helps ensure completeness of the specification.
- An executable specification ensures unambiguous interpretation of the specification. Whenever implementers are in doubt about the design, they can run the executable specification to determine what the system is supposed to be doing.
- An executable specification helps validate system functionality before implementation begins.
- An executable specification helps create early performance models of the system and validate system performance.
- The testbench used to test the executable specification 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.

## *SystemC Highlights*

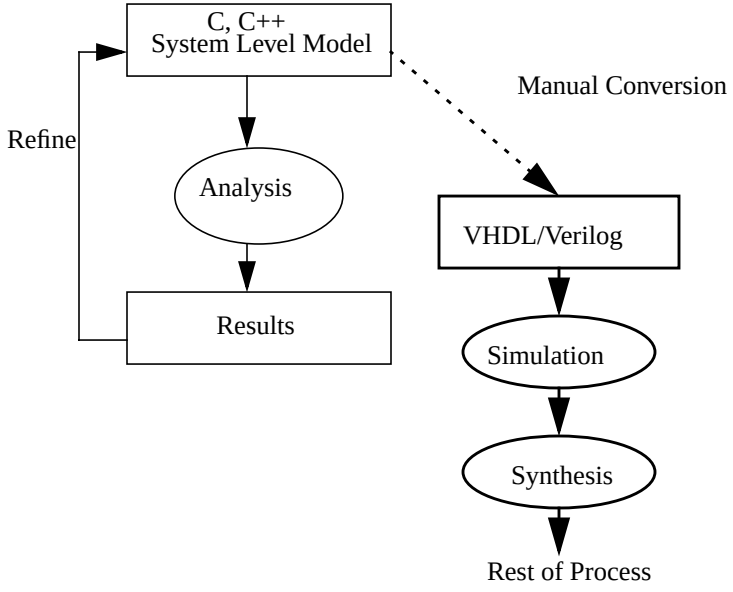
SystemC supports hardware-software co-design and the description of the architecture of complex systems consisting of both hardware and software components. It supports the description of hardware, software, and interfaces in a C++ environment. The following features of SystemC version v1.0 allow it to be used as a co-design language:

- Modules: SystemC has a notion of a container class called a module. This is a hierarchical entity that can have other modules or processes contained in it.
- Processes: Processes are used to describe functionality. Processes are contained inside modules. SystemC provides three different process abstractions to be used by hardware and software designers.
- Ports: Modules have ports through which they connect to other modules. SystemC supports single-direction and bidirectional ports.
- Signals: SystemC supports resolved and unresolved signals. Resolved signals can have more than one driver (a bus) while unresolved signals can have only one driver.
- Rich set of port and signal types: To support modeling at different levels of abstraction, from the functional to the RTL, SystemC supports a rich set of port and signal types. This is different than languages like Verilog that only support bits and bit-vectors as port and signal types. SystemC supports both two-valued and four-valued signal types.
- Rich set of data types: SystemC has a 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 supports both two-valued and four-valued data types. There are no size limitations for arbitrary precision SystemC types.
- Clocks: SystemC has the notion of clocks (as special signals). Clocks are the timekeepers of the system during simulation. 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.
- 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. It supports iterative refinement of high level models into lower levels of abstraction.

- Communication protocols: SystemC provides multi-level communication semantics that enable you to describe SoC and system I/O protocols with different levels for abstraction.
- Debugging support: SystemC classes have run-time error checking that can be turned on with a compilation flag.
- Waveform tracing: SystemC supports tracing of waveforms in VCD, WIF, and ISDB formats.

## *Current System Design Methodology*

The current system design methodology starts with a system engineer writing a C or C++ model of the system to verify the concepts and algorithms at the system level. After the concepts and algorithms are validated, 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. This is shown in the figure below:



There are a number of problems with this approach.

### Manual Conversion from C to HDL Creates Errors

With the current methodology, the designer creates the C model, verifies that the C model works as expected, and then translates the design manually into an HDL. This process is very tedious and error prone.

### Disconnect Between System Model and HDL Model

After the model is converted to HDL, 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.

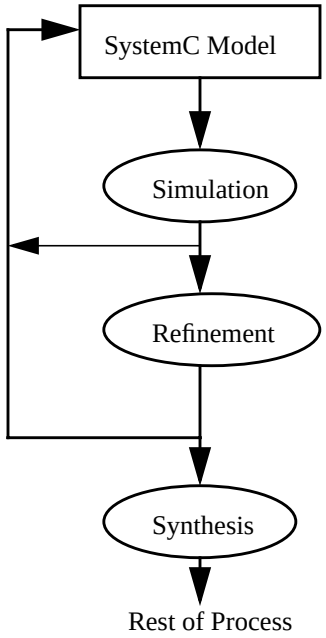
### Multiple System Tests

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.

For the parts of the original model to be implemented in software, the model has to be rewritten with calls to an RTOS. The model is the simulated and verified with an RTOS emulator. Though parts of the original code can be reused, the change in abstraction from the original model to an RTOS-based model requires significant manual recoding and verifying the changes becomes a significant problem.

## *SystemC Design Methodology*

The SystemC design approach offers many advantages over the traditional approach for system level design. The SystemC design methodology for hardware is shown in the figure below:




This technique has a number of advantages over the current design methodology, including the following:

### Refinement Methodology

With the SystemC approach, 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.

### Written in a Single Language

Using the SystemC approach, the designer does not have to be an expert in multiple languages. SystemC allows modeling from the system level to RTL, if necessary.

The SystemC approach provides higher productivity because the designer can model at a higher level. Writing at a higher level can result in 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. Using the same testbench also gives the designer a higher confidence that the system level and the RTL model implement the same functionality.

Though the current release of SystemC does not have the appropriate constructs to model RTOS, future version will. That will enable a similar refinement-based design methodology for the software parts of the system. Software designers will reap similar benefits as hardware designers.

## *Compatibility with Earlier Versions of SystemC*

SystemC v1.0 is fully backwards compatible with earlier versions of SystemC. However we strongly discourage the use of SystemC 0/9 syntax and the following classes from earlier versions:

- sc_bool_vector
- sc_logic_vector
- sc_array
- sc_2d
- sc_signal_bool_vector
- sc_signal_logic_vector
- sc_signal_logic
- sc_signal_resolved
- sc_signal_resolved_vector

CHAPTER 2

# Starting with a Simple Example

This section shows you a simple data protocol model example written first in C. The same model is then implemented in SystemC to show the highlights of using SystemC, along with instructions for compiling, executing, and debugging the design.

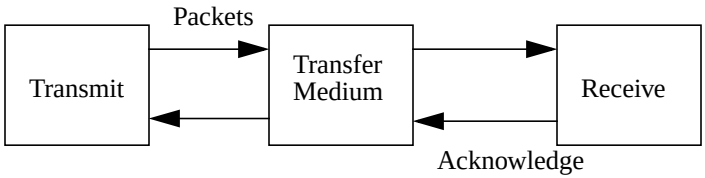
SystemC syntax and details about usage are described in subsequent chapters.

## Simplex Data Protocol

The simplex data protocol is a simple data protocol used to transfer data from one device to another in a single direction. (A duplex data protocol would transfer data in both directions.) The simplex data protocol can detect transfer errors, and it can resend data packets to successfully complete the data transfer if errors are detected.

The basic design consists of a transmitter, a receiver, and a model representing the data transfer medium (or channel). The data transfer medium can model wired and wireless networks. It can be a simple or complex model of data and error rates to match the actual physical medium.

A block diagram of the system is shown below:




The transmitter sends data packets to the data transfer medium. The data transfer medium receives those packets, and sends them on to the receiver. The data transfer medium can introduce errors to represent the actual error rate of the physical medium.

The receiver receives the data packets from the data transfer medium and analyzes the data packets for errors. If the data packet has no errors, the receiver generates an acknowledge packet and sends the acknowledgement packet back to the data transfer medium. The data transfer medium receives the acknowledge packet and sends this packet to the transmitter. The data transfer medium can introduce errors when sending the acknowledge packet that causes the acknowledge packet to not be properly received. After the transmitter has received the acknowledge packet for the previously sent data packet, the transmitter sends the next packet. This process continues until all data packets are sent.

This protocol works well for sending data in one direction across a noisy medium.

## C Model

An example model that implements this system in C++ is shown below:

```
frame data; //global data frame storage for Channel

void transmit(void) { //Transmits frames to Channel
  int framenum;      // sequence number for frames
  frame s;           // Local frame
  packet buffer;     // Buffer to hold intermediate data
  event_t event;     // Event to trigger actions
                     //in transmit

  framenum = 1;      // initialize sequence numbers

  get_data_fromApp(&buffer); // Get initial data
                             // from Application
  while (true) {
    s.info = buffer; // Put data into frame to be sent
    s.seq = framenum; // Set sequence number of frame
    send_data_toChannel(&s); // Pass frame to Channel
                             // to be sent
    start_timer(s.seq);     // Start timer to wait
                            // for acknowledge
    // If timer times out packet was lost
    wait_for_event(&event); // Wait for events from
                            // channel and timer
      if (event==new_frame) { // Got an event,
                              // check which kind
        get_data_fromChannel(s); // Read frame
                                 // from channel
        if (s.ack==framenum){ // Did we get the correct
                              // acknowledge
          get_data_fromApp(&buffer);
          // Yes, then get more data from
          // application, else send old packet again

          inc(framenum);      // Increase framenum
                              // for new frame
```

```
        }
      }
    }
}

void receiver(void) { // Gets frames from channel
  int framenum;       // Scratchpad frame number
  frame r,s;       // Temp frames to save information
  event_t event;   // Event to cause actions in receiver

  framenum = 1;   // Start framenum at 1
  while (true) {
    wait_for_event(&event);// Wait for data from channel
    if (event==new_frame){ // Event arrived see
                           //if it is a frame event
      get_data_fromChannel(r); // If so get the data
                               // from channel
      if (r.seq==framenum) {   // Is this the frame
                               // we expect
        send_data_toApp(&r.info); // Yes, then send
                                  //data to application
        inc(framenum); // Get ready for the next frame
      }
      s.ack = framenum -1;
      // Send back an acknowledge that frame
      // was received properly

      send_data_toChannel(&s);  // Send acknowledge
    }
  }
}

void send_data_toChannel(frame &f) { // Stores data
                                     // for channel
  data = f;    // Copy frame to storage area
}

void get_data_fromChannel(frame &f) { // Gets data from
                                      // channel
  int i;
```

```
  i = rand();  // Generate a random number
               // to cause receive errors

  if ( i > 10 && i < 500) {
    // If the random number is between 10 and 500
    // mess up the sequence number in the packet

    data.seq = 0;
      // This will cause the packet reception to
  } // fail - protocol should resend packet

  f = data;  // Copy data out of channel
}
```

The C++ model contains a transmit function, a receiver function, and two data transfer medium (or channel) functions. These channel functions get data from and put data to the channel (data transfer medium). *This description is not a complete implementation of the entire algorithm but only a fragment to show the typical style of a C++ model. Some of the model complexity is hidden in the wait_for_event() function calls. These calls are needed to take advantage of a scheduling mechanism built into the operating system, or you can implement a user defined scheduling system. In either case, this is a complex task.*

The transmit function, at the beginning of the C model, has local storage to keep frames and local data, and then it calls the function get_data_fromApp(). This function gets the first piece of data to send from the transmitter to the receiver.

The next statement is a while loop that continuously sends data packets to the receiver. In a real system, this while loop would have a termination condition based on how many packets were sent. However, in this example the designer wants to determine the data rate with varying noise on the channel, rather than sending real packets from one place to another.

The statements in the while loop fill in the data fields of the packet, the sequence number of the packet, and send the packet to the channel. The sequence number is used to uniquely identify the data packet so the correct acknowledge packets can be sent.

After the transmitter sends the packet to the channel, a timer is started. The timer allows the receiver to receive the frame and send back an acknowledge before the

the timer times out. If the transmitter does not receive an acknowledge after the timer has timed out, then the transmitter determines that the data frame was not successfully sent, and it will resend the packet.

When the transmitter sends a data packet and starts the timer, the transmitter waits for events to occur. These events can be timeout events from the timer, or they can be new_frame events from the channel. If the event received is a new_frame event, the transmit function gets the frame from the channel and examines the sequence number of the frame to determine if the acknowledge is for the frame just sent. If the sequence number is correct, the frame has been successfully received. Then, the transmitter gets the next piece of data to send and increments the sequence number of the frame. The transmitter sends the data frame and waits again for events.

If the timeout event was received, the test for a new_frame event fails and the transmitter resends the frame. This process continues until the frame is successfully sent.

The receiver function also has temporary storage to keep track of local data. At the first invocation, it initializes the frame sequence number to 1, similar to the transmitter function. This allows the two functions to get synchronized.

The receiver function has a main loop that waits only for new_frame events. After a new_frame is received, the receiver gets the frame from the channel and analyzes the contents.

If the sequence number of the frame matches the framenum variable, then the expected frame was sent and received properly. The receiver increments the framenum to get ready for the next frame.

The receiver generates an acknowledgement frame containing the sequence number minus 1. Because the frame sequence number is already incremented, the acknowledgement frame needs to subtract 1 from the framenum to acknowledge the last frame received. If the wrong frame was received, the acknowledgement contains an improper sequence number to inform the transmitter that the proper frame was not correctly transmitted.

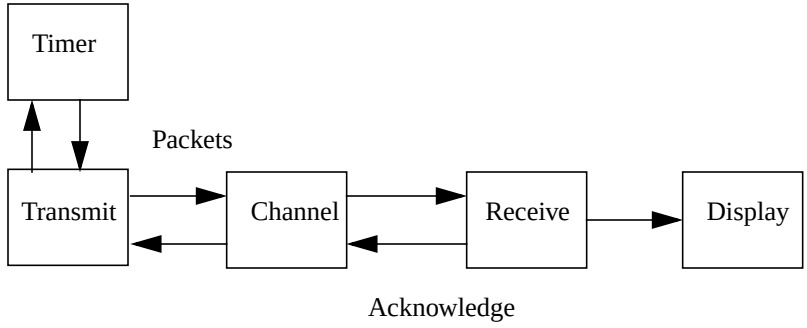
The last two functions in the C model send data to the channel and get data from the channel. These two functions are very simple in this model, but they could be complex, depending on the factors to be analyzed. Function send_data_to_channel() simply copies the received frame from the transmitter to a local variable. Function get_data_from_channel() reads the data from the local variable, but adds noise to

the data so some frames are not passed intact. Noise is generated by a random number generator that selectively zeroes the sequence number of the frame. The amount of noise is dependent on the total range of the sequence numbers and the range of numbers that cause the sequence number to be zeroed.

Using the C model, the designer can analyze the total data rate, effective data rate, error recovery, error recovery time, and numerous other factors. The designer can modify parameters such as frame rate size, error range size, data packet size, timer length to verify that the protocol works, and analyze the effects of these parameters.

## *SystemC Model*

Using SystemC the designer can design at a high level of abstraction using C++ high level techniques, and refine the design down to a level that allows hardware or software implementation. The block diagram for the SystemC implementation is shown below:




This block diagram is slightly different than the C model because the SystemC implementation is a more complete model. The SystemC description contains the transmit block, the receiver block, the channel block, a timer block, and a display block. The transmit, the receiver, and the channel blocks are the same as the C++ implementation. The display block emulates the application interface on the receiver side and the timer block generates timeout events. Packets are generated by a function in the transmit block and are sent through the channel to the receiver

block. The receiver block sends data to the display block where the data is displayed.

Let’s examine each block to see the descriptions and how they work.

## *User Defined Packet Type*

Before we describe the blocks, we need to look at the underlying packet data structure that passes data from module to module. The packet type is defined by a struct as shown below:

```
// packet.h file
#ifndef PACKETINC
#define PACKETINC

#include "systemc.h"

struct packet_type {
    long info;
    int  seq;
    int  retry;

  inline bool operator == (const packet_type& rhs) const
  {
    return (rhs.info == info && rhs.seq == seq &&
       rhs.retry == retry);
  }
};

extern
void sc_trace(sc_trace_file *tf, const packet_type& v,
const sc_string& NAME);


#endif

// packet.cc file
```

```
#include "packet.h"

  void sc_trace(sc_trace_file *tf, const packet_type& v,
                const sc_string& NAME) {
     sc_trace(tf,v.info, NAME + ".info");
     sc_trace(tf,v.seq, NAME + ".seq");
     sc_trace(tf,v.retry, NAME + ".retry");
  }
```

The struct has three fields, info, seq, and retry. Field info carries the data sent in the packet. The goal of this simulation is to measure the protocol behavior with respect to noise, not the data transfer characteristics. Therefore, the info field for data is of type long. Future versions of this data packet type could use a struct type for the data.

The second field is named seq and represents the sequence number assigned to this packet. For better error handling, this number will uniquely identify the packet during data transfers.

The third field in the packet is the retry field. This field contains the number of times the packet has been sent.

Other constructs in the packet.h and packet.cc files will be discussed later.

Let’s now take a look at the first block, the transmit block.

## *Transmit Module*

Notice that the transmit module includes the packet.h file which includes systemc.h. The systemc.h file gives the design access to all of the SystemC class methods and members. The packet.h file gives the design access to the packet definition and methods associated with the packet.

Note: In C++, function members are similar to C functions and data members are similar to C variables.

The SystemC description of the transmit module, described in the sections that follow, is shown below:

```
// transmit.h

#include "packet.h"

SC_MODULE(transmit) {
  sc_in<packet_type> tpackin;    // input port
  sc_in<bool> timeout;           // input port
  sc_out<packet_type> tpackout;  // output port
  sc_inout<bool> start_timer;      // output port
  sc_in<bool>  clock;

  int buffer;
  int framenum;
  packet_type packin, tpackold;
  packet_type s;
  int retry;
  bool start;

  void send_data();
  int get_data_fromApp();

  // Constructor
  SC_CTOR(transmit) {
    SC_METHOD(send_data);            // Method Process
    sensitive << timeout;
    sensitive_pos << clock;
    framenum = 1;
    retry = 0;
    start = false;
    buffer = get_data_fromApp();
  }
};

// transmit.cc

#include "transmit.h"
```

```
int transmit::get_data_fromApp() {
  int result;

  result = rand();
  cout <<"Generate:Sending Data Value = "<<result
       << "\n";
  return result;
}

void transmit::send_data() {
  if (timeout) {
    s.info = buffer;
    s.seq = framenum;
    s.retry = retry;
    retry++;
    tpackout = s;
    start_timer = true;
    cout <<"Transmit:Sending packet no. "<<s.seq
         << "\n";

   } else {
     packin = tpackin;
     if (!(packin == tpackold)) {
       if (packin.seq == framenum) {
         buffer = get_data_fromApp();
         framenum++;
         retry = 0;
        }
      tpackold = tpackin;
      s.info = buffer;
      s.seq = framenum;
      s.retry = retry;
      retry++;
      tpackout = s;
      start_timer = true;
      cout <<"Transmit:Sending packet no. "<<s.seq
           << "\n";
      }
   }
}
```

### *Module*

A module is the basic container object for SystemC. Modules include ports, constructors, data members, and function members. A module starts with the macro SC_MODULE and ends with a closing brace. A large design will typically be divided into a number of modules that represent logical areas of functionality of the design.

### *Ports*

Module transmit has three input, one output, and one inout ports as shown below:

```
sc_in<packet_type>  tpackin;      // input port
sc_in<bool>         timeout;      // input port
sc_out<packet_type> tpackout;     // output port
sc_inout<bool>      start_timer;  // inout port
sc_in<bool>         clock;        // input port
```

Port tpackin is used to receive acknowledgement packets from the channel. Port timeout is used to receive the timeout signal from the timer module and lets the transmit module know that the acknowledge packet was not received before the timer times out. The clock port is used to synchronize the different modules together so that events happen in the correct order.

Output port tpackout is the port that module transmit uses to send packets to the channel. Inout port start_timer is used by the transmit module to start the timer after a packet has been sent to the channel.

### *Data and Function Members*

After the port statements, local data members used within the module are declared.

The function members send_data() and get_data_fromApp() are declared in the transmit.h file and implemented in the transmit.cc file. This is the standard way to describe functionality in C++ and SystemC.

### *Constructor*

The module constructor identifies process send_data() as an SC_METHOD process, which is sensitive to clock and timeout. The constructor also initializes the variables used in the module. This is an important step. In HDL languages such as

VHDL and Verilog, all processes are executed once at the beginning of simulation to initialize variable and signal values. In SystemC the constructor of a module is called at initialization, and all initialization that needs to be performed is defined in the constructor.

The constructor initializes variable framenum to 1, which is used as the sequence number in all packets. Variable retry is initialized to 0, and variable start is initialized to false. Then the buffer, used to hold data is initialized by getting the first piece of data.

### *Implementation of Methods*

Let's now take a look at the transmit.cc file. This file implements the two methods declared in the header file. One method is a process, i.e. it executes concurrently with all other processes, the other is not.

Method get_data_fromApp() is a local method that generates a new piece of data to send across the channel. In the implementation you can see that this method calls the random number generator rand() to generate a new data value to send. Because we are validating the effects of noise on the protocol, the data values sent are not important at this stage of the design.

The method send_data() is a process because it is declared as such in the constructor for module transmit.The process checks first to see if timeout is true. Timeout will be true when the timer has completed.

Next the process checks to see if the current value of port tpackin is equivalent to the old value. This check is used to see if an acknowledgement packet was received from the channel.

If the values differ an acknowledgement packet has been received from the channel. Notice that tpackin was copied to local variable packin. Using a local variable allows you to access to the packet fields that cannot be accessed directly from the port.

The sequence number of the packet is checked against the last sent packet to see if they match. If they match, a correct acknowledgement was received and the next piece of data can be sent. The buffer will be filled with the next piece of data and the framenum is incremented. Field retry is also reset to 0, which is the initial value.

If no acknowledgement packet was received, the process was triggered by an event from the timeout port. This means the timer block completed its count or "timed out". If a timeout event occurs, that means the packet was sent, but no acknowledge was received. In this case, the packet must be transmitted again.

A local packet named s has all of its fields filled with the new data values to be sent, and it is assigned to tpackout. Notice that retry is incremented each time the packet is sent. This number is required to ensure the uniqueness of each packet.

Whenever a packet is written, the timer is started by the transmit process by setting the start_timer signal to true.

In summary, the transmit module sends a new packet to the channel. The timer is started to keep track of how long ago the packet was sent to the channel. If the acknowledge packet does not return before the timer times out, then the packet or acknowledge were lost and the packet needs to be transmitted again.

## *Channel Module*

The channel module accepts packets from the transmitter and passes them to the receiver. The channel also accepts acknowledge packets from the receiver to send back to the transmit module. The channel adds some noise to the transmission of packets to model the behavior of the transmission medium. This causes the packets to fail to be properly received at the receiver module, and acknowledge packets fail to get back to the transmit module. The amount of noise added is dependent on the type of transfer medium being modeled.

The SystemC channel description is shown below:

```
// channel.h

#include "packet.h"

SC_MODULE(channel) {
  sc_in<packet_type> tpackin;      // input port
  sc_in<packet_type> rpackin;      // input port
  sc_out<packet_type> tpackout;    // output port
  sc_out<packet_type> rpackout;    // output port
```

```
  packet_type packin;
  packet_type packout;

  packet_type ackin;
  packet_type ackout;

  void receive_data();
  void send_ack();

  // Constructor
  SC_CTOR(channel) {
    SC_METHOD(receive_data);   // Method Process
    sensitive << tpackin;

    SC_METHOD(send_ack);       // Method Process
    sensitive << rpackin;
  }
};

// channel.cc

#include "channel.h"

  void channel::receive_data() {
    int i;
    packin = tpackin;
    cout << "Channel:Received packet seq no.
             = " << packin.seq << "\n";
    i = rand();
    packout = packin;
    cout <<"Channel: Random number = "<<i<<endl;

    if ((i > 1000) && (i < 5000)) {
      packout.seq = 0;
    }
    rpackout = packout;
  }

  void channel::send_ack(){
     int i;
```

```
    ackin = rpackin;
    cout <<"Channel:Received Ack for packet
           = " << ackin.seq << "\n";
    i = rand();
    ackout = ackin;

    if ((i > 10) && (i < 500)) {
      ackout.seq = 0;
    }
    tpackout = ackout;
}
```

### *Ports*

The channel description contains four ports, two input ports and two output ports. Port tpackin accepts packets from the transmit module and port rpackin accepts acknowledge packets from the receiver. Port tpackout sends acknowledge packets to the transmit module and port rpackout sends data packets to the receiver.

### *Data and Function Members*

Notice the four local packet variable declarations to hold the values of the packet ports. Local variables are necessary so you can access the packet internal data fields.

The channel description has two processes. Process receive_data() is used to get data from the transmit module and pass the data to the receiver module. Process send_ack() gets acknowledge packets from the receiver module and sends them to the transmit module.

Process receive_data is sensitive to events on port tpackin. Thus when a new packet is sent from the transmit module, process receive_data() is invoked and analyzes the packet. Process send_ack() is sensitive to events on port rpackin. When the receiver module sends an acknowledge packet to the channel module, the value of port rpackin is updated and causes process send_ack to be invoked.

### *Constructor*

In the channel constructor, we can see that both processes are SC_METHOD processes.

### *Implementation of Methods*

Let’s take a closer look at process receive_data. The first step is to copy tpackin to a local variable so that the packet fields can be accessed. A message is printed for debugging purposes. A random number is generated to add noise to the channel. The packet is assigned to the output packet and another debugging message is printed displaying the random number value. Finally, an if statement determines whether the packet passes through as received, or if the packet is altered by adding noise.

If the random number is within the specified range, the sequence number of the packet is set to 0. This means the packet was corrupted. The last two statements of the process receive_data copies the altered or unaltered packet to output port rpackout

You can modify the range of random numbers generated and the range of numbers that modify the packet sequence number to control the amount of noise injected.

The send_ack process, triggered by events on port rpackin, is very similar to the receive_data process. It assigns rpackin to the local packet ackin so the fields of the packet can be examined. Next, a debug message is written, and a random number representing the noise in the channel for acknowledgements is generated. This process also uses a specified range to modify the sequence number field of the packet. Finally, the packet is assigned to tpackout where it will be passed to the transmit module.

## *Receiver Module*

The receiver module accepts packets from the channel module and passes the data received to the virtual application. In this design the virtual application is a display, modeled in the display module. When the receiver module successfully receives a packet, it send an acknowledgement packet back to the transmit module. Incoming packet sequence numbers are compared with an internal counter to ensure the correct packets are being transmitted.

The receiver module is shown below:

```
// receiver.h

#include "packet.h"

SC_MODULE(receiver) {
  sc_in<packet_type> rpackin;      // input port
  sc_out<packet_type> rpackout;    // output port
  sc_out<long> dout;               // output port
  sc_in<bool>  rclk;

  int framenum;
  packet_type packin, packold;
  packet_type s;
  int retry;

  void receive_data();

  // Constructor
  SC_CTOR(receiver) {
    SC_METHOD(receive_data);       // Method Process
    sensitive_pos << rclk;
    framenum = 1;
    retry = 1;
  }
};

// receiver.cc

#include "receiver.h"


  void receiver::receive_data(){
    packin = rpackin;
    if (packin == packold) return;
    cout <<"Receiver: got packet no. = "<<packin.seq
         << "\n";
    if (packin.seq == framenum) {
      dout = packin.info;
      framenum++;
      retry++;
```

```
      s.retry = retry;
      s.seq = framenum -1;
      rpackout = s;
    }
    packold = packin;
  }
```

### *Ports*

The receiver module has four ports, two input port, and two output ports. Input port rpackin accepts packets from the channel. Input rclk is the receiver block clock signal. Output port rpackout is used to send acknowledge packets to the channel where they may be passed to the transmit module. Output port dout transfers the data value contained in the packet to the display module for printing.

### *Constructor*

The receiver module contains one SC_METHOD process named receive_data, which is sensitive to positive edge transitions on input port rclk. Notice in the receiver module constructor that variable framenum is initialized to 1. Module transmit also initializes a framenum variable to 1, so both transmit and receiver are synchronized at the start of packet transfer.

### *Implementation of Methods*

As we have already seen, process receive_data is invoked when a new packet arrives on the rpackin port. The first step in the process is to copy rpackin to the local variable packin for packet field access. The receiver block, like the transmit block, compares the new packet value with the old packet value to determine if a new packet has been received. A debug message is printed and the process checks to see if the sequence number of the incoming packet matches the expected framenum. If so, the packet data is placed on the dout port where it will be sent to the display module. Next, the framenum variable is incremented to reflect the next framenum expected.

If a packet is successfully received an acknowledge packet needs to be sent back to the transmit module. A local packet named s has its sequence number filled in with framenum. After the sequence number field is updated, packet s is assigned to port rpackout.

## *Display Module*

The display module is used to format and display the packet data received by the receiver module. In this example the data is a very simple type long value. This makes the display module very simple. However, as the simplex protocol is more completely implemented, the data being sent could grow in complexity to the point that more complex display formatting is needed. The display module is shown below:

```
// display.h

#include "systemc.h"
#include "packet.h"

SC_MODULE(display) {
  sc_in<long> din;     // input port

  void print_data();

  // Constructor
  SC_CTOR(display) {
    SC_METHOD(print_data);  // Method process to print
data
    sensitive << din;
  }
};
```

// display.cc

#include "display.h"

void display::print_data() {

```
  cout <<"Display:Data Value Received, Data = "<<din << "\n";

 }
```

This module only has one input port named din. It accepts data values from the receiver module. When a new value is received, process print_data is invoked and writes the new data item to the output stream.

## *Timer Module*

The timer module implements a timer for packet retransmission. The delay allows a packet to propagate to the receiver and the acknowledge to propagate back before a retransmit occurs. Setting the delay properly is a key factor in determining the maximum data rate in a noisy environment. Without the timer, the transmitter would not know when to retransmit a packet that was lost in transmission. The delay value is a parameter that can easily be modified to find the optimum value.

The timer module is shown below:

```
// timer.h

#include "systemc.h"

SC_MODULE(timer) {
  sc_inout<bool> start;       // input port
  sc_out<bool> timeout;    // output port
  sc_in<bool> clock;       // input port

  int count;


  void runtimer();

  // Constructor
  SC_CTOR(timer) {
    SC_THREAD(runtimer);    // Thread process
    sensitive_pos << clock;
```

```
    sensitive << start;
    count = 0;
  }
};

// timer.cc

#include "timer.h"

  void timer::runtimer() {
    while (true) {
      if (start) {
       cout <<"Timer: timer start detected"<<endl;
        count = 5;   // need to make this a constant
        timeout = false;
        start = false;
      } else {
        if (count > 0) {
          count--;
          timeout = false;
        } else {
          timeout = true;
        }
      }
      wait();
    }
  }
```

### *Ports*

The timer module has one input port, one inout port, and one output port. Port start is an inout port of type bool. The transmit module activates the timer by setting start to true. The timer module starts the count and resets the start signal to false. Port clock is an input port of type bool that is used to provide a time reference signal to the timer module. Output port timeout connects to the transmit module and alerts the transmit module when the timer expires. This means that either the packet or acknowledge were lost during transmission and the packet needs to be transmitted again.

### *Constructor*

The timer module contains one process called runtimer. This process is sensitive to the positive edge of port clock.

### *Implementation of Methods*

If the value of the start signal is 1 the timer is started. When the timer is started, a debug message is written out, the value of variable count is set to the timer delay value, and output port timeout is set to false. The false value signifies that the timer is running and has not timed out yet. The timer process then resets the start signal to 0.

If variable count is greater than 0, the timer is still counting down. Variable count is decremented and port timeout stays false. If variable count is equal to 0, the timer has expired and timeout is set to true. At this point the transmit module knows that the packet was lost and retransmits the packet.

## *Putting it all together - The main routine*

The sc_main routine is the top-level routine that ties all the modules together and provides the clock generation and tracing capabilities. The sc_main routine is shown below:

```
// main.cc

#include "packet.h"
#include "timer.h"
#include "transmit.h"
#include "channel.h"
#include "receiver.h"
#include "display.h"


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

```
  sc_signal<packet_type> PACKET1, PACKET2, PACKET3,
PACKET4;
  sc_signal<long> DOUT;
  sc_signal<bool> TIMEOUT, START;

  sc_clock CLOCK("clock", 20);  // transmit clock
  sc_clock RCLK("rclk", 15);    // receive clock

  transmit t1("transmit");
  t1.tpackin(PACKET2);
  t1.timeout(TIMEOUT);
  t1.tpackout(PACKET1);
  t1.start_timer(START);
  t1.clock(CLOCK);

  channel c1("channel");
  c1.tpackin(PACKET1);
  c1.rpackin(PACKET3);
  c1.tpackout(PACKET2);
  c1.rpackout(PACKET4);

  receiver r1("receiver");
  r1.rpackin(PACKET4);
  r1.rpackout(PACKET3);
  r1.dout(DOUT);
  r1.rclk(RCLK);

  display d1("display");
  d1 <<DOUT;

  timer tm1("timer");
  tm1 <<START<<TIMEOUT<<CLOCK.signal();

  // tracing:
  // trace file creation
  sc_trace_file *tf = sc_create_vcd_trace_file
     ("simplex");
  // External Signals
  sc_trace(tf, CLOCK.signal(), "clock");
  sc_trace(tf, TIMEOUT, "timeout");
  sc_trace(tf, START, "start");
```

```
  sc_trace(tf, PACKET1, "packet1");
  sc_trace(tf, PACKET2, "packet2");
  sc_trace(tf, PACKET3, "packet3");
  sc_trace(tf, PACKET4, "packet4");
  sc_trace(tf, DOUT, "dout");

  sc_start(10000);


  return(0);
}
```

### *Include Files*

Notice that the sc_main file includes all of the other modules in the design. You instantiate each of the lower level modules and connect their ports with signals to create the design in sc_main. To instantiate a lower level module, the interface of the module must be visible. Including the .h file from the instantiated module provides the necessary visibility.

### *Argument to sc_main*

The sc_main routine takes the following arguments:

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

The argc argument is a count of the number of command line arguments and the argv is an array containing the arguments as char* strings. This is the standard C++ way of parsing command line arguments to programs.

### *Signals*

After the sc_main statement, the local signals are declared to connect the module ports together. Four signals are needed for packet_type to cross connect the transmit, receiver, and channel modules. There are two clock declarations, clock and rclk. Clock is used as the transmitter clock and will synchronize the transmit block and the timer block. Rclk is used as the receiver clock and will synchronize the receiver block and the display block.

### *Module Instantiation*

After the declaration statements, the modules in the design are instantiated. The transmit, channel, receiver, display, and timer are instantiated and connected together with the locally declared signals. This completes the implementation of the design.

### *Positional and Named Connections*

In the sc_main file two different types of connections were used to connect signals to the module instantiations. Modules transmit, channel, and receiver used named connections. A named connection connects a port name to a signal name. Notice that the port names were in lowercase and the signal names in uppercase.

Modules display and timer used positional connections to connect signals to the module instantiations. With this style of connection a list of signals is passed to the instantiation and the first signal in the list connects to the first port, the second signal to the second port, etc.

### *Using Trace*

The program can now be built and run. To make is easier to determine if the design works as intended, you can create a trace file with the built-in signal tracing methods in SystemC. The first trace command, shown below, creates a trace file named simplex.vcd into which the results of simulation can be written:

```
sc_trace_file *tf = sc_create_vcd_trace_file
 ("simplex");
```

Next, a set of sc_trace commands trace the signals and variables of a module, as follows:

```
sc_trace(tf, CLOCK.signal(), "clock");
sc_trace(tf, TIMEOUT, "timeout");
```

These commands write the value of the signal specified to the trace file previously created. The last argument specifies the name of the signal in the trace file.

After simulation is executed, you can examine the results stored in the trace file with a number of visualization tools that generate waveforms and tables of results.

### *Simulation Start*

After the trace commands, the following function call instructs the simulation kernel to run for 10,000 time units and stop:

```
sc_start(10000);
```

Alternatively, you can use an sc_start value of -1, as shown below:

```
sc_start(-1);
```

This command tells the simulation to run forever.

After the example is completely described in SystemC, the commands to build the simulator need to be specified. The following sections provide procedures for compiling under UNIX and Windows.

## *Compiling the Example for UNIX*

The following steps are needed to compile the design for the UNIX environment:

1. Create a new directory for the design and create all the design files in it.
2. Copy file Makefile and Makefile.defs from the SystemC installation directory into the new directory.
3. Edit the Makefile so that the list of files includes all of the design source files. An example Makefile is shown below:

```
TARGET_ARCH = gccsparcOS5

MODULE = demo
SRCS = channel.cc display.cc packet.cc receiver.cc
timer.cc transmit.cc main.cc
OBJS = $(SRCS:.cc=.o)

include ./Makefile.defs
```

   Edit the SRCS line to list all of the source files in the design.

   Don’t remove the line "include ./Makefile.defs".

The MODULE line specifies the name of the executable to run when the compilation is done. In this example, the compilation creates a program named demo.

4. Open the Makefile.defs and make sure that the SYSTEMC line points to the current location of the SystemC class libraries.

   An example is shown below:

```
TARGET_ARCH = gccsparcOS5
CC     = g++
OPT    =
DEBUG  = -g
SYSTEMC = /remote/dtg403/dperry/systemc-1.0
INCDIR = -I. -I.. -I$(SYSTEMC)/include
LIBDIR = -L. -L.. -L$(SYSTEMC)/lib-$(TARGET_ARCH)
CFLAGS = -Wall $(DEBUG) $(OPT) $(INCDIR) $(LIBDIR)
        -fexceptions
LIBS   =  -lsystemc -lnumeric_bit -lstdc++ -lqt -lm
          $(EXTRA_LIBS)

// rest of file not shown
```

5. By default the simulation is built with debugging turned on. Modify the DEBUG line to turn on or off the debugging options as desired.
6. To compile the design, enter the following in the command line:

```
   unix% gmake
or
   unix% make
```

## *Compiling the Example for Windows*

The SystemC distribution includes project and workspace files for Visual C++. If you use these project and workspace files the SystemC source files are available to your new project. For Visual C++ 6.0 the project and workspace files are located in directory:

```
<systemc installation directory>\lib_VC6
```

The project file is lib_VC6.dsp and the workspace file is lib_VC6.dsw. Double-click on the .dsw file to launch Visual C++ with the workspace file. The workspace

file will have the proper switches set to compile for Visual C++ 6.0. For models that do not use the fixed point types add lib_VC6\release\systemc.lib and numeric_bit.lib. If you are going to use fixed point types you also need to add fx.lib. Do not add this library if fixed point types are not needed in your design.

Make sure that the Run Time Type Information switch is on by using the Project->Project Settings menu item. Select the C++ Language tab and make sure that the Enable Run Time Type Information checkbox is checked.

To create a new design, first create a new project by using the "Add to Project->New" selection under the Project menu. Select the Project tab on the dialog box that appears and specify a Win32 Console application. Create an empty project.

Next add the source files to the project by using the Project->Add to Project->Files menu item. Make sure that the files are added to the new project directory just created.

Now use the Compile and Build menu selections to compile and build the SystemC application. When the application has been built, the design can be run from Visual C++ to debug the application.

## *Executing the Example*

After the simulation executable is built, you run the simulation by executing the simulation executable created in the compilation step. The simulation executable is a batch program that executes the simulation. For example, to run a simulation for a module named demo, simply type demo at the command prompt and press return. If you built a console application in Visual C++ you can run the application in a Windows Command Prompt window by typing the name of the project created.

The duration of the simulation is specified by method sc_start in the sc_main module. The data created by the simulation is specified with the sc_trace commands in the sc_main module.

When the simulation is complete, a trace file of the traced signals is created. You can use tools to view waveforms and tables from this data and analyze the results of simulation and determine whether or not the simulation succeeded.

CHAPTER 3

# *Modules and Hierarchy*

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

Modules are declared with the SystemC keyword SC_MODULE as shown by the example below:

```
SC_MODULE(transmit) {
```

The identifier after the SC_MODULE keyword is the name of the module, which is transmit in this example. This syntax uses a macro named SC_MODULE to declare a new module named transmit. Another way to declare a module is the following:

```
struct transmit : sc_module {
```

This form of declaration resembles a typical C++ declaration of a struct or a class. The macro SC_MODULE provides an easy and very readable way to describe the module.

A module can contain a number of other elements such as ports, local signals, local data, other modules, processes, and constructors. These elements implement the required functionality of the module.

## *Module Ports*

Module Ports pass data to and from the processes of a module. You declare a port mode as in, out, or inout. You also declare the data type of the port as any C++ data type, SystemC data type, or user defined type.


The figure above shows a fifo module with a number of ports. The ports on the left are input ports or inout ports while the ports on the right are output ports. Each port has an identifying name. Graphic symbols like the one shown above typically do not contain port types, so it is not clear from the symbol which port types are present. The SystemC description of these ports is shown below:

```
SC_MODULE(fifo) {
  sc_in<bool>   load;
  sc_in<bool>   read;
```

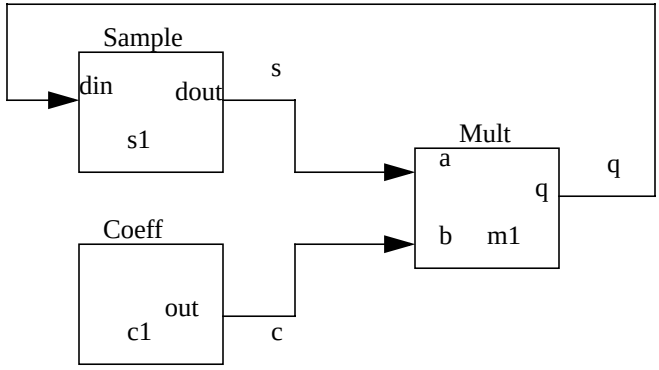
```
  sc_inout<int> data;
  sc_out<bool>  full;
  sc_out<bool>  empty;

  //rest of module not shown
}
```

Each port on the block diagram has a matching port statement in the SystemC description. Port modes sc_in, sc_out, and sc_inout are predefined by the SystemC class library.

## *Module Signals*

Signals can be local to a module, and are used to connect ports of lower level modules together. These signals represent the physical wires that interconnect devices on the physical implementation of the design. Signals carry data, while ports determine the direction of data from one module to another. Signals aren't declared with a mode such as in, out, or inout. The direction of the data transfer is dependent on the port modes of the connecting components.

**FIGURE 1. Filter Design**



The example in Figure 1 shows the data path of a simple filter design. There are three lower level modules instantiated in the filter design, sample, coeff, and mult modules. The module ports are connected by three local signals q, s, and c.

Note: Instantiation means that an instance of an object is created. It is the same as declaring a new object in C++.

### Positional Connection

There are two ways to connect signals to ports in SystemC.

- named mapping
- positional mapping

First let's examine the example in Figure 1 using positional mapping.

The SystemC description for this example looks as follows:

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

The four include files at the beginning of the module give the designer access to the SystemC classes and the declarations of the instantiated modules. The top level of the design and the module are both named filter. The top level module does not have any ports, which is legal for the top of the design.

Below the include statements are pointer declarations that allow allocation of the objects to be instantiated in the design. You declare a pointer variable for each object that will be instantiated later.

Next, the local signal are declared using the SystemC template class sc_signal. The type of the signal being passed is entered between the angle brackets (<>). In this example the type of the signal is a SystemC data type sc_uint. Notice that there is an extra space inserted between the "32>" and the ">" in the declaration. This is required to allow the description to compile. The three modules in this design are instantiated in the constructor SC_CTOR.

Each instantiation contains two line of SystemC description. The first line creates a new object and a pointer to the object. The second line uses the object pointer to map signals to the object ports. This style of mapping is called positional mapping. Each signal in the mapping matches the port of the instantiated module on a positional basis. The first signal listed in the mapping connects to the first port in the instantiation, the second signal connects to the second port, etc. The order and number of ports in this style of mapping is very important. If the order is not followed properly signals of one type can get connected to ports of another type. This will produce a runtime error.

Positional connections can work very well for small instantiations with few ports to make the description small. However, for instantiations with a large number of ports, connecting with positional connection can be confusing. For these cases, it is better to use named connection.

### Named Connection

The same design with named mapping is shown below:

```
#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);
  }
}
```

This example uses named connection for the component instantiations. The first named connection connects port din of module s1(sample) to signal q of module filter. The second named connection connects port dout of module s1 to signal s of module filter. Using named connection the designer can create the signal to port connections in any order.

## *Internal Data Storage*

For storage of data within a module, the designer can declare local variables. Internal data storage can be of any legal C++ type, SystemC type, or user defined type. Local storage is not visible outside the module unless the designer specifically makes the data visible.

```
// 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 "systemc.h"
#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;
  }
```

The example above implements an integer counter. On a rising edge of port clock, the process count_up executes. If the load input is true, port din is loaded into the counter. Otherwise, the counter increments its value by 1. The count_val variable is used to store the intermediate value of the counter. It is local storage, not visible outside the counter module.

## Processes

So far the interface and storage of modules have been discussed, but not the part of the module that provides the functionality. The real work of the modules are performed in processes. Processes are functions that are identified to the SystemC kernel and called whenever signals these processes are “sensitive to” change value. A process contains a number of statements that implement the functionality of the process. These statements are executed sequentially until the end of the process occurs, or the process is suspended by one of the wait function calls.

Processes look very much like normal C++ methods and functions with slight exceptions. Processes are methods that are registered with the SystemC kernel. There are a number of different types of processes including method processes, thread processes, and clocked thread processes. Process types are discussed in Chapter 4, “Processes,”. The process type determines how the process is called and executed. Processes can contain calls to a function named wait() that will halt execution of the process at different points. Signal value changes cause the process to receive events and execute statements in a process. An example process is shown below:

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

```
void dff::doit() {
  dout = din;
}
```

This module describes a flip flop device. The module has a clock input (clock), a data input (din), and a data output (dout). When a rising edge (0 to 1 value) occurs on the clock input object, input port data is assigned to output port dout. The value change on input clock triggers method doit to execute. Let's take a closer look at how this occurs in SystemC.

Process doit() is described as a method in the module. This method will be called whenever a positive edge occurs on port clock.

This behavior is described by the following statements in the constructor for module dff:

```
SC_METHOD(doit);
sensitive_pos << clock;
```

The first statement specifies that module dff contains a process named doit. It also specifies that this process is an SC_METHOD process. An SC_METHOD process is triggered by events and executes all of the statements in the method before returning control to the SystemC kernel (more about processes later). The second statement specifies that the process is 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 and then returns control to the SystemC kernel. Another event causes the process to be invoked again, and the assignment statement is executed again.

## *Module Constructors*

The final item that makes up a module is the constructor. The module constructor creates and initializes an instance of a module. The constructor creates the internal data structures that are used for the module and initializes these data structures to known values. The module constructors in SystemC are implemented such that the

instance name of the module is passed to the constructor at instantiation (creation) time. This helps identify the module when errors occur or when reporting information from the module. Example constructors have already been looked at briefly, but let's take a more detailed look at slightly more complex constructors. Below is an example RAM:

```
// 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
```

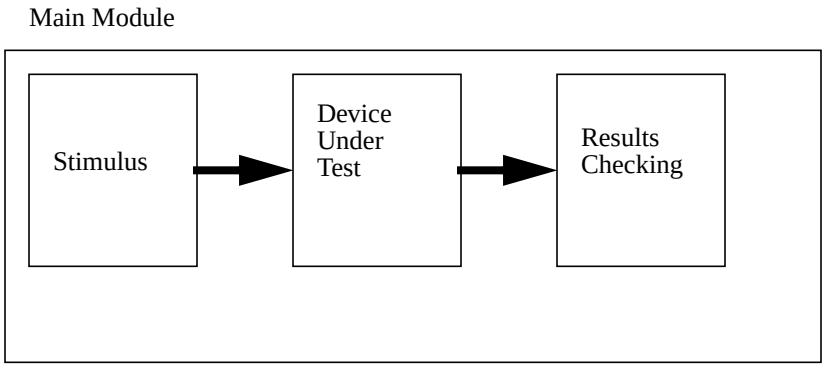
This example implements a RAM memory device. The RAM can be written to and read from the two processes, read() and write(). The constructor contains declarations for each of the processes. Both are described as SC_METHOD type processes[1]. The for loop is used to initialize the memory to 0 values.

When a RAM module is instantiated the constructor will be called, data allocated for the module, and the two processes registered with the SystemC kernel. Finally the for loop will be executed which will initialize all the memory locations of the newly created ram module.

## *TestBenches*

Testbenches are used to provide stimulus to a design under test and check design results. The testbench can be implemented in a number of ways. The stimulus can be generated by one process and results checked by another. The stimulus can be embedded in the main program and results checked in another process. The checking can be embedded in the main program, etc. There is no clear "right" way to do a testbench, it is dependent on the user application.

A typical testbench might look as follows:




The stimulus module will provide stimulus to the Device Under Test (DUT) and the Results Checking module will look at the device output and verify the results are correct.

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

1. Described in Chapter 4, “Processes,”

results checking module. Some designers combine the stimulus and results checking modules into one module. Also the results checking module can be left out if the designer does manual analysis of the output results. For some designs this technique works well because the output results are easy to check. For example if the device under test is a graphics manipulation device and the stimulus is a picture to be manipulated, the designer just needs to look at the output picture to verify that the results are as expected.

An example testbench for the counter example described on page 44 is shown below:

```
// 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 "systemc.h"
#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
  }
}
```

The testbench will drive the load and din inputs of the count module. The clock input of the count module and the clock input of the count_stim module will be generated from a clock object located in the sc_main routine discussed in the next section.

The first two statements in the while loop of the process will load the value 0 into the count module. The count module is loaded when the load input is true. The value loaded into the count module is the value of din. When the load signal goes to false and a positive edge occurs on input clock, the counter will count up. After the first wait() call, the load input will be set to false allowing the counter to count up. Successive clocks will allow the counter to keep counting up until the end of the while loop is reached. At this point, execution will start at the beginning of the while loop and the counter will be loaded with 0.

Since the count module is a simple design, the stimulus for it is trivial. More complex designs will have more complex stimulus. This style of test bench will support more complex stimulus. As mentioned earlier stimulus can also be read from a file. This has the added benefit of changing the stimulus without recompiling the design.

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 the following:

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

CHAPTER 4

# Processes

Processes are the basic unit of execution within SystemC. Processes are called to emulate the behavior of the target device or system. Three types of SystemC processes are available:

- Methods
- Threads
- Clocked Threads

Each of these processes has unique behavior and are discussed in the next few sections.

In a typical programming language, methods are executed sequentially as control is transferred from one method to another to perform the desired function. Typical programming languages can be used to model sequential behavior of systems very easily. However electronic systems are inherently parallel with lots of parallel activity constantly taking place. Modeling these parallel activities with a sequential language can be difficult. Typical solutions to these problems brought about the creation of special Hardware Description Languages such Verilog and VHDL for

modeling the hardware part of the system, and linking in C or C++ descriptions for the software part of the design. SystemC has the concept of Methods, Threads, and Clocked Threads to model the parallel activities of a system.

## *Basics*

Some processes behave just like functions, the process is started when called, and returns execution back to the calling mechanism when complete. Other processes are called only once at the beginning of simulation and are either actively executing or suspended waiting for a condition to be true. The condition can be a clock edge or a signal expression or combination of both.

Processes are not hierarchical, so no process will call another process directly. Processes can call methods and functions that are not processes.

Processes have sensitivity lists, i.e. a list of signals that cause the process to be invoked, whenever the value of a signal in this list changes. Processes cause other processes to execute by assigning new values to signals in the sensitivity list of the other process.

To trigger a process a signal in the sensitivity list of the process must have an event occur. The event on the signal is the triggering mechanism to activate the 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. Any processes sensitive to that signal will recognize that there was an event on that signal and invoke the process.

## *Method Process*

When events (value changes) occur on signals that a process is sensitive to, the process executes. A method executes and returns control back to the simulation kernel. A simple method is shown below:

```
// 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 "systemc.h"
#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;
  }
}
```

This example shows a module called rcv that has an input named xin and an output named id. The module contains a single method named extract_id. The method is sensitive to any changes on input xin. When input xin changes, method extract_id is invoked. Method extract_id will execute and assign a value to port id. When the method terminates, control is returned back to the SystemC scheduler.

Method processes cannot be suspended or contain infinite loops. When a method process is invoked, it executes until it returns.

## *Thread Processes*

Thread Process can be suspended and reactivated. The Thread Process can contain wait() functions that suspend 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().

The input signals that cause the process to reactivate are specified by the sensitivity list. The sensitivity list is specified in the module constructor with the same syntax used in the Method Process example.

A sample Thread Process is shown below:

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

void traff::control_lights() {
  NSred = false;
  NSyellow = false;
  NSgreen = true;
  EWred = true;
  EWyellow = false;
  EWgreen = false;

  while (true) {
    while (roadsensor.delayed() == 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
  }
}
```

This module is a simple traffic light controller. There is a main highway running North-South that normally has a green light. A highway sensor exists on the East-West road that crosses the highway. A car on the East-West side road will trigger the sensor causing the highway light to go from green to yellow to red, and the side road to change from red to green. The model uses two different time delays. The green to yellow delay is longer than the yellow to red delay to represent the way that a real traffic light works.

The starting state of the model will wait for an event on the road sensor. When this occurs the NS (North-South) lights will change to yellow, and the model will wait for the yellow to red delay. After the delay the NS lights are changed to red and the EW (East-West) lights are changed to green. The model will now wait for the green to yellow delay to allow the cars to have time to cross the highway. After this delay is complete the EW lights are changed to yellow and finally to red. The module waits one more long delay after the highway light goes back to green so that another car will not trip the sensor immediately.

The module has one SC_THREAD process named control_lights. As can be seen from the constructor it is sensitive to the roadsensor, shorttimer, and longtimer input ports. In the steady-state condition the process will be waiting for events on the roadsensor input.

The 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. Each change of state in the traffic light controller would have to be declared as a state in a state machine.

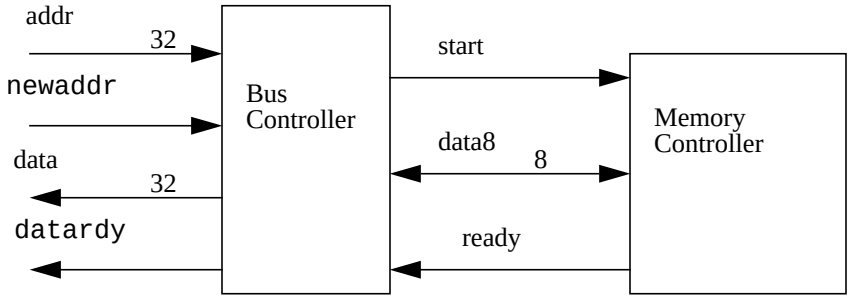
Thread processes are implemented as co-routines and with the SystemC class library. This implementation is slower than SC_METHOD processes. If simulation speed is a current goal of the simulation, limit the SC_THREAD processes as needed to maintain the highest simulation speed.

## *Clocked Thread Process*

Clocked Thread Processes are a special case of a Thread Process. Clocked Thread Processes help designers describe their design for better synthesis results. Clocked Thread Processes are only triggered on one edge of one clock, which matches the way hardware is typically implemented with synthesis tools. Clocked threads can be used to create implicit state machines within design descriptions. 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. 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 but 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.




A 32-bit address will be passed to the bus controller process. This 32-bit address will be multiplexed byte by byte through the 8-bit data bus to form the 32-bit address in the memory controller. After the address has been sent the bus controller will wait until the ready signal from the memory controller is active and start receiving the 32 bits of data from the memory controller. After all of the data is received the bus controller will send the data back to the microcontroller on the 32-bit data bus. Each of these transfers takes 4 cycles of the clock to transfer the 32-bit data 8 bits at a time.

The bus controller will initially wait for the newaddr signal to become active. When newaddr becomes active a new address is present on the addr inputs. The start signal is sent to the memory controller with the first byte of the address on the data8 bus. Successive bytes are passed on bus8 until all of the bytes have been sent. The bus controller will now wait for the ready signal from the memory controller. This signal tells the bus controller that the data from the memory controller is ready. Now the bus controller will transfer a byte at a time from bus8 to a temporary location in the bus controller. Once the entire data value is received the data value is transferred to output data and the datardy signal activated.

The SystemC description of the bus controller is shown below:

```
// 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());
    datardy = true;  // ready to accept new address
  }
};

// bus.cc

#include "systemc.h"
#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();
    datardy = false;   // cannot accept new address now
    data8 = taddr.range(7,0);
    start = true;      // new addr for memory controller
    wait();

    // wait 1 clock between data transfers
    data8 = taddr.range(15,8);
    start = false;
    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;
    datardy = true;   // data is ready, new addresses ok

  }
}
```

Notice that the constructor for module bus contains one SC_CTHREAD process. The SC_CTHREAD process is different from the SC_THREAD process in a number of ways. First the SC_CTHREAD process specifies a clock object. When other process types are described in a module constructor they only have the name of the process specified, but the SC_CTHREAD process has the name of the process and the clock that triggers the process. An SC_CTHREAD does not have a separate sensitivity list like the other process types. The sensitivity list is just the specified clock edge. The SC_CTHREAD process will be activated whenever the specified clock edge occurs. In this example the positive edge of the clock is specified so process xfer will execute on every positive edge of the clock.

When the process first starts execution will stop at the first wait_until() method. A wait_until() function will suspend execution until the expression passed as an argu-

ment is true. Once the newaddr signal has become true the process will assume that a new address value exists on port addr. One point to keep in mind is that signals assigned new values by an SC_CTHREAD process will be not be available until after the next clock edge occurs.

The addr value will now be placed on output signal data8 one byte at a time. When the first value of addr is output the start signal is activated to let the memory controller know that a new address is coming. Once all of the address values have been sent the process will now wait for the ready signal to come back from the memory controller signaling that the memory data is ready to be read. The SC_CTHREAD process will continue to be activated every clock edge, but execution will not continue until the wait_until() condition becomes true. (See the wait_until() description in the next section)

Once the wait_until() condition becomes true the process will continue by reading the data values from port data8 into temporary data structure tdata. Once all of the data values have been read tdata is transferred to output data and the datardy signal is set to true signaling the microcontroller that the data is ready to be read.

An SC_CTHREAD process can only be triggered by one sc_clock object. It is also necessary that the specified clock is an sc_clock object. In the example above an sc_clock object is passed to the bus module through port clock. Port clock is an sc_in_clk port. This port is used to pass clock objects and is necessary for this example to compile. The pos() or neg() method of this object is passed to the SC_CTHREAD constructor to specify the clock edge which triggers the method.

## *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.

An example wait_until() method is shown below:

```
wait_until(roadsensor.delayed() == true);
```

This statement will 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. A compilation error will result if the delayed() method is not present.

The wait_until() function only works with expressions of sc_signal<bool> types.

More complex expressions are possible using boolean expressions. For instance the statement below is also legal.

```
wait_until(clock.delayed() == true &&
           reset.delayed() == false);
```

## *Watching*

SC_THREAD and SC_CTHREAD processes 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.

An example is shown below:

```
// datagen.h file
#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 file
#include "systemc.h"
#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();
  }
}
```

This module is a simple data generator that will 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.

In the constructor of the example is the following statement:

```
watching(reset.delayed() == true);
```

This statement specifies that signal reset will be watched for this process. If signal reset changes to true then the watching expression will be true and the SystemC scheduler will halt execution of the while loop for this process and start the execution at the first line of the process.

The delayed() function is required for the signal in a watching expression in order for the description to compile properly. 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.

This behavior allows the designer to reset a design, or jump out of a loop, without having to check the reset condition at each wait statement. To enable this behavior for a particular process the watching statement must be added to the constructor, and the implementation of the method must look like below:

```
void data_gen::gen_data () {
  // variable declarations

  // watching code
  if (reset == true) {
    data = 0;
  }

  // infinite loop
  while (true) {
    // Normal process function
  }
}
```

The process will execute the normal process functionality until the watched condition becomes true. When this happens the loop will be exited and execution of the process will start at the beginning. In this example execution would start with the statement shown below:

```
  if (reset == true) {
```

If reset is true, then data would be set to 0 and execution of the loop would start again from the first 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.

One unexpected consequence of control exiting the while loop and starting again at the beginning of the process is that all of the variables defined locally within the process will lose their value. If a variable value is needed to be kept between invocations of the process, declare the variable in the process module, and not local to the process.

Multiple watches can be added to a process. The data type of the watched object must be of type bool. If multiple watches are added to a process be sure to test which watch expression triggered the exit from the loop. Then perform the appropriate watch action based on the expression that triggered the exit.

This type of watching is called global watching and cannot be disabled. If you need to watch different signals at different times, then use local watching discussed in the next section.

## *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. This functionality is specified with 4 macros that define the boundaries of each of the areas. A blank example is shown below:

```
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
```

The W_BEGIN macro marks the beginning of the local watching block. Between the W_BEGIN and W_DO macros are where all of the watching declarations are placed. These declarations look the same as the global watching events. Between the W_DO macro and the W_ESCAPE macro is where the process functionality is placed. This is the code that gets executed as long as none of the watching events occur. Between the W_ESCAPE and the W_END macros is where the event handlers reside. The event handlers will check to make sure that the relevant event has

occurred and then perform the necessary action for that event. The W_END macro ends the local watching block.

There are a few interesting things to note about local watching:

- 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.

To show an example of local watching let's modify the microcontroller bus example from the SC_CTHREAD description on page 60 and allow the bus controller to be interrupted during the memory to databus transfer, but not during the databus to memory transfer. We will add local watching to the second part of the while loop where data is transferred from the memory to the databus.

The new example is shown below:

```
// watchbus.cc

#include "systemc.h"
#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;
    datardy = false;   // cannot accept new address now
    data8 = taddr.range(7,0);
    start = true;      // new addr for memory controller
    wait();

    // wait 1 clock between data transfers
    data8 = taddr.range(15,8);
    start = false;
```

```
    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);

    W_BEGIN
      watching(reset.delayed());
      // Active value of reset will trigger watching

    W_DO
    // the rest of this block is as before

    // 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;
    datardy = true;   // data is ready, new addresses ok

    W_ESCAPE
      if (reset) {
        datardy = false;
      }
    W_END

  }
}
```

The second half of the model has been altered as shown by the statements in bold. Macro W_BEGIN marks the beginning of the watched area. Inside the watched area is a watching expression of signal reset. More than one watching expression can be put into the declaration area.

After macro W_DO is the statement area for the process functionality of the module. These statements are exactly the same as in the original model. The difference is that if signal reset becomes active, execution will be transferred to the handler statement area and not to the next statement in the block.

The W_ESCAPE macro marks the beginning of the handler area. This is the area where statement execution will be transferred if one of the watched events becomes active. Inside this area we have one handler for the reset event that is being watched. If there were more events being watched then a corresponding handler would be needed for each event.

Finally the W_END macro marks the end of the local watching block, any statements outside of this macro will only be subject to global watching not local watching.

## Triggering Processes with Events

In order to generate an event and trigger a process, the port the process is sensitive to must have an event. An important point to remember when trying to trigger a process is that in order to generate an event the input signal must change value. In the simplex example from Chapter 2 the retry field was added to the packet structure so that successive transmissions of the same packet would cause events. The retry field was updated on every packet transmission causing the new packet value to be different from the old and generating an event.

CHAPTER 5

# Ports and Signals

Ports of a module are the external interface that pass information to and from a module, and trigger actions within the module. Signals create connections between module ports allowing modules to communicate.

A port can have three different modes of operation.

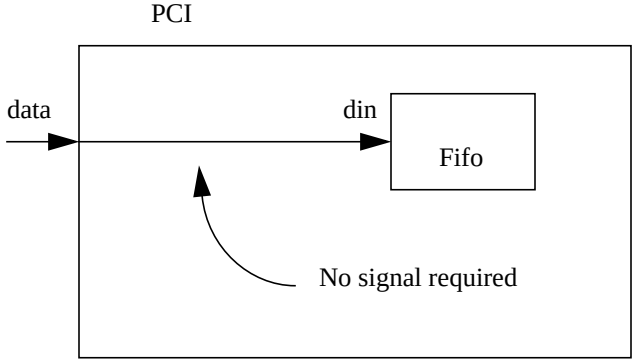
- Input
- Output
- InOut

An input port transfers data into a module. An output port transfers data out from a module, and an inout port transfers data both into and out of a module depending on module operation.

A signal connects the port of one module to the port of another module. The signal transfers data from one port to another as if the ports were directly connected. 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. This is done so that all operations within the process will work with the same value of the signal. This is to prevent 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. These signal semantics are the same as VHDL signal operation and Verilog deferred assignment behavior.

Ports are always bound to a signal except for one special case, when a port is bound directly to another port. Ports are always bound to only one signal. That signal may be a complex signal such as a structure, but it is still treated as one signal. 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. The special case binding mentioned earlier occurs when a top level module port is directly bound to a lower level module port during instantiation. This is shown in the figure below:




In this example port data of module PCI is directly connected to port din of module fifo. For this case no local signal is required.

Ports and signals also come in different sizes as hinted to earlier. Scalar ports have a single dimension. A scalar port can be one of the following types:

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

Input, output and inout ports are described using the following syntax as we have seen in a number of examples already:

```
sc_in<porttype>      // input port of type porttype
sc_out<porttype>     // output port of type porttype
```

```
sc_inout<porttype>   // inout port of type porttype
```

Type porttype can be any of the types mentioned above. Types will be described in more detail in Chapter 6, “Data Types,”.

## *Array Ports and Signals*

For some applications an array of ports might be desirable. For instance computer generated design descriptions might use an array of ports for mapping configurable sized objects. To declare an array port or signal the same syntax as C++ is used. An example is shown below:

```
sc_in<sc_logic> a[32];  // creates ports a[0] to a[31]
                        // of type sc_logic
```

This declaration creates an array of ports named a[0] to a[31] of type sc_logic. Each port has to be individually bound to a port, assigned, and read.

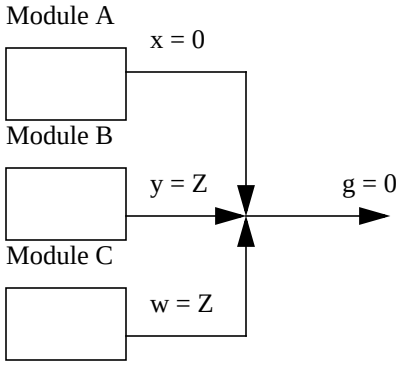
Signal arrays can be created using similar syntax. An example signal array is shown below:

```
sc_signal<sc_logic> i[16]; // creates signals i[0] to
                           // i[15] of type sc_logic
```

This statement creates an array of signals named i[0] to i[15] of type sc_logic. Each signal has to be individually bound to a port, assigned, and read.

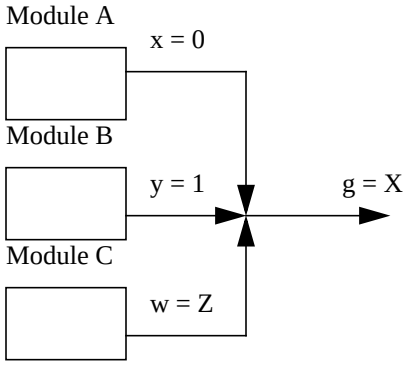
## *Resolved Logic Vectors*

Bus resolution becomes an issue when more than one driver is driving a signal. SystemC uses a Resolved Logic Vector signal type to handle this issue. Take a look at the example below with three drivers x, y, w, driving signal g.




Modules a, b, and c are driving signal g through ports x, y, and w respectively. Port x is driving a 0 value, and ports y and w are driving Z values. The resolution of these values will be assigned to signal g. In this example the resolved value will be 0. Ports y and w have their drivers disabled and are driving Z values. Therefore the 0 value from port x will “win”.

Another interesting case is shown below:



In this case ports x and y are driving a value while port w is not. However ports x and y are driving opposite values. Since values 0 and 1 are the same strength or priority the final value of signal g cannot be determined and the value assigned will be X.

To create a resolved logic vector port use the following syntax:

```
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
```

The only limitation on the size of n is underlying system limitations. Resolved Logic Vector ports should only be used where absolutely necessary as extra simulation overhead is added versus standard ports. Typically a standard port with a scalar or vector type should be used for better simulation efficiency.

## *Resolved Vector Signals*

Signals are used to interconnect ports. Vector signals can be used to connect vector ports. The vector signal types are the same as the vector port types. The currently supported vector signal type is sc_signal_rv. This is a resolved vector of sc_logic signals. An example is shown below:

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

Signals of this type can be used to connect to resolved logic vector ports.

## *Signal Binding*

As mentioned previously each port is bound to a single signal. When reading a port the variable assigned the port value must have the same type as the port type. For example a port of type sc_logic cannot be read into an int variable or signal.

When ports are bound to other signals or ports, both types must match. The example below shows a port bound to another port (special case) and a port bound to a signal.

```
// 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 “systemc.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”);
    s1->clock(clk);    // special case port to
                       // port binding
    s1->en(lstat);     // port en bound to signal lstat
    s1->dir(down);     // port dir bound to signal down
    s1->st(status);    // special case port to
                       // port binding
  }
};
```

This example shows a controller module with a number of input and output ports. The module also includes local signals lstat and down. The controller module instantiates module state_machine with an instance label of s1. Below the state machine instance are the port binding statements. The first statement:

```
s1->clock(clk);
```

binds port clock of instance s1 to external port clk of the controller. This is an example of a special case binding in which a port is bound directly to another port instead of a signal. The second port binding is shown below:

```
s1->en(lstat);
```

This statement binds port en of s1 to local signal lstat. This is an example of Named Mapping as discussed in “Named Connection” on page 43. Positional Mapping is discussed in “Positional Connection” on page 42.

# Clocks

Clock objects are special objects in SystemC. They generate timing signals used to synchronize events in the simulation. Clocks order events in time so that parallel events in hardware are properly modeled by a simulator on a sequential computer.

A clock object has a number of data members to store clock settings, and methods to perform clock actions. To create a clock object use the following syntax:

```
sc_clock  clock1(“clock1”, 20, 0.5, 2, true);
```

This declaration will create a clock object named clock with a period of 20 time units, a duty cycle of 50%, the first edge will occur at 2 time units, and the first value will be true. All of these arguments have default values except for the clock name. The period defaults to 1, the duty cycle to 0.5, the first edge to 0, and the first value to true.

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. This allows areas of the design or the entire design to be synchronized by the same clock. In the example below the sc_main routine of a design creates a clock and connects the clock to instantiated components within the main module.

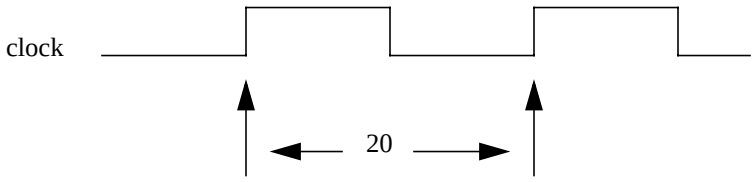
```
int sc_main(int argc, char*argv[]) : sc_module {
  sc_signal<int>  val;
  sc_signal<sc_logic> load;
  sc_signal<sc_logic> reset;
  sc_signal<int> result;

  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
```

```
}
```

In this example the top level routine sc_main instantiates a module called filter and declares some local signals that will connect filter with other module instantiations. Notice that a clk signal is not declared, instead a clock object is instantiated, its parameters are setup, and its signal method is used to provide the clock signal. Function ck1.signal() is mapped to the clk port of the filter object.

In this example the clock is named ck1 and the clock frequency is specified as 20 time units. Every 20 time units the clock will make a complete transition from true to false and back to true as shown by the following figure.

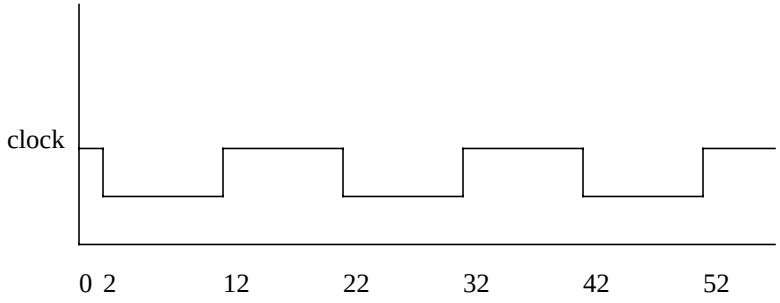
**FIGURE 2. Clock Waveform:**




The duty cycle of the clock is the ratio of the high time to the entire clock period. In this example the duty cycle is specified as 0.5. This means that the clock will be true for 10 time units and false for 10 time units. If the duty cycle were specified as 0.25 then the clock would be true for 5 time units and false for 15 time units.

The next parameter of the clock object is the start time of the first edge. This is a time offset from 0 of the first edge, expressed in time units. In this example the specified value is 2 time units. The last argument is the starting value of the clock object. The clock object will toggle the clock signal at appropriate times, but this value is used to specify the first value of the clock. Based on the parameters specified the clock object will produce a clock signal as shown in Figure 2 below:

**FIGURE 3. Offset Clock Waveform**




When binding the clock to a port the designer must use the clock signal generated by the clock object to map to a port. This done by using the signal method of the clock object. Notice that the clk port of filter is mapped to ck1.signal(). This is the clock signal generated by the clock object.

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

CHAPTER 6

# Data Types

SystemC provides the designer the ability to use any and all C++ data types as well as unique SystemC data types to model systems.

C++ data types are discussed in C++ books, so they will not be discussed here. The SystemC data types include the following:

- sc_bit – 2 value single bit type
- sc_logic – 4 value single bit type
- sc_int – 1 to 64 bit signed integer type
- 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

Each of these types will be discussed in more detail in the next sections. The fixed point types are described in more detail in the next chapter.

## *Type sc_bit*

Type sc_bit is a two valued data type representing a single bit. A variable of type sc_bit can have the value ‘0’(false) or ‘1’(true) only. This type is useful for modeling parts of the design where Z (hi impedance) or X (unknown) values are not needed.

There are a number of logical and comparison operators that work with sc_bit including:

**TABLE 1. sc_bit Operators**

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

For those not familiar with the special assignment operators of C/C++ here is how these work. In a typical language the designer might write:

```
a = a & b;
a = a | b
```

In C++ this can also be written as:

```
a &= b
a |= b
```

Values are assigned using the character literals ‘1’ and ‘0’. When performing boolean operations type sc_bit objects can be mixed with the C/C++ bool type. Objects of type sc_bit are good for representing single bits of a design where logical opera-

tions will be performed. To declare an object of type sc_bit use the following syntax.

```
sc_bit s;
```

## *Type sc_logic*

A more general single bit type is sc_logic. This type has 4 values, ‘0’(false), ‘1’(true), ‘X’ (unknown), and ‘Z’ (hi impedance or floating). This type can be used to model designs with multi driver busses, X propagation, startup values, and floating busses. Type sc_logic has the most common values used in VHDL and Verilog simulations at the RTL level.

Type sc_logic has a number of logical, comparison, and assignment operators that can be used with objects of this type. These include the following:

**TABLE 2. sc_logic Operators**

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

These operators are implemented such that operands of type sc_logic can be mixed with operands of type sc_bit. One of the operands must be type sc_logic, the other operands can be sc_logic or sc_bit.

Values are assigned to sc_logic objects using the character literals shown below:

- ‘0’ – 0 or false value
- ‘1’ – 1 or true value

- ‘X’ – unknown or indeterminate value
- ‘Z’ – hi impedance or floating value

An example assignment is shown below:

```
sc_logic x;  // object declaration

x = ‘1’;    // assign a 1 value
x = ‘Z’;    // assign a Z value
```

The comparison operators == and != are implemented so that a designer can compare two sc_logic objects, an sc_logic object and an sc_bit object, or an sc_logic object and one of the character literal values. The following comparisons are implemented:

```
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
```

The assignment operator allows assigning a character literal value or another sc_logic object to an sc_logic object. Additionally an sc_bit can be converted to an sc_logic through the assignment. The following assignments are conversions.

```
sc_bit x;
sc_logic y;

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

The first assignment will convert an sc_logic type to an sc_bit type. Since an sc_bit object has 2 values while an sc_logic type has 4 values, the values ‘Z’ and ‘X’ cannot be converted to an sc_bit. If the value of the sc_logic object is ‘Z’ or ‘X’ when

assignment occurs, the result of the assignment is undefined and a runtime warning is issued.

## *Fixed Precision Unsigned and Signed Integers*

Some systems need arithmetic operations on fixed size arithmetic operands. The Signed and Unsigned Fixed Precision Integer types provide this functionality in SystemC. The C++ int type is machine dependent, but usually 32 bits. If the designer were only going to use 32 bit arithmetic operations then this type would work. However the SystemC integer type provides integers from 1 to 64 bits in signed and unsigned forms.

The underlying implementation of the fixed precision type is a 64 bit integer. All operations are performed with a 64 bit integer and then converted to the appropriate result size through truncation. If the designer multiplies two 44 bit integers the maximum result size is 64 bits, so only 64 bits are retained. If the result is now assigned to a 44 bit result, 20 bits are removed. If more precision is needed use Arbitrary Precision Integers described in the next section. The fastest simulation speed will be obtained by using the built-in C++ data types int, long, etc. However these types only work for a fixed data size of 8, 16 or 32 bits. The second fastest simulation speed can be obtained by using the Fixed Precision Integers. The slowest simulation time will come from using the Arbitrary Precision Integers. So whenever possible use the Fixed Precision Integers over Arbitrary Precision Integers for the fastest simulation speed.

Type sc_int<n> is a Fixed Precision Signed Integer, while type sc_uint<n> is a Fixed Precision Unsigned Integer. The signed type is represented using a 2's complement notation. The underlying operations use 64 bits, but the result size is determined at object declaration. For instance the following declaration declares a 64 bit unsigned integer and a 48 bit unsigned integer.

```
sc_int<64>  x;
```

```
sc_uint<48>  y;
```

Integer types have a very rich set of operators that work with them as shown by the list below:

**TABLE 3. Fixed Precision Integer Operators**

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

Bitwise operators work on operands bit by bit. The not(~) operator will invert all bits, and the shift operators will shift left(<<) or right(>>) an operand by the specified number of bits. An example is shown below:

```
sc_int<16> x, y, z;

z = x & y;   // perform and operation on x and y bit
             // by bit
z = x >> 4;  // assign x shifted right by 4 bits to z
```

With the addition of arithmetic operators for SystemC Integer types, new assignment operators are also available. For instance the += operator will allow a more terse description of the following statement:

```
x = x + y;   // traditional way
```

```
x += y;   // terse method
```

To select one bit of an integer use the bit select operator as shown below:

```
sc_logic mybit;
sc_uint<8> myint;

mybit = myint[7];
```

To select more than one bit use the range method as shown below:

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

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

Finally the concatenation operator can be used to make a larger value from one or more smaller values. An example is shown below:

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

intc = (inta, intb);
```

Operands inta, and intb are concatenated together to form an 8 bit integer and then assigned to integer intc.

The auto increment and auto decrement operators are another method of making the description more concise and terse. The auto increment operator will increment the operand it is attached to, and the auto decrement operator will decrement the operand. For instance instead of writing:

```
a = a + 1;
```

The auto increment operator will allow:

```
a++;
```

Variable of type sc_uint (unsigned) can be converted to type sc_int (signed) with the = (assignment) operator. In the same way variables of type sc_int can be converted to sc_uint. When the = operator is used any extra bits are removed and sign bits are added and extended as necessary. An example is shown below:

```
sc_uint<8> uint1, uint2;
sc_int<16> int1, int2;

uint1 = int2;  // convert int to uint
int1 = uint2;  // convert uint to int
```

In the first statement an integer is converted to an unsigned integer. The absolute value of int2 will be assigned to uint1. If int2 is a negative value only the magnitude will be assigned to uint1. Since int2 is 16 bits while uint1 is 8 bits uint2 will be converted to a 64 bit unsigned number and then truncated to 8 bits before assignment to uint1.

In the second statement uint2 is assigned to int1. First uint2 will be converted to a 64 bit signed value then truncated and assigned to int1.

Type sc_int and sc_uint can be used with C++ integer types without restriction. C++ integer types can be freely mixed with SystemC types.

## *Speed Issues*

As previously mentioned when SystemC integers are used 64 bits of precision are used. However if no more than 32 bits are ever needed simulation speed can be increased by using 32 bits for the underlying precision. This is accomplished by

compiling with a special compiler flag, -D_32BIT_. This compile flag will limit the size of the underlying arithmetic precision to 32 bits instead of 64.

## *Arbitrary Precision Signed and Unsigned Integer Types*

There are cases in HDL based design where operands need to be larger than 64 bits. For these types of designs sc_int and sc_uint will not work. For these cases use type sc_biguint (arbitrary size unsigned integer) or sc_bigint (arbitrary sized signed integer). These types allow the designer to work on integers of any size, limited only by underlying system limitations. Arithmetic and other operators also use arbitrary precision when performing operations. Of course this extra functionality comes at a price. These types execute more slowly than their fixed precision counterparts and therefore should only be used when necessary. While sc_bigint and sc_biguint will work with any operand sizes, they should only be used on operands larger than 64 bits or for operations where more than 64 bits of precision are required.

Type sc_bigint is a 2's complement signed integer of any size. Type sc_biguint is an unsigned integer of any size. When using arbitrary precision integers the precision used for the calculations depends on the sizes of the operands used. Look at the example below:

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

b3 = b1*b2;
```

In this example b1, a 128 bit integer is multiplied by b2, a 64 bit integer. The result will be a 192 bit integer. However since b3 is only 150 bits wide 42 bits will be removed from the result before assignment to b3.

For performance reasons a variable named MAX_NBITS is defined in sc_constants.h. This constant specifies the maximum number of bits to be used for an arbitrary precision integer operation. Defining this variable provides a 2-3X per-

formance increase. The default value is 512, but can be changed if larger operands are required.

The same operators used for Fixed Precision Integers are also available for Arbitrary Precision Integers. These operators are shown in the table below:

**TABLE 4. Arbitrary Precision Integer Operators**

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

These operators use arbitrary precision for the underlying operations, unlike the fixed precision types. The real difference between the two types is the underlying precision and the slower simulation speed. Arbitrary Precision Integer types can have much greater precision but may simulate much slower so their use should be limited to only where needed.

Types sc_biguint, sc_bigint, sc_int, sc_uint, and C++ integer types can all be mixed together in expressions. Also the = operator can be used to convert from one type to another.

## *Arbitrary Length Bit Vector*

SystemC also contains a 2 valued arbitrary length vector to be used for large bit_vector manipulation. If the designer does not need tristate capability and no arithmetic operations are to be performed directly on the data, then sc_bv is the ideal type for the object. The sc_bv type will simulate faster than the sc_lv type yet still allow data manipulations on very large vectors. Type sc_biguint could also be used for these operations but type sc_biguint is optimized for arithmetic operations, not bit manipulation operations and type sc_bv will simulate faster.

The sc_bv type introduces some new operators that perform bit reduction. These operators take the entire set of bits of the operand and generate a single bit result. For instance to find out if databus is all 0's the following operation could be performed:

```
sc_bv<64> databus;
sc_logic result;

result = databus.or_reduce();
```

If databus contains 1 or more 1 values the result of the reduction will be 1. If no 1 values are present the result of the reduction will be 0 indicating that databus is all 0's.

Bit selection, part selection and concatenation all work with sc_bv objects. Remember these operators work on both sides of an assignment operator and in expressions. The following expressions are legal.

```
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);
```

In the first example a range of a large sc_bv object is assigned a smaller sc_bv object. In the second example a small sc_bv object is assigned the concatenation of two fields from a larger sc_bv object. In the final example a concatenated range of a smaller sc_bv object is assigned a range from a large sc_bv object.

The operations supported by sc_bv are shown in the table below:

**TABLE 5. Arbitrary Length Bit Vector Operators**

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

A single bit can be selected from an sc_bv object using the bit selection operator []. An example is shown below:

```
sc_bit y;
sc_bv<8> x;

y = x[6];
```

More than one bit can be selected using part selection. Part selection uses the range function to specify the range of bits to select. An example is shown below:

```
sc_bv<16> x;
sc_bv<8>  y;

y = x.range(0,7);
```

Notice that sc_bv types cannot have arithmetic performed directly on them. To perform arithmetic functions first assign sc_bv objects to the appropriate SystemC integer. Perform the arithmetic operation on the integer type. If the application war-

rants then copy the results of the arithmetic operations 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.

The = operator will convert sc_bv objects to sc_lv objects and vice versa. Strings of ‘0’ and ‘1’ characters can be assigned to type sc_bv objects. For instance to set a 16 bit sc_bv to all 1’s use the following statement:

```
sc_bv<16>  val;

val = “1111111111111111”;
```

Only the values ‘0’ and ‘1’ can be assigned and the string must be at least as long as the sc_bv object.

## *Arbitrary Length Logic Vector*

Different data types are used to model the types of data used in a typical design. Types sc_logic and sc_bit work well for modeling single bits accurately. Types sc_int, sc_uint, sc_bigint, and sc_biguint work well for modeling parts of the design where arithmetic operations can occur, but no tristate busses. However for parts of the design that need to be modeled with tristate capabilities yet contain items that are wider than 1 bit, SystemC contains a type called sc_lv<n>. This type represents an arbitrary length vector value where each bit can have one of four values. These values are exactly the same as the four values of type sc_logic. Type sc_lv<n> is really just a variable sized array of sc_logic objects.

To declare a signal of type sc_lv<n> use the following syntax:

```
sc_signal<sc_lv<64> > databus; // extra space is
                               // required
```

This declaration describes a 64 bit wide signal called databus in which each of the bits of the signal can have the value ‘0’, ‘1’, ‘X’, and ‘Z’. This signal can be driven by a number of sources to model a tristate bus.

It is very important to note that the extra space after the first > is required to allow the declaration to compile.

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. The design implemented with sc_bv will simulate much faster than the design implemented with sc_lv.

Notice that sc_lv types cannot have arithmetic performed directly on them. To perform arithmetic functions first assign sc_lv objects to the appropriate SystemC integer. Perform the arithmetic operation on the integer type. If the application warrants then copy the results of the arithmetic operations back to the sc_lv type. The = operator is overloaded to allow assignment of a sc_lv type to a SystemC integer and vice versa.

To convert an sc_lv type to an arithmetic type use the = operator. This is shown below:

```
sc_uint<16> uint16;
sc_int<16> int16;
sc_lv<16> lv16;

lv16= uint16;  // convert uint to lv
int16 = lv16;  // convert lv to int
```

The first statement converts an unsigned integer to a logic vector 16 bits wide. The second statement converts a logic vector to a 16 bit signed integer. Any X's or Z's in the logic vector will produce a runtime warning and the results will be undefined.

A common function needed to properly model a tristate bus is the ability to turn off all drivers to the bus. To perform this step assign a string of 'Z' values to the sc_lv object. This is shown below:

```
sc_lv<16> bus1;

if (enable) {
  bus1 = in1
} else {
  bus1 = "ZZZZZZZZZZZZZZZZ";
```

```
}
```

This assignment will assign a Z value to all 16 locations of bus1. The character string has to be at least as long as the logic vector object. The character string can contain any combination of the four values, ‘0’, ‘1’, ‘X’, and ‘Z’. So another legal string for bus1 would be the following:

```
bus1 = “01XZ01XZ01XZ01XZ”;
```

To print a human readable character string of the value from an sc_lv object use the to_string() method as shown:

```
sc_lv<32> bus2;

cout << “bus = “ << bus2.to_string();
```

## *Logic Vector Speed Issues*

The bit vector type will always simulate faster than the logic vector type. When creating a design try to use the bit vector types over the logic vector types as much as possible. The logic vector type will be needed to model the cases where the reset behavior of the design is important or the vector will be used in a tristate environment. For all other cases the bit vector type should be used to create the fastest simulation.

## *User Defined Type Issues*

### Comparison Operator

For scalar types the built-in comparison operators are used to determine whether or not a value changed, which generate an event. For user defined types such as

packet_type used in the simplex example in Chapter 2 the designer needs to provide the == operator. Looking back at packet.h we see the following:

```
inline bool operator == (const packet_type& rhs) const
  {
  return (rhs.info == info && rhs.seq == seq &&
     rhs.retry == retry);
  }
```

This method defines the fields that are to be compared and how the comparison is made. An event occurs if the comparison result indicates that the previous packet and new packet are different.

### Tracing a User Defined Type

Notice that the packet.h file from Chapter 2 also traces the signals of user defined type packet_type. Because this type has a number of fields, you need to specify tracing of each field to see the contents of the packet. This process is not automatic. You can define a special user trace method that is called when an object of this type is traced. This user method can be defined in the user defined type.

Looking back at files packet.h and packet.cc from Chapter 2, we can see that the user defined type packet_type has an sc_trace method defined in packet.h. This method defines how to trace an object of type packet_type. The declaration of the method, the argument types, and the return value in the packet.h file is shown below:

```
extern
void sc_trace(sc_trace_file *tf, const packet_type& v,
const sc_string& NAME);
```

Notice that the second argument is of type packet_type, which makes this method unique. File packet.cc contains the implementation of the sc_trace method as shown below:

```
void sc_trace(sc_trace_file *tf, const packet_type& v,
 const sc_string& NAME) {
   sc_trace(tf,v.info, NAME + ".info");
   sc_trace(tf,v.seq, NAME + ".seq");
   sc_trace(tf,v.retry, NAME + ".retry");
}
```

The implementation of the trace method has a trace for each field of the struct. This method is called by the designer to perform a trace on a signal of type packet_type, and is automatically created by the compiler. Each call to the trace method will perform a trace on all of the fields of the user defined type.

CHAPTER 7

# System Design in SystemC

This chapter presents higher levels of abstraction in SystemC for describing behavior inside modules as well as inter-module communication. SystemC allows you to separate communication of a module from its internal behavior to a large degree (a complete separation is not always possible).

## Benefits of Separating Communication from Behavior

The two primary benefits of separating communication from behavior are the ease with which you can embed IP (Intellectual Property) blocks into a SoC (System on a Chip) and refine communication protocols.

### Embedding of IP Blocks

Because communication is separate from behavior, you can more easily embed an IP block such as a processor core into a SoC. High-level communication protocols at the functional level can be more easily mapped to the architecture of a particular core, relieving the designer of the details of bus timing, pin mapping, etc.

### *Refining Communication Protocols*

SystemC allows you to gradually refine communication by simply assigning a new protocol to a port or a communication link. You do not have to change the interconnection between modules. For example, you can define an abstract communication protocol at the functional level, then refine it to a FIFO communication link or a bus. This enables modular design and design re-use.

## *Abstraction Levels in SystemC*

The abstraction levels supported in SystemC are presented in this section.

### *Untimed Functional (UTF) Level*

At the untimed functional level, you create an executable specification of the system without defining the system architecture. The system is decomposed into functional modules that communicate over abstract communication channels. System behavior is modeled in a maximally sequential form, which usually requires least effort. However, when needed, concurrent behavior can be specified with synchronization between concurrent threads. Data transactions are modeled accurately but the notion of time is absent. All processes execute in zero time but in a (mostly) defined order.

### *Timed Functional (TF) Level*

At the timed functional level, some functional processes can be assigned a “run time” or duration while others may be untimed. The concept of “time” in TF should not be confused with the clock paradigm of digital synchronous systems. In this sense, a TF system is timed but not clocked.

### *Bus-Cycle Accurate (BCA) Level*

At the bus-cycle accurate level, the processes in the system are synchronized using a clock signal. You model the transactions on the bus cycle-accurately, but some behavior may be left at an untimed level.

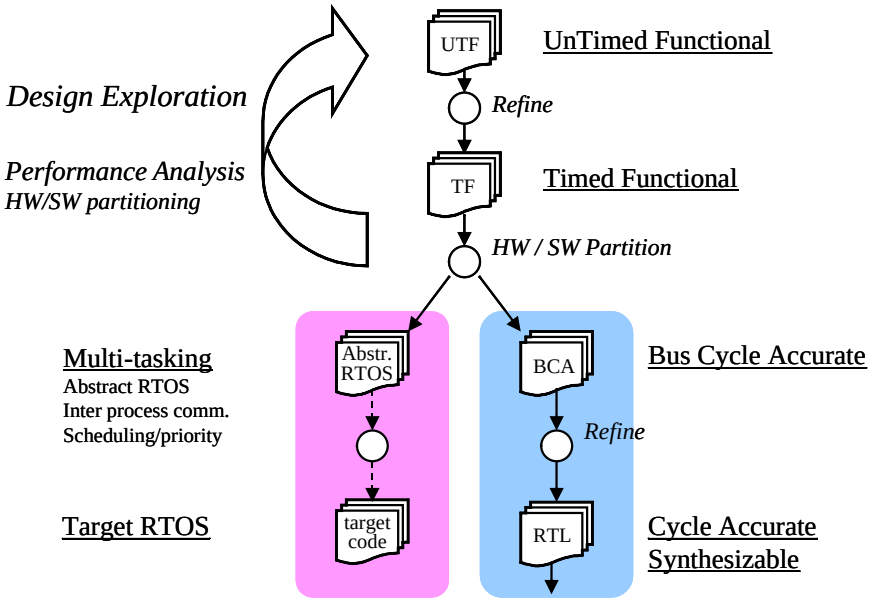
### *Cycle Accurate (CA) Level*

At the cycle accurate level, behavior is clock cycle accurate. This is already well described throughout this User Guide and will not be discussed further in this chapter.

## *Design Flow in SystemC*

The SystemC design flow is shown in Figure 4.

**FIGURE 4. SystemC Design Flow**



## *Untimed Functional (UTF) Level*

The UTF level is well suited to develop an executable specification of a complete system with both hardware (HW) and software (SW) components. The system architecture, system timing, and details of inter-block communication are abstracted at this level. There is no distinction between hardware (HW) and software (SW). The system designer is interested in modeling the algorithmic behavior and a functional decomposition of the system.

System behavior is described in a maximally sequential form. At the functional level (UTF & TF) this is done using the Remote Procedure Call (RPC) protocol, which defines a sequential control and data flow between functional modules of the system.

Concurrent behavior should be minimal whenever possible for two reasons:

- Concurrent behavior is in general harder to describe.
- Concurrency reflects an architectural decision and should best be deferred to a later stage in the design process.

However, system specifications, which are concurrent in nature, should of course preserve this concurrency in the executable model.

The RPC protocol allows a master[1] process in a module to invoke a remote function, called the slave process, in a different module. The slave function executes in zero time and returns control to the master. RPC defines a sequential, untimed data and control flow. The control flow is data dependent.

Note that at this level functional processes communicate over specialized master-slave ports, which are called *abstract* ports.

## *Timed Functional (TF) Level*

Sequential and concurrent modeling paradigms in TF are the same as in UTF with the distinction that, in TF, processes may be assigned a “run time”. TF is very useful for high-level performance modeling and time budgeting. A system may be described as an interconnection of timed and untimed processes.

1. The calling process in an RPC is sometimes called a master process.

Note that the process execution order of UTF is not changed in TF since the order is still defined by RPC-chains and signaling between concurrent threads.

A process run time may reflect different things:

- A system specification for the maximum execution time of a process.
- An estimated execution time of a process for performance modeling.
- A budgeted execution time of a process.

In SystemC, you assign a run time to a process through *waitfor(delta)*[1] statements, which put a process to sleep for a time duration of *delta.* The process will continue execution after a delay of *delta*.

In TF, time is used to express duration only. Synchronization and ordering of processes is achieved through signaling and RPC. In this sense, a TF system is timed but not clocked.

### *Hardware-Software Partitioning*

Design exploration, performance modeling and analysis together with hardware (HW) - software (SW) partitioning are done at the functional level since this is much easier and faster at the functional level than at lower levels of abstraction. After functional design, HW modules are mapped to an implementation architecture. SW modules are partitioned into tasks and inter-task communication, and synchronization is implemented using an RTOS (Real Time Operating System). The SW flow in SystemC is currently under development. Therefore, we will focus the rest of this discussion on the HW flow.

In the HW flow, the communication protocols between modules will be refined and functional modules transformed into cycle accurate modules. This takes place over different design steps with a gradual refinement of abstraction levels. A mixture of abstraction levels is possible whereby some modules can be at a functional level while others are at a bus cycle accurate or RTL levels.

1. Not implemented in release version 1.1 of SystemC. In SystemC v1.1, you can emulate a *delta* delay by using wait()'s on a timer clock. Note that the clock is used here as a timer and not as a synchronizer.

### *Bus-Cycle Accurate (BCA) Level*

The BCA level is a useful abstraction to model hardware (HW) with bus architectures. At the bus-cycle accurate level, the processes in the system are synchronized using a clock. You model the transactions on the bus cycle-accurately, but some behavior may be left at an untimed level. This allows a gradual refinement of the design.

Abstract ports of the functional level are now refined to bus ports, which may have data, address, and control terminals. In addition, a bus protocol, which defines the communication protocol of the port, may be associated with bus ports. Examples of communication protocols are no-handshake, enable-handshake, and full-handshake protocols. The bus protocol specification reflects design intent but does not implement the protocol behavior. It must be implemented explicitly by the user in the body of the process that is attached to the port.

Cycle Accurate Level is not discussed here because this level is already described in the previous chapters.

## *The RPC Mechanism*

With RPC, a process in a module can call a function in another module. The called function should also be a SystemC process, which is called the *slave* process. This is very similar to RPC semantics in Unix. The two processes must be connected through specialized ports to a specialized communication link. The calling process connects to the link using a master port, while the slave connects using its slave port.

The calling process invokes the slave process by a read from or a write to its master port. Upon invocation, the slave process executes and returns as with a normal function call.

A slave process is a process of SC_SLAVE type. The member function that is registered as the SC_SLAVE process is the function that will be called by the master. The slave process has one and only one slave port. The slave function can only be invoked through this unique slave port.

Additionally, a slave process may have one or more master ports of its own through which it can invoke other slave processes.

The calling process can be of any type (i.e. SC_METHOD, SC_THREAD, or SC_CTHREAD) and can have multiple master ports through which it can invoke multiple slaves.

### *Abstract Ports*

Master and slave ports are specialized port classes, called *abstract* ports. They have a data type and a direction (in, out, or inout).

There are a total of eight abstract port types. <T> stands for the data type communicated over the port.

- sc_master<>, sc_inmaster<T>, sc_outmaster<T>, sc_inoutmaster<T>
- sc_slave<>, sc_inslave<T>, sc_outslave<T>, sc_inoutslave<T>

### *Communication Links*

Master-slave processes communicate over multi-point link objects. A multi-point link also has a data type T.

Syntax for links is:

```
sc_link_mp<T>  link1;
```

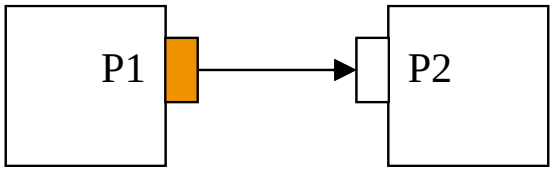
Ports connected to a link must have the same data type as the link. Abstract ports must not be bound to signals.

The rules for binding ports to links or other ports are the same as for primitive[1] ports (see Chapter 3, “Modules and Hierarchy” starting on page 39).

1. sc_in, sc_out, and sc_inout (used at RTL level) are called primitive ports to distinguish them from ports at higher abstraction levels (abstract and bus ports).

Figure 5 shows an example of RPC communication. By convention, a master port is represented as a black box and a slave port as a white box.

**FIGURE 5. RPC Example**




In the following example, a producer process P1 (master) produces a set of numbers and for each number invokes the consumer process P2 (slave), which accumulates the numbers and prints out the result.

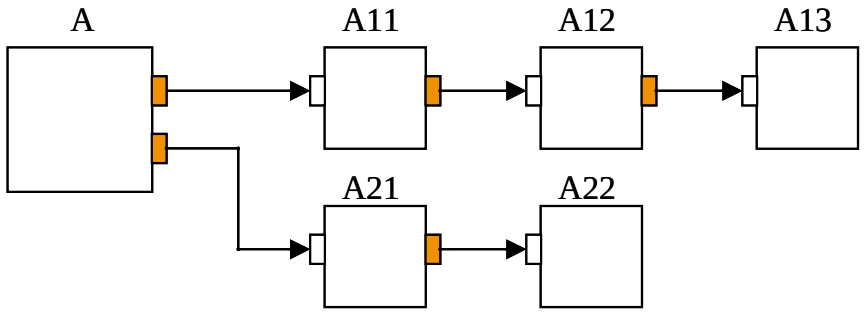
```
SC_MODULE(producer) {
    sc_outmaster<int>  out1;
    sc_in<bool>         start;

    void generate_data() {
        for (int i = 0;  i < 10;  i++) {
           out1 = i ;// this will invoke the slave;
        }
    }

    SC_CTOR(producer) {
        SC_METHOD(generate_data);
        sensitive << start;
    }
};

SC_MODULE(consumer) {
    sc_inslave<int>in1;
    int  sum; // declare as a module state variable

    void accumulate() {
        sum += in1;
        cout << “Sum =   “  << sum << endl;
    }

    SC_CTOR(consumer) {
        SC_SLAVE(accumulate, in1);
        sum = 0;// initialize the accumulator
```

```
    }
};

SC_MODULE(top) { // structural module
producer *A1;
consumer *B1;
    sc_link_mp<int> link1;

    SC_CTOR(top) {
        A1 = new producer("A1");
        A1.out1(link1);
        B1 = new consumer("B1");
        B1.in1(link1);
    }
};
```

## *RPC Chains*

A slave process can have one or more master ports that are connected to other slave processes such that RPC chains can be formed. This is illustrated with the example in Figure 6.

**FIGURE 6. RPC Chain Example**




Process A connects over two master ports to slaves A11 and A21. A11 connects over its master ports to slave A12. A12 connects to slave A13, and A21 connects to slave A22. The execution order in this example may be A, A11, A12, A13, A21, and A22.

RPC chains cannot form a loop since this would require a slave process to have more than one slave port.

### *Autonomous Process*

A process that has only master ports is called an autonomous process. An autonomous process forms the root process of an RPC-chain. In Figure 6, process A is an autonomous process. An autonomous process with all its RPC-chains forms a single execution thread.

### *Concurrency in UTF*

Concurrency in UTF of course has no relationship to time. Concurrency model in UTF is the following:

All autonomous threads in a system form concurrent execution threads. Concurrent threads can synchronize and communicate through signals. The only order relationship between concurrent threads exists through this synchronization.

RPC communication between two concurrent threads is per definition not possible since this would require them to be in the same thread.

### *Slave Process Types*

There are two types of slave processes: non-blocking-slave and blocking-slave types. In the non-blocking-slave, the slave process is a pure function and does not have any wait() statements that can suspend the execution of the slave process. The calling process (autonomous or slave) calls the slave process and blocks until the slave returns as in a normal function call.

In blocking-slave, the slave process is a function with embedded wait() statements to allow for synchronization with signals.

Note that blocking and non-blocking descriptions refer to the behavior of the slave process, not the calling process. The latter always blocks until the slave returns.

### *Autonomous Process Types*

In a blocking-slave process, the autonomous process must be a thread process, either SC_THREAD or SC_CTHREAD to allow blocking in the slave process. If the autonomous process is not a thread process, a run-time error will occur.

In a non-blocking-slave, the autonomous process can be of any type.

### *Slave Process*

The syntax for a non-blocking slave type is shown below. The slave process takes two arguments: *slave_method*, which implements the behavior of slave process, and the unique *slave_port*, which must be an abstract port.

```
SC_CTOR(my_mod) {
    // module constructor
    …
    SC_SLAVE(slave_method, slave_port);
    // defines slave process with its unique slave port;
    …
}
```

A blocking-slave example is shown below. With a blocking-slave process, a sensitivity list may be specified, which defines trigger signals for embedded wait() statements in the slave. If no sensitivity list is specified with the slave, the sensitivity list of the autonomous process will be active. If a sensitivity list is specified for a slave, then the slave's sensitivity list will be active for wait()'s inside the slave. The implication of this is that, in an RPC-chain, the sensitivity list that is active depends on which slave the RPC-chain is blocked. This is called *dynamic sensitivity*[1].

Note that the slave process is only invoked from its slave port. Events on signals in the sensitivity list of the slave will not invoke the slave. Such events can only wake up a slave that is sleeping at a wait() statement.

```
SC_CTOR(my_mod) {
    // module constructor
    …
    SC_SLAVE(slave_method, slave_port);
    // defines slave process with
    // its unique slave port;
    sensitive << a << b;   // sensitivity list
                          // for blocking RPC
    …
}
```

1. Dynamic sensitivity is not implemented in SystemC version 1.1 release. In version 1.1, all slaves in an RPC-chain share the sensitivity list of the autonomous process.

### *Slave Process Rules*

The following rules hold for slave processes:

- A slave port must be an abstract port type. You cannot specify a primitive port as a slave port.
- The slave port of a slave process must not also be in its sensitivity list. This could give rise to a deadlock during simulation.
- You can specify primitive input ports, terminals of bus ports (see “Bus Port Classes” on page 125), and signals in the sensitivity list of a slave.

### *Point-to-Point (p2p) and Multi-Point (mp) Communication*

In a *p2p* communication, one master process communicates with one slave process. In a *mp* communication, a master process communicates with multiple slaves as follows: a master performs an access to its master port, which will invoke all slaves. Slaves are invoked sequentially but in an undefined order. Each slave executes and returns, then the next slave is invoked. If a slave blocks at a wait(), then the remaining slaves will block as well.

### *Multi-Point Links*

Multiple master and slave processes can be interconnected in a multi-point link.

#### Rules

The following rule is enforced for multi-point connections: with each master port connection, there should at least be one slave connection that can respond to a transaction initiated by the master. This rule is statically checked at the start of a simulation.

In practice, this means the following:

- With an sc_inmaster port in a link, there must at least be one sc_outslave or one sc_inoutslave connection.
- With an sc_outmaster port in a link, there must at least be one sc_inslave or one sc_inoutslave connection.

- With an sc_inoutmaster port in a link, there must at least be one sc_inslave and one sc_outslave or one sc_inoutslave connection.

### Access Operations in a Multi-Point Link

When a sc_inmaster makes a read access, all slave processes connected to the link that are of a compatible port type (sc_outslave or sc_inoutslave) will be invoked, but only one must respond with a write operation to its slave port. This condition can be satisfied by an indexed access (see “Indexed Ports” on page 120) in which the master selects the slave by specifying an address with the transaction. Failing to do so will result in a run-time error with unpredictable simulation results.

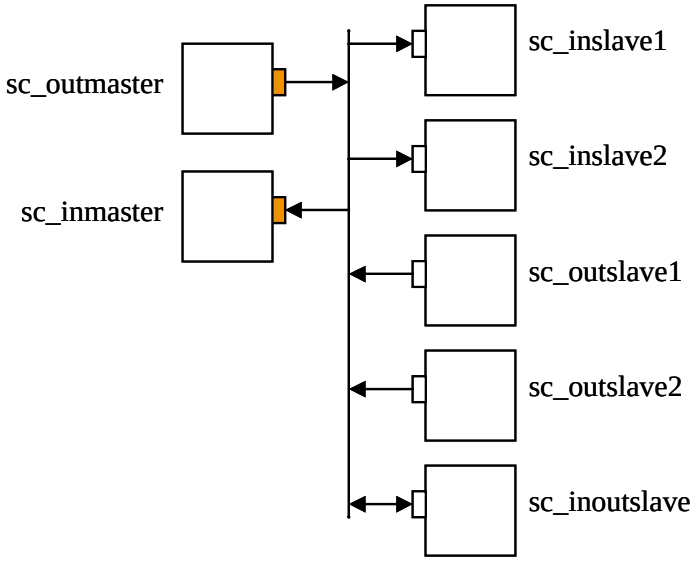
For a write operation to a sc_outmaster port, all slave processes connected to the link with compatible slave ports (sc_inslave or sc_inoutslave) will be invoked and perform a read operation from the slave port. This constitutes broadcast communication.

It is legal to have different autonomous processes connected to the same link, which can initiate transactions in the same delta cycle. The transactions will be performed sequentially but the order of transactions is undefined.

## Multi-Point Link Example

A multi-point link is illustrated in Figure 7, where two autonomous processes are connected to multiple slaves.

**FIGURE 7. Multi-Point Link Example**

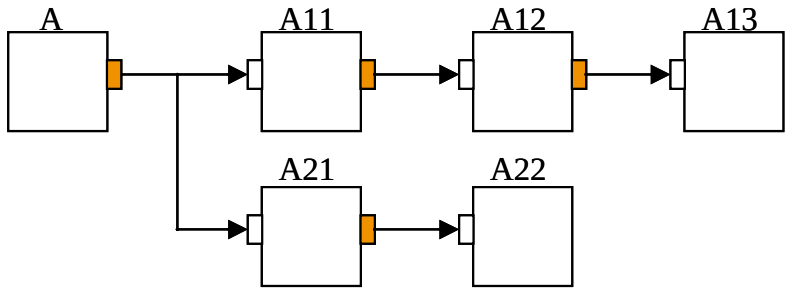



When the sc_outmaster writes to its sc_outmaster port, all inslave and inoutslave processes will be invoked. In this case, sc_inslave1, sc_inslave2 and sc_inoutslave will be invoked. Similarly a read operation from an sc_inmaster port will invoke all sc_outslave and sc_inoutslave processes. In the example illustrated in Figure 7, sc_outlsave1, sc_outslave2 and sc_inoutslave will be invoked. However, only one slave should write a value to its output port.

The example in Figure 8 shows RPC chains in a multi-point link. Autonomous process A connects through the same master port to slave processes A11 and A21 in a multi-point connection. A11 connects to slave A12, which connects to A13. Similarly A21 connects to slave A23. A, A11, A12, A13 form an RPC-chain. Similarly, A, A21, A23 form an RPC-chain. When A accesses its master port, which chain will be executed first is undefined. A chain that is invoked must return before the

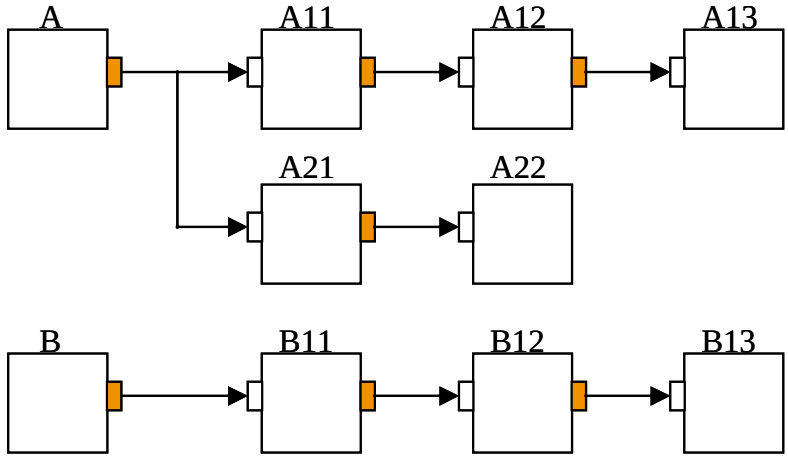
next one will be invoked. In the example, a valid execution order could be: A, A11, A12, A13, A21, A23. The RPC-chains execute in the same delta cycle as A.

**FIGURE 8. Execution Order in a Multi-Point Link**




Execution order between autonomous processes in a given delta cycle is undefined. In the example of Figure 9, A and B are two autonomous processes each with their own RPC chains. A valid execution order is: A, A11, A12, A13, A21, A22, B, B11, B12, and B13.

**FIGURE 9. Execution Order With Multiple Masters**



## Simulation Semantics in UTF

The simulation semantics in UTF are based on the concept of delta cycles (see Chapter 9, “Simulation and Debugging Using SystemC” starting on page 207) and are defined as follows:

- All autonomous processes execute concurrently in delta cycles such that simulation time does not advance. Autonomous processes synchronize over signals, which follow the evaluate-update semantics (see Chapter 9, “Simulation and Debugging Using SystemC” starting on page 207).
- A slave process executes in the same delta cycle as the process that invokes it.
- A process that activates one or more slaves blocks until all slaves return. The slaves are invoked sequentially but in an undefined order.
- The new value assigned to an abstract link is updated immediately[1].

The consequences are the following:

- All processes in an RPC-chain execute in the same delta cycle as the initiating autonomous process.
- A slave process can be invoked multiple times in the same delta cycle from the same or different autonomous processes connected to the same link.
- The new value on a link is visible in the same delta cycle to all connected processes. This contrasts with the evaluate-update semantics of signals.
- All slave processes invoked by an autonomous process execute before the next autonomous process in the current delta cycle.

## Abstract Port Classes

Abstract ports are specialized port classes, which aggregate the following characteristics into a port:

- Define control flow as master or slave type
- Data flow direction: input, output or input-output
- Data type

1. This can give rise to in-deterministic behavior if not used carefully.

Example:

```
sc_inmaster<T>    port_name;
```

This is a master port with data type <T> and it is of input type.

### *Abstract Protocol Master Port Classes*

Input, output, and input-output master ports of type T are declared as sc_inmaster<T>, sc_outmaster<T>, and sc_inoutmaster<T>.

The syntax to declare master ports is shown in the following example:

```
sc_inmaster<T>          inp1;

sc_outmaster<T>         out1;

sc_inoutmaster<T>       inout1;
```

where T can be any of the C++ built-in types (long, int, char, short, float, double, and so forth) or any of the SystemC types (sc_int<N>, sc_uint<N>, sc_bigint<N>, sc_biguint<N>, sc_bit, sc_logic, or fixed point types), or a user-defined data type.

### *Abstract Protocol Slave Port Classes*

Input, output, and inout slave ports of type T are declared as sc_inslave<T>, sc_outslave<T>, and sc_inoutslave<T>.

The syntax to declare slave ports is shown in the following example:

```
sc_inslave<T>           inps1;

sc_outslave<T>          outs1;

sc_inoutslave<T>        inouts1;
```

where T can be any of the C++ built-in types (long, int, char, short, float, double, and so forth) or any of the SystemC types (sc_int<N>, sc_uint<N>, sc_bigint<N>, sc_biguint<N>, sc_bit, sc_logic, or fixed point types), or a user-defined data type.

### *Master and Slave Without Direction*

Two specialized port classes, sc_master and sc_slave, only convey control flow without data flow direction or data type. They are used for control signals, for example an interrupt signal.

To define master or slave ports without direction, use the following syntax:

```
sc_master<>        mport;

sc_slave<>         sport;
```

In the calling process, a transaction is invoked by calling the function method on the master port, which will invoke the slave process connected to the master port. For the above example this would be:

```
mport();
```

It is illegal to write to or read from ‘directionless’ ports. Data type can be specified optionally but will have no effect.

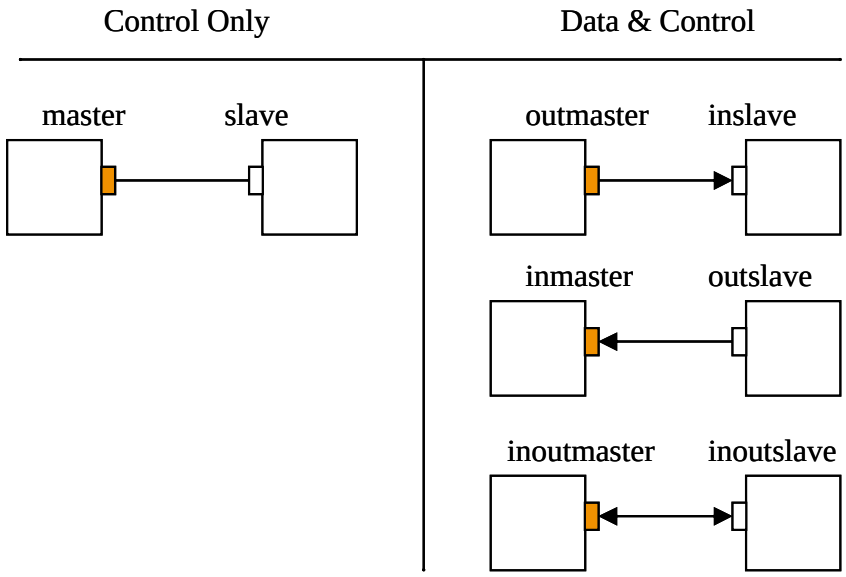
### *Point-to-Point (p2p) Communication*

Abstract ports can be used in the following combinations in a p2p link:

| | | |
|---|---|---|
| sc_master | --- link --- | sc_slave |
| sc_outmaster | --- link --- | sc_inslave |
| sc_inmaster | --- link --- | sc_outslave |
| sc_inoutmaster | --- link --- | sc_inslave |
| sc_inoutmaster | --- link --- | sc_outslave |
| sc_outmaster | --- link --- | sc_inoutslave |
| sc_inmaster | --- link --- | sc_inoutslave |
| sc_inoutmaster | --- link --- | sc_inoutslave |

A master and a slave port must always be used as a pair at both ends of a p2p communication link. You need to properly match an output port with an input port. A pair of two masters, two slaves, two outputs, or two inputs at the opposite ends of a p2p link is illegal. Figure 10 illustrates some legal p2p connections.

**FIGURE 10. Some Legal p2p Connections**




## *Port Arrays*

Arrays of ports are defined using the C array syntax. For example, the following is an array of sc_inmaster ports of type int with 10 elements.

```
sc_inmaster<int>  P1[10];
```

## *Indexed Ports*[1]

Master and slave processes can perform data transactions that have an address, also called an index. Using this mechanism, a master process can write to or read from an address in a memory block in a slave process. Indexed ports are declared with an integer-valued range parameter that specifies the upper limit of the address range, which always starts from 0. Master and slave ports in a link must have matching address ranges. The master specifies the index value of a transaction as a master port index, while the slave reads this value using the get_address() method of its slave port. You cannot specify the index of a transaction at a slave port. It assumes the index of the master transaction. Access to an index outside the range is illegal and will result in a run-time error.

Syntax:

sc_outmaster<T, sc_indexed<range> > port;

sc_inslave<T, sc_indexed<range> > port;

Ports of type sc_master and sc_slave do not take a range parameter.

For example, you define an sc_outmaster port P2 of type int with an address range of (0,1023) as follows:

```
sc_outmaster<int, sc_indexed<1024> >   P2;
```

The following code illustrates a master write of an int value with address 68 on sc_outmaster P2:

```
SC_MODULE(my_master) {
       sc_outmaster<int, sc_indexed<1024> > P2;
       int data;
       …
       { // body of master process tied to port P2
              …
              data =  10;
              P2[68] = data;
              . . .
       }
```

1. Indexed ports are an abstraction at the functional level. They do not exist at the BCA/CA levels.

```
};
```

In the following code for the slave, assume port pslave of my_slave is connected to port P2 above. The result of this transaction is that the value 10 is written into memory location 68.

```
SC_MODULE(my_slave) {
       sc_inslave<int, sc_indexed<1024> >      pslave;
       int  memory[1024];     // memory block
       …
       { // body of slave process triggered by pslave
              int  index;
              index = pslave.get_address();
              // index gets value 68 of the master transaction
              memory[index] = pslave;
              // memory[68] is assigned  the value 10
              // don’t specify an index with pslave
              . . .
       }
};
```

In the example above, a master process attached to port P2 writes out a value with index 68. The slave process reads the index value of the transaction by invoking the member function get_address() of the pslave port.

### *Indexed Port Arrays*

You can define an array of indexed ports. An indexed port array P2 with 10 members and an address range of (0,1023) for each member is specified in the following example:

```
sc_inmaster<int, 1024>  P2[10];
```

To read the value of port member 5 at address 68, you use the C++ syntax for two-dimensional arrays, as shown in the following example. The first index indicates the member of the port array, and the second index indicates the address of the data.

```
int  data;
data = P2[5] [68];
```

### *Inout Ports*

When a slave process is triggered by an sc_inoutslave port in response to a transaction initiated by a master process, the slave process needs to determine the direction of the data transfer requested by the master. This is done by calling the input() method of the slave port. This method returns true for an input transaction into the slave port and false for an output. The master always determines the direction of a transaction by writing to or reading from its master port. A slave response that is incompatible with the master request will result in a run-time error.

In the following example, the direction of the data transfer on port P1 is obtained by the P1.input() method.

```
sc_inoutslave<int>  P1;
 // the rest of the code is not shown
if  ( P1.input() )
             { val = P1;  // read from P1
             }
else
             { P1 = val;  // write to P1
             }
```

### *Abstract Port Read/Write Operations*

The following abstract port read/write operations are supported:

- A master process writes to an sc_outmaster port and reads from an sc_inmaster port.
- A port access may or may not involve a data transfer, depending on the port type.
- Accesses to sc_master and sc_slave ports do not involve a data transfer.
- A write to an sc_outmaster port or a read from an sc_inmaster port will trigger the slave processes connected to the port.
- A write access to an sc_inmaster port is illegal and will result in a run-time error. A read from an sc_outmaster port is illegal. A master process can read from or write to an sc_inoutmaster port.
- A slave reads from a sc_inslave port and writes to a sc_outslave port. A write to a sc_inslave port or a read from a sc_outslave is illegal and will cause a run-time error. Legal operations on an sc_inoutslave port are determined by the master. This is discussed in “Inout Ports” on page 122.

- Only one process in a module should write to an output or an inout port (master or slave). Otherwise, the result will be unpredictable. The SystemC class library does not perform a check on this condition. An input port (master or slave) can be bound to multiple processes in a module. An sc_inslave port can be bound to multiple slave processes, but not an sc_outslave or sc_inoutslave port.

## *Abstract Ports and Module Hierarchy*

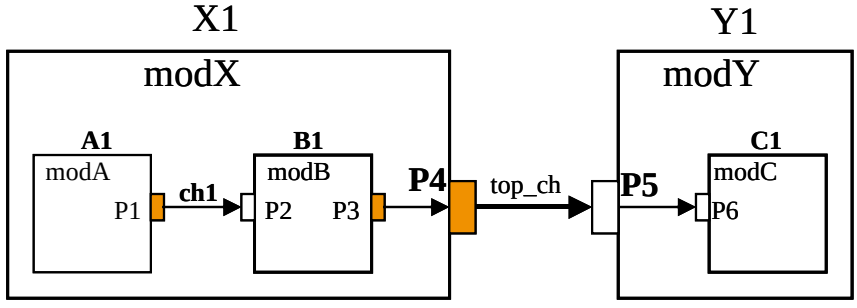
When modules are instantiated inside other modules, a child module port that connects to a parent module port must be of compatible type as follows:

- A master port must connect to a master port, a slave port to a slave port.
- A child input port can connect to a parent input or inout port.
- A child output port can connect to a parent output or inout port.
- A child inout port can connect to a parent inout port.
- Data type of child and parent module must be the same.

These rules are checked at initialization by the SystemC class library.

Figure 11 illustrates how abstract ports and links are used in a hierarchical system. In this system, P4 is a representation of P3 at another level of hierarchy, so it retains the same abstract protocol specification as P3. In module modY, P5 has the same relationship to P6. The code is also shown below.

**FIGURE 11. Hierarchical System Example**



```
SC_MODULE(modA) {
    sc_outmaster<int> P1;
    . . .
};

SC_MODULE(modB) {
    sc_inslave<int> P2;
    sc_outmaster<int> P3;
    . . .
};

SC_MODULE(modC) {
    sc_inslave<int> P6;
    . . .
};
SC_MODULE(modX) {
    sc_outmaster<int> P4;
    sc_link_mp<int> ch1;
    modA*A1;
    modB   *B1;
    . . .
    SC_CTOR(modX) {
        A1 = new modA(“A1”);
        A1(ch1);
        B1 = new modB(“B1”);
        B1(ch1, P4);
    }
};

SC_MODULE(modY) {
    sc_inslave<int> P5;
    modC  *C1;
    …
    SC_CTOR(modY) {
        C1 = new modC(“C1”);
        C1(P5);
    }
};

SC_MODULE(top) {
    sc_link_mp<int>  top_ch;
    modX*X1;
    modY  *Y1;
    . . .
    SC_CTOR(top) {
```

```
        X1 = new modX("X1);
        X1(top_ch);
        Y1 = new modY("Y1");
        Y1(top_ch);
    }
};
```

## BCA & CA Abstraction Levels

When going from the functional abstraction to the BCA or CA abstraction levels, you must make the following transformations:

- Abstract ports are replaced by bus ports, which have bus communication protocols.
- Autonomous & slave processes are transformed into synchronous processes with explicit clock and reset signals.
- Bus communication protocols are implemented in the body of the processes.

These transformations are discussed in the remainder of this section.

### Bus Port Classes

Bus ports refine abstract ports as an aggregation of an abstract port type, a set of bus terminals, and a bus protocol specification. Bus ports are useful in bus architectures.

A bus port can have a combination of the following terminal types, depending on the bus protocol:

- data
- address
- control

Bus terminals are defined as data members of a bus protocol class (see "Read/Write Operations on Bus Ports" on page 128).

Bus ports do not implement protocol semantics, they only carry the terminals of a protocol. The semantics must be implemented in the body of the process that is

bound to the port. There is no check to ensure that the port protocol specification matches the implementation in the process.

Syntax for declaring a bus port with the sc_enableHandshake protocol is given below.

```
sc_outmaster<T, sc_enableHandshake<T> >  myPort;
```

This defines a bus port with data type <T> and an enable handshake protocol. Abstract port type is sc_outmaster. The data type <T> is specified twice, once for the abstract port, once for the bus protocol. Both data types must be identical. The sc_enableHandshake protocol has a ‘data’ terminal and an ‘enable’ control terminal.

The following protocols are provided as a part of the SystemC class library:

- sc_noHandshake<T>
- sc_enableHandshake<T>
- sc_fullHandshake<T>

Figure 12 shows the terminals for built-in protocols.

**FIGURE 12. Terminals of Built-In Bus Protocols**

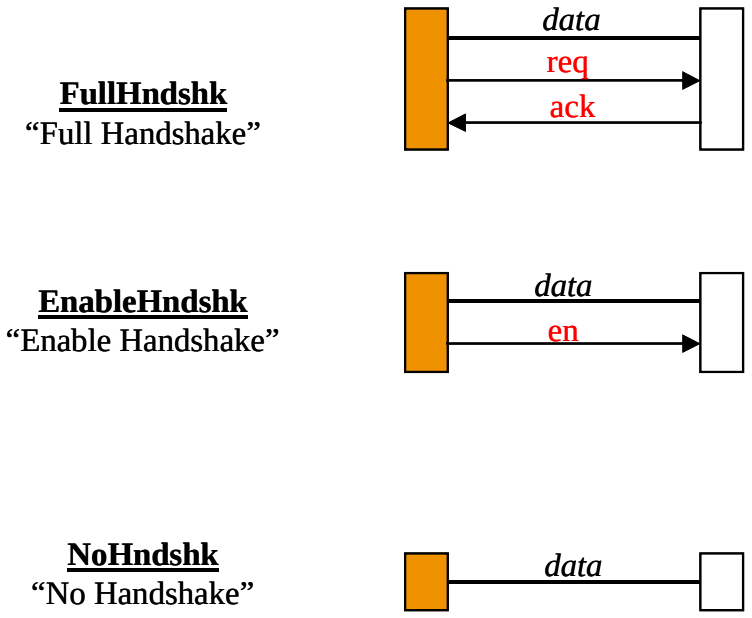
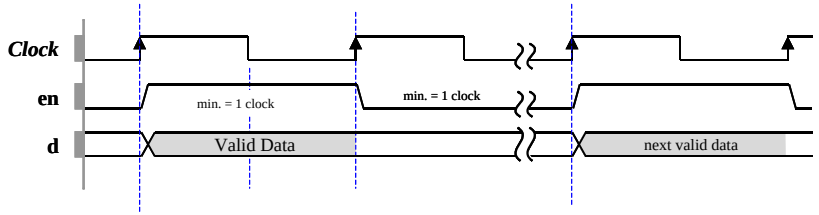



Figure 13 shows the timing diagram for the sc_enableHandshake protocol.

**FIGURE 13. Enable-Handshake Timing Diagram**



### *Read/Write Operations on Bus Ports*

You can read from and write to the terminals of a bus protocol. However, you must use the indirection '->' operator to access the terminals of a bus port. You cannot use the member '.' operator because a bus terminal is not a data member of a port. Instead, it is a data member of the bus protocol associated with the port.

This example illustrates the use of bus ports.

```
sc_inoutslave<int, sc_enableHandshake<int>  >  myPort;
value = myPort->data;  // read from data terminal of port
myPort->data = value;  // write to data terminal of port
```

Other terminal accesses on the same port are shown below:

```
if (myPort->enable) {…}
          // checks the value of the enable terminal
if (myPort->input) {…}
          // checks the direction of the
          // requested transaction
```

The following example shows an sc_enableHandshake protocol with an sc_inoutslave port. On each invocation, the *enable* terminal is checked. If *enable* is true then the process proceeds; otherwise, it goes to sleep waiting for the next invocation. To determine the direction on the inoutslave port, the *input* terminal is checked and a matching (read/write) access to the port is performed.

```
SC_MODULE(myMod) {
   sc_inoutslave<int, sc_enableHandshake<int> >  myPort;
   sc_in_clkclk;
   sc_in<bool>reset;
   int value;

   void  myfunc() {
      … // do process initialization here
      while(1) {
           wait();// wait for clock
      if reset { … // do reset
}
       else {
          if (myPort->enable) {
              if myPort->input {
                 // determine inout direction
                 value = myPort->data;// read from port
```

```
            }
            else {
               myPort->data = value;// write to port
            } // if input
          } // if enable
        } // else
       } // while
     } // myfunc

   SC_CTOR(myMod) {
      SC_THREAD(myfunc);
      sensitive_pos << clk;
   }
};
```

The following example illustrates a sc_enableHandshake protocol communication between a master and a slave process at the CA level. The master sets the control terminals; the slave reads them and responds accordingly.

```
// First module
SC_MODULE(modA) {

   sc_inoutslave<int, sc_enableHandshake<int> > P1;
   sc_in_clk clk;
   int    value;

   void A_ftion() {
      if (P1->enable) {
            if (P1->input){        // read
                value = P1->data;
                 …
            }
            else { // write
                P1->data = value;
                …
            }
         }
      }

   // Constructor
   SC_CTOR(modA) {
      SC_METHOD(A_ftion);
      sensitive_pos << clk;
   }
};
```

```
// Master module
SC_MODULE(modB) {

   sc_outmaster<int, sc_enableHandshake<int> > P2;
   sc_in_clk  clk;
   int       value;

   void B_ftion() {
      …
      wait();
      P2->enable = 1;
      P2->input = 1;          // write to slave
      P2->data = value;
      …
      wait();
      P2->input = 0;          // change direction to read
      …
      wait();
      value = P2->data;       // read from slave
            …
   }

   SC_CTOR(modB) {
      SC_THREAD(B_ftion);
      sensitive_pos << clk;
   }
};

SC_MODULE(top) {
   sc_link_mp<int>  C1;
   sc_clock clk("clk", 20);
   modA *A1;
   modB *B1;

   // Constructor
   SC_CTOR(top) {
      A1 = new modA("A1");
      B1 = new modB("B1");

      A1.P1(C1);
      A1.clk(clk);
      B1(C1, clk);     // bind by order

   }
```

```
};
```

### *Multi-Point Links*

Bus ports can be connected to other bus ports using a multi-point link. Connecting bus ports with incompatible protocols or data types constitutes an error. Mixing non-bus (abstract and primitive ports) and bus ports in the same link is illegal. The syntax for a link declaration is as follows:

```
sc_link_mp<T>  link1;
```

The link does not have a protocol specifier since the link is a mere connection and does not implement the protocol as explained before. The link must have a data type that is the same as the ports that are connected to it.

You cannot bind terminals of a bus port, you must bind the bus port[1] as a whole.

The rules for binding ports to links or other ports are the same as for primitive ports.

## *Supported Mixed Levels of Abstraction*

Mixing different levels of abstraction allows for a gradual refinement of a system. Interfaces between functional and BCA/CA levels are based on the concept of a BCA shell module, called BCASH. BCASH provides an untimed shell around a BCA/CA module. In the BCASH module, protocol conversion interface modules are provided to convert between untimed and timed communication protocols. Protocol conversion is based on the following:

- Protocol conversion interface modules have ports at different abstraction levels.
- A functional level slave process can have primitive (sc_in, sc_out or sc_inout) or bus type ports for read and write access.
- A functional process can be sensitive to signals, primitive ports, or to terminals of a bus-port.

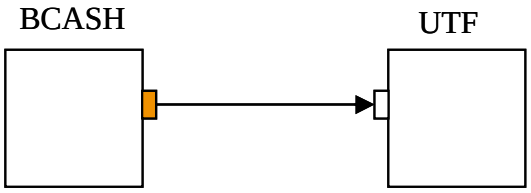
1. This restriction will be removed in the future.

- A cycle accurate process can have abstract master but not abstract slave ports. The latter requirement is because a cycle accurate process cannot be invoked directly as a function call but instead must be triggered through signals as one would expect.

### *Communication From BCASH Master To UTF Slave*

In Figure 14, a BCASH module with a master port is linked to a slave process. This is straightforward since a BCASH process can invoke a slave function through its master port.

**FIGURE 14. BCASH Master to UTF Slave Communication**

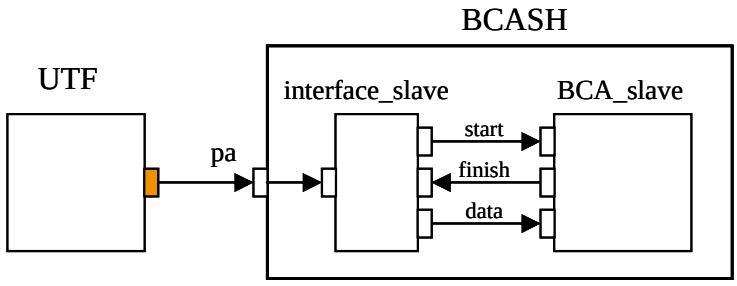



### *Communication From UTF Master To BCA Slave*

In Figure 15, a BCASH module is linked through slave port *pa* to a functional module, which is the master. Since a BCA process cannot have a slave port, it interfaces to the module at the functional level through an interface_slave process in the BCASH module. The role of the interface module is as follows: when the master invokes the slave interface through port *pa*, the latter triggers the BCA_slave pro-

cess using the *start* signal. The BCA_slave executes and signals back to the interface_slave through port *finished*.

**FIGURE 15. UTF Master to BCA Slave Communication**




The BCASH module is shown below.

```
SC_MODULE (BCASH) {
    sc_signal<bool>         start;    // internal signal
    sc_signal<bool>         finished; // internal signal
    sc_in_clk               clk;
    sc_in                   reset;
    sc_inslave<int>         pa;    // abstract slave port
    sc_signal<sc_uint<16> >data;  // internal data link to BCA
                                   // level

    void interface_slave() {// blocking RPC
        sc_uint<16> temp;// temperorary for type conversion
        start = 1;       // set 'start'
        temp=pa;         // assign to temp for type conversion
        data = temp;
        wait();          // wait for 'finished' 0->1 event
        start = 0;
    }

    void BCA_slave() {            // cycle accurate process
        if reset { finished = 0; } // synchronous reset
        else
        if start {

            int val;              // temporary storage
            finished = 0;         // reset
```

```
            val = data;             // read data signal
            …                       // process
            finished = 1;           // signal back
        }
    }

    SC_CTOR(BCASH) {
        SC_SLAVE(interface_slave, pa);
        sensitive_pos << finished;

        SC_METHOD(BCA_slave);
        sensitive_pos << clk ;
    }
};
```

## *Examples*

This section describes the following examples:

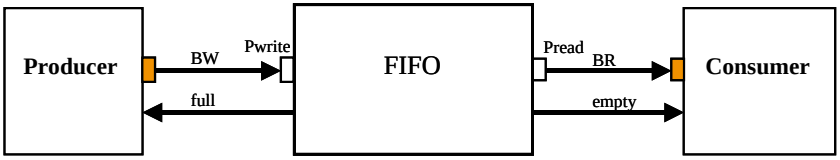
- FIFO model at the functional level
- FIFO model at the BCA level
- Simple arithmetic processor at the functional level
- Simple arithmetic processor at the BCA level

### *FIFO Model at the Functional Level*

This example illustrates how blocking RPC is used to model blocking write and read in a FIFO communication link. A producer process produces data items (integers in this example) and sends it through the FIFO link to the consumer. The producer runs at a higher speed than the consumer process such that the FIFO buffer will get full after some time. This will block the producer process since it will not be able to write into the FIFO buffer. As soon as the consumer removes an item

from the FIFO buffer, the producer will be unblocked and write another item into the buffer.

**FIGURE 16. FIFO Model at the Functional Level**




In the example code below, producer and consumer are implemented as clocked SC_THREAD processes to allow the two processes to run at different ‘speeds’. Producer is sensitive to the clock and to the ‘full’ signal coming from the FIFO. Producer has a sc_outmaster port through which it invokes the blockingWrite() slave process in the FIFO. The blockingWrite() blocks when ‘full’ signal is high and is woken up when ‘full’ goes low. Consumer is sensitive to a different clock and to the ‘empty’ signal. Consumer has an sc_inmaster port through which it invokes the blockingRead() slave process in the FIFO. The blockingRead() blocks when ‘empty’ signal is high and is woken up when ‘empty’ goes low. Both slave processes are of blocking-slave type and are woken up by the sensitivity signals of their respective calling processes[1]. For the sake of brevity, the buffer class is not shown in the code below. Simulation results following the code show a correct operation of the FIFO.

```
// file buffer.h
#define BUFFER_SIZE 10

template <class itemT>
class buffer
{
public:
  buffer(); // constructor
  virtual ~buffer() {}
  itemT get();          // get item from the top of buffer
  bool put(itemT);      // put item at the bottom of buffer
  bool full();          // buffer full
  bool empty();         // buffer empty
```

1. This is due to the limitation in SystemC v1.1, which requires slaves to share the sensitivity list of their respective autonomous processes.

```
private:
    // private section not shown
};

// File Name : fifo.h
// A module with inslave and outslave abstract ports,
// and a fifo store

#ifndef SYSTEMC_H
#include "systemc.h"
#endif

#ifndef BUFFER_H
#include "buffer.h"
#endif

SC_MODULE(fifo)
{

       // ports
       sc_out<int> full;
       sc_out<int> empty;
       sc_inslave<int> Pwrite; // slave port
       sc_outslave<int> Pread; // slave port

       buffer<int> buf;       // FIFO buffer
       int item;              // buffer item

       // Slave methods

       void blockingWrite()
       {
         if (buf.full() )
         {// wait for not full signal from blockingRead;
          // ignore clock events
          do {wait();}
          while( buf.full());
         }

       // buffer is not full
       // write into buffer
       if (buf.empty()) // check before put()
       empty=0;         // generate not empty event
       item = Pwrite;   // read from slave port
       cout << "Writing into buffer: item = " << item << endl;
```

```
        buf.put(item);
        if (buf.full()) full=1; // generate full event

        }

        void blockingRead()
        {
          cout << "\nfifo:blockingRead called" << endl;
          if (buf.empty())
          {   // wait for not empty signal from blockingWrite
              // ignore clock events
              do {wait();}
               while (buf.empty() );
          }

           // buffer is not empty
           // read from buffer
           if (buf.full())     // check before get()
               full=0;         // generate not full event
           item = buf.get();
           cout << "  Item read = " << item << endl;
            Pread = item; // write to slave port
            if (buf.empty()) empty=1; // generate empty event

        }


        SC_CTOR(fifo)
         {
           SC_SLAVE( blockingWrite, Pwrite);
           SC_SLAVE( blockingRead, Pread);
         }

};


//
// producer.h  ::  producer with outmaster
//

#ifndef SYSTEMC_H
#include "systemc.h"
#endif
```

```
SC_MODULE(producer)
{

        // port declaration
        sc_in<int> full;
        sc_outmaster<int> Pout; // refinable port
        sc_in_clk clk;

       // Internal variable
        int val;

       // outmaster process
        void producer_thread()
        {
            while (true)
            {
              val += 2;
              Pout = val;

             // wait for pos_edge clock event
             // ignore full events
             do { wait(); }
             while (!clk.event() || (!clk));
            }
        }

        SC_CTOR(producer)
        {
           SC_THREAD(producer_thread);
           sensitive << full << clk;
          val=0;
        }

};

//
// consumer.h  ::  consumer module with inmaster
//


#ifndef SYSTEMC_H
#include "systemc.h"
#endif

SC_MODULE(consumer)
```

```
{
        // declare ports
        sc_in<int>   empty;
        sc_inmaster<int> Cin;
        sc_in_clk            clk;
        // Internal variable
        int x;


        // inmaster process
        void consumerFunc()
        {
          while (true)
          {
                  x = Cin;
                 // wait for pos_edge clk
                 // ignore empty events
                 do { wait();}
                 while (!clk.event() || (!clk));
          }
        }

       SC_CTOR(consumer)
        {
          SC_THREAD(consumerFunc);
          sensitive << empty << clk;

        }

};

//
// top.cc : contains sc_main ; Instantiates FIFO design
//

#ifndef SYSTEMC_H
#include "systemc.h"
#endif

#include "consumer.h"
#include "producer.h"
#include "fifo.h"

#ifndef BUFFER_H
#include "buffer.h"
```

```
#endif


int sc_main(int ac, char *av[] )
{

  // declare channels/signals
  sc_signal<int> full;
  sc_signal<int> empty;

  sc_channel_mp<int> BW;
  sc_channel_mp<int> BR;

  // create clocks with diff frequencies
       sc_clock clock1 ("Clock1", 5, 0.5, 0.3,true);
       sc_clock clock2 ("Clock2", 40, 0.5, 0.1,true);

  // instantiate all blocks and connect to channels, signals

       producer        p1("Master");
       fifo            f1("fifo");
       consumer        c1("Slave");

       p1.Pout(BW);
       p1.full(full);
       p1.clk(clock1);
       f1.Pwrite(BW);
       f1.full(full);
       f1.empty(empty);
       f1.Pread(BR);
       c1.Cin(BR);
       c1.empty(empty);
       c1.clk(clock2);

       sc_start(100);
    // return zero if no error
       return 0;
}
```

### Simulation Results

Output results from simulation of the FIFO example are shown below. Note that the FIFO buffer gets filled up by blockingWrite's starting from the left, and gets emp-

tied by blockingRead from the right. When the buffer is full, blockingWrite is blocked and unblocks when an item is removed from the buffer.

```
fifo:blockingRead called
Writing into buffer: item = 2
  Array after shifting = 2 0 0 0 0 0 0 0 0 0
Writing into buffer: item = 4
  Array after shifting = 4 2 0 0 0 0 0 0 0 0
Writing into buffer: item = 6
  Array after shifting = 6 4 2 0 0 0 0 0 0 0
Writing into buffer: item = 8
  Array after shifting = 8 6 4 2 0 0 0 0 0 0
  Item read = 2
Writing into buffer: item = 10
  Array after shifting = 10 8 6 4 0 0 0 0 0 0
Writing into buffer: item = 12
  Array after shifting = 12 10 8 6 4 0 0 0 0 0
Writing into buffer: item = 14
  Array after shifting = 14 12 10 8 6 4 0 0 0 0
Writing into buffer: item = 16
  Array after shifting = 16 14 12 10 8 6 4 0 0 0

fifo:blockingRead called
  Item read = 4
Writing into buffer: item = 18
  Array after shifting = 18 16 14 12 10 8 6 0 0 0
Writing into buffer: item = 20
  Array after shifting = 20 18 16 14 12 10 8 6 0 0
Writing into buffer: item = 22
  Array after shifting = 22 20 18 16 14 12 10 8 6 0
Writing into buffer: item = 24
  Array after shifting = 24 22 20 18 16 14 12 10 8 6

fifo:blockingRead called
  Item read = 6
Writing into buffer: item = 26
  Array after shifting = 26 24 22 20 18 16 14 12 10 8
```

## *FIFO Model at the BCA Level*

The same FIFO at the BCA level is shown below. Full-handshake protocol is used between the producer/consumer threads and the FIFO. Enable-handshake protocol would not work here since the FIFO is blocking. Note how the full-handshake pro-

tocol is implemented inside the bodies of the processes. For the sake of brevity, the consumer process is not shown, which is very analogous to the producer.

```
//
// producer_bp.h  ::  producer with fullHandshake protocol
//

#include "systemc.h"

SC_MODULE(producer)
{
  // Internal variable
  int val;
  int state;  // state variable of FSM
  // port declaration
  sc_outmaster<int, sc_fullHandshake<int> > Pout; // bus port
  sc_in_clk clk;

  // outmaster process
  void producer_thread()    {
      state=0;
      Pout->req=0;
      while (true) {
     wait();
     switch(state) {
     case 0:
       Pout->req=0;
       if (Pout->ack==0) {
         state=1;
       }
       break;
     case 1:
       Pout->req=1;
       val += 2;
       Pout->d = val;
       state=2;
       break;
     case 2:
       if (Pout->ack==1){
         state=0;
         Pout->req=0;
       }
       break;
     }
   }
```

```
    }

  SC_CTOR(producer)
    {
      SC_THREAD(producer_thread);
      sensitive_pos << clk;
      val=0;
    }
};



// File Name : fifo_bp.h
// A module with inslave and outslave fullHandshake bus ports,
// and a fifo store

#include "systemc.h"

#ifndef BUFFER_H
#include "buffer.h"
#endif

SC_MODULE(fifo)
{
  // ports
  sc_in_clk wclk;
  sc_in_clk rclk;
  sc_inslave<int, sc_fullHandshake<int> > Pwrite;
                        // slave port
  sc_outslave<int, sc_fullHandshake<int> > Pread;
                        // slave port

  buffer<int> buf;
  int item;

  // Slave methods
  void blockingWrite()
    {
      Pwrite->ack=0;
      while(1) {
        wait();  // wait for clock
        if (Pwrite->req) {
     if (buf.full() ) {
     // wait for not full signal from blockingRead;
     do {
```

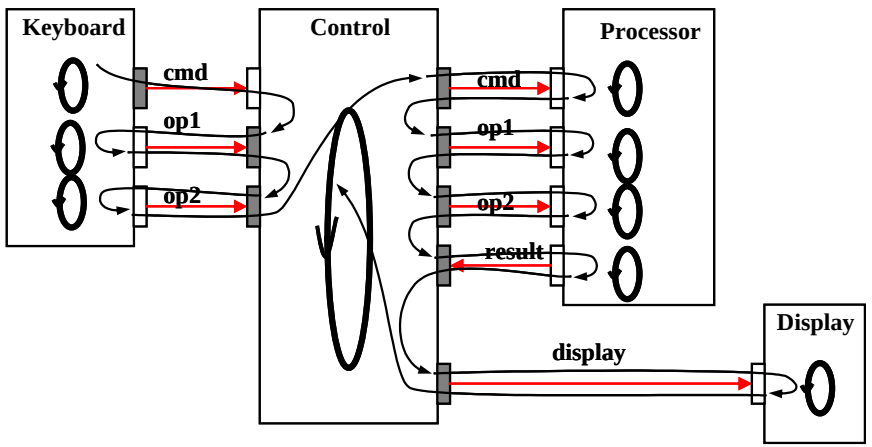
```
        wait();
      }  // wait for clock
    while( buf.full());
  }

  item = Pwrite->d; // read from slave port
  cout << "Writing into buffer: item = " << item << endl;

  buf.put(item);
  Pwrite->ack=1;
  while(Pwrite->req==1)
    wait();
  Pwrite->ack=0;
      }
    }
  }

void blockingRead()
  {
    Pread->ack=0;
    while(1) {
      wait(); // wait for clock
      if (Pread->req) {
  cout << "\nfifo:blockingRead called" << endl;
  if (buf.empty())
    {  // wait for not empty signal from blockingWrite
      do {wait();} // wait for clock
      while (buf.empty() );
          }

  item = buf.get();
  // cout << "  Item read = " << item << endl;
  Pread->d = item; // write to slave port
  Pread->ack=1;
  while(Pread->req==1)
    wait();
  Pread->ack=0;
      }
    }
  }

SC_CTOR(fifo)
  {
    SC_THREAD(blockingWrite);
    sensitive_pos << wclk;
```

```
      SC_THREAD(blockingRead);
      sensitive_pos << rclk;
    }
};
```

## *Simple Arithmetic Processor at the Functional Level*

In this example, the key module (keyboard) serves as a test bench to generate inputs for a simple processor that takes a command for an arithmetic operation together with two operands and calculates the results. The control module (ctrl) dispatches the commands coming from the keyboard to the processor, then gets the results from the processor and sends it to the display module for display. There is a single autonomous thread (key_input) in the key module. All other processes are slaves executing in this single autonomous thread. The execution thread is shown in Figure 17.

**FIGURE 17. Execution Thread of a Simple Arithmetic Processor**




```
#include "systemc.h"

# define ADD 1
# define SUBTRACT 2
# define MULTIPLY 3
# define MYMAX 5
```

```
SC_MODULE(key)
{

  // ports
  sc_outmaster<int > cmd;
  sc_outslave<int > op1;
  sc_outslave<int > op2;

  static int array1[MYMAX];
  static int array2[MYMAX];
  int i, state;
  int scratch;

  void key_input()
  { // implements autonomous thread
     // runs forever
   while (true)
   {
     switch (state)
      {
      case 0:
        scratch = ADD;
        cmd = scratch;
        if (i == MYMAX-1)
          state = 1;
        break;
      case 1:
        cmd = SUBTRACT;
        if (i == MYMAX-1)
          state = 2;
        break;
      case 2:
        cmd = MULTIPLY;
        if (i == MYMAX-1)
          state = 0;
        break;
      }
      if (i == MYMAX-1)
        i = 0;
      else i++;

   }

  }
```

```
  void slave_to_op1()
    {
      op1 = array1[i];
    }

  void slave_to_op2 ()
    {
      op2 = array2[i];
    }

  // constructor
  SC_CTOR(key)
    {
      SC_THREAD(key_input); // autonomous thread
      SC_SLAVE(slave_to_op1, op1);
      SC_SLAVE(slave_to_op2, op2);
      state = 0;
      i = 0;
    }
};

#include "systemc.h"

SC_MODULE(ctrl)
{
  // ports
  sc_inslave<int > cmd_in;
  sc_inmaster<int > op1_in;
  sc_inmaster<int > op2_in;

  sc_outmaster<int > op1_out;
  sc_outmaster<int > op2_out;
  sc_outmaster<int > cmd_out;
  sc_inmaster<int > result;

  sc_outmaster<int > display;

  // internal variables
  int operand1;
  int operand2;
  int cmd_2;


  // slave "methods"
  void slave_to_cmd_in ()
```

```
    {
      operand1 = op1_in;
      operand2 = op2_in;

      op1_out = operand1;
      op2_out = operand2;
      cmd_out = cmd_in;
      display = result;

      cmd_2 = cmd_in;

      cout << "\n operand1 = " << operand1 << "  operand2 = "
<< operand2
<< "  cmd = " << cmd_in;

    }

    SC_CTOR(ctrl)
    {
      SC_SLAVE(slave_to_cmd_in, cmd_in);
    }

};

#include "systemc.h"

SC_MODULE(fun)
{
  // ports
  sc_inslave<int > op1;
  sc_inslave<int > op2;
  sc_inslave<int > cmd;
  sc_outslave<int > result;
  // internal variables
  int operand1;
  int operand2;
  int command;


  void slave_to_op1 ()
    {
      operand1 = op1;           // read operand 1
    }
```

```
  void slave_to_op2 ()
    {
      operand2 = op2;           // read operand 2
    }


  void slave_to_cmd ()
    {
      command = cmd;            // read command
    }


  void slave_to_result ()
    {
      switch (command)
        {
        case 1 :
          result = operand1 + operand2; // Add
          break;
        case 2 :
          result = operand1 - operand2; // Subtract
          break;
        case 3 :
          result = operand1 * operand2; // Multiply
          break;
        default :
          printf ("\nfun: Undefined operation\n");
          break;
        }
    }

  // constructor
  SC_CTOR(fun)
    {
      SC_SLAVE(fun,slave_to_op1, op1);
      SC_SLAVE(slave_to_op2, op2);
      SC_SLAVE(slave_to_result, result);
      SC_SLAVE(slave_to_cmd, cmd);
    }
};

#include "systemc.h"

SC_MODULE(disp)
{
```

```
  // ports
  sc_inslave<int > din;

  // internal variables
  int first_time;

  // slave "method"
  void slave_to_din ()
    {
         if (first_time) first_time = 0;
    else cout << "     Result = " << din << endl;
    }

  // constructor
  SC_CTOR(disp)
    {
      SC_SLAVE(slave_to_din, din);
      first_time = 1;
     }
};

#include "systemc.h"
#include "key.h"
#include "ctrl.h"
#include "fun.h"
#include "disp.h"

int key::array1[MYMAX] = { 1,2,3,4,5 };
int key::array2[MYMAX] = {1,3,5,7,9 };
int sc_main(int argc, char* argv[])
{
  sc_link_mp<int > cmd;
  sc_link_mp<int > in1;
  sc_link_mp<int > in2;
  sc_link_mp<int > op1out;
  sc_link_mp<int > op2out;
  sc_link_mp<int > cmd_out;
  sc_link_mp<int > result;
  sc_link_mp<int > display;


  key k1("key");
  k1.cmd(cmd);
  k1.op1(in1);
  k1.op2(in2);
```

```
  k1.clk(clock);

  ctrl c1("ctrl");
  c1.cmd_in(cmd);
  c1.op1_in(in1);
  c1.op2_in(in2);
  c1.op1_out(op1out);
  c1.op2_out(op2out);
  c1.cmd_out(cmd_out);
  c1.result(result);
  c1.display(display);


  fun f1("fun");
  f1.op1(op1out);
  f1.op2(op2out);
  f1.cmd(cmd_out);
  f1.result(result);


  disp d1("display");
  d1.din(display);

  sc_start(50);

  return (0);

};
```

### Simulation Results

```
 operand1 = 0  operand2 = 0  cmd = 1     Result = 0

 operand1 = 1  operand2 = 1  cmd = 1     Result = 2

 operand1 = 2  operand2 = 3  cmd = 1     Result = 5

 operand1 = 3  operand2 = 5  cmd = 1     Result = 8

 operand1 = 4  operand2 = 7  cmd = 1     Result = 11

 operand1 = 5  operand2 = 9  cmd = 2     Result = -4

 operand1 = 1  operand2 = 1  cmd = 2     Result = 0
```

```
 operand1 = 2  operand2 = 3  cmd = 2     Result = -1

 operand1 = 3  operand2 = 5  cmd = 2     Result = -2

 operand1 = 4  operand2 = 7  cmd = 2     Result = -3

 operand1 = 5  operand2 = 9  cmd = 3     Result = 45
```

### *Simple Arithmetic Processor at the BCA Level*

For the sake of brevity, only two modules of this example at the BCA level are shown.

```
#include "systemc.h"

# define ADD 1
# define SUBTRACT 2
# define MULTIPLY 3
# define MYMAX 5

SC_MODULE(key)
{

  // ports
  sc_outmaster<int, sc_fullHandshake<int> > cmd;
  sc_outslave<int, sc_enableHandshake<int> > op1;
  sc_outslave<int, sc_enableHandshake<int> > op2;

  sc_in_clk    clk;
  sc_in<int>   rstp;

  static int array1[MYMAX];
  static int array2[MYMAX];

  int i, state;

  void key_f ()  {
     if (rstp == 0) {
   state = i = 0;
   cmd->d = 1;
   cmd->req = false;
      }
      else {
```

```
  switch (state) {
  case 0:
    cmd->req = false;
    if (cmd->ack == false)
        state = 1;
    else
      state = 0;
    break;
  case 1:
    cmd->req = true;
    cmd->d = ADD;
    state = 2;
    break;
  case 2:
    cmd->req = true;
    cmd->d = SUBTRACT;
    state = 3;
    break;
  case 3:
    cmd->req = true;
    cmd->d = MULTIPLY;
    state = 0;
    break;
  }

  if (op1->en == true)
    {
      if (i == MYMAX-1)
        i = 0;
      else
        i++ ;
      op1->d = array1[i];
    }
  if (op2->en == true)
    {
      op2->d = array2[i];
    }
     }
}

// constructor
SC_CTOR(key)
 {
    SC_METHOD(key_f)
    sensitive << clk;
```

```
    }
};

#include "systemc.h"

# define idle      0
# define transmit  1

SC_MODULE(ctrl)
{
  // ports
  sc_inslave<int, sc_fullHandshake<int> > cmd_in;
  sc_inmaster<int, sc_enableHandshake<int> > op1_in;
  sc_inmaster<int, sc_enableHandshake<int> > op2_in;

  sc_outmaster<int, sc_enableHandshake<int> > op1_out;
  sc_outmaster<int, sc_enableHandshake<int> > op2_out;
  sc_outmaster<int, sc_enableHandshake<int> > cmd_out;
  sc_inmaster<int, sc_enableHandshake<int> > result;

  sc_outmaster<int, sc_enableHandshake<int> > display;

  // sc_inslave<bool> rstp;
  // sc_inslave<bool> clk;
  //
  sc_in_clk clk;
  sc_in<int> rstp;

  static int state;

  void ctrl_f ()    {
      cmd_in->ack=false;
      if (rstp == 0)
   {
     state = idle;
     cmd_in->ack = false;
   }
      else{

     if (cmd_in->req == true)
       {
         //state = transmit;
         cmd_out->en = true;

         op1_in->en = true;
```

```
            op2_in->en = true;
            op1_out->en = true;
            op2_out->en = true;
            result->en = true;
            display->en = true;


            int x = cmd_in->d;
            cmd_out->d = x;
            //  cout << "cmd =   "<< x ;

            x = op1_in->d;
            op1_out->d = x;
            // cout << " op1 = "<< x;

            x = op2_in->d;
            op2_out->d = x;
            // cout << " op2 = "<< x << endl;

            x = result->d;
            display->d = x;
            // cout << "    Result = " << x << endl;
            //cmd_in->ack = true;
          }
        else {
          int x = result->d;
          display->d = x;
          //  cout << "    Result2 = " << x << endl;
        }
      }
    }

    // constructor of the module
    SC_CTOR(ctrl)
      {
        SC_METHOD(ctrl_f);
        sensitive << clk;
      }
};
```

CHAPTER 8

# Fixed Point Types

When designers model at a high level, floating point numbers are useful to model arithmetic operations. Floating point numbers can handle a very large range of values and are easily scaled. 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. To model the behavior of fixed point hardware designers need bit accurate fixed point data types. Fixed point types are also used to develop DSP software.

SystemC contains signed and unsigned fixed point data types that can be used to accurately model hardware. The SystemC fixed point data types are accurate to the bit level and support a number of features that allow a high level of modeling. These features include modeling quantization and overflow behavior at a high level.

There are 4 basic types used to model fixed point types in SystemC. These are:

- sc_fixed
- sc_ufixed
- sc_fix
- sc_ufix

Types sc_fixed and sc_ufixed uses static arguments to specify the functionality of the type while types sc_fix and sc_ufix can use argument types that are nonstatic.

Static arguments must be known at compile time, while 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.

Types sc_fixed and sc_fix specify a signed fixed point data type. Types sc_ufixed and sc_ufix specify an unsigned fixed point data type.

An object of a fixed point type is declared with the following syntax:

- sc_fixed(wl, iwl, q_mode, o_mode, n_bits) x;
- sc_ufixed(wl, iwl, q_mode, o_mode, n_bits) y;
- sc_fix x(list of options);
- sc_ufix y(list of options);

The arguments to sc_fixed and sc_ufixed are used as follows:

wl - Total word length, used for fixed point representation. Equivalent to the total number of bits used in the type.

iwl - Integer word length - specifies the number of bits that are to the left of the binary point (.) in a fixed point number.

q_mode - quantization mode, this parameter determines the behavior of the fixed point type when the result of an operation generates more precision in the least significant bits than is available as specified by the word length and integer word length parameters.

o_mode - overflow mode, this parameter determines the behavior of the fixed point most significant bits when an operation generates more precision in the most significant bits than is available.

n_bits - number of saturated bits, this parameter is only used for overflow mode and specifies how many bits will be saturated if a saturation behavior is specified and an overflow occurs.

x,y - object name, name of the fixed point object being declared.

The designer can configure the fixed point data type to perform a number of different types of operations. The designer will pass different values of the parameters

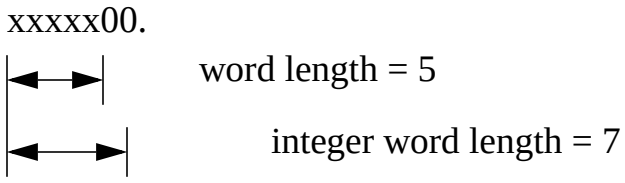
shown above. These parameter values will be used during the construction of the fixed point type to create the desired data type. These types can be the basis for adders, subtractors, multipliers, accumulators, FFTs, etc. All of these devices can be built with bit accurate results

A simple fixed point declaration is shown below:

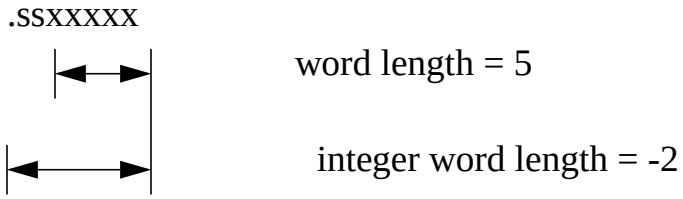
```
sc_fixed<8,4,SC_RND, SC_WRAP,2> val;
```

## *Word Length and Integer Word Length*

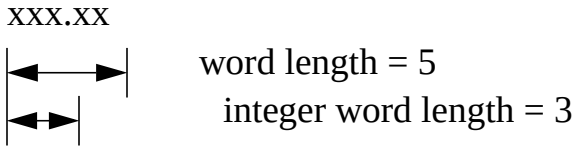
Two of the arguments specified to the fixed point data type were word length (wl) and integer word length (iwl). Word length must be greater than 0. Integer word length can be positive or negative, and larger than the word length. For instance if the word length is specified as 5 bits but the integer word length is 7 then two zeroes will be added to the end of the object. This is shown below:




If the integer word length is a negative value then sign bits after the binary point will be extended. For instance if wl = 5 and iwl = -2 then two sign bits will be added to the object as shown below:




More typical cases will not add bits. For instance if wl = 5 and iwl = 3 then the following will result:




## *Quantization Modes*

As mentioned previously quantization effects are used to determine what happens to the LSBs of a fixed point type when more bits of precision are required than are available. For instance if the result of a multiplication operation generates more LSB precision than can be represented by the result type, quantization will occur. After quantization the result is a function of the deleted bits and remaining bits of the original fixed point number.

The quantization modes available are shown by the table below:

| **Quantization Mode** | **Name** |
|---|---|
| Rounding to plus infinity | SC_RND |
| Rounding to zero | SC_RND_ZERO |

| | |
|---|---|
| Rounding to minus infinity | SC_RND_MIN_INF |
| Rounding to infinity | SC_RND_INF |
| Convergent rounding | SC_RND_CONV |
| Truncation | SC_TRN |
| Truncation to zero | SC_TRN_ZERO |

Operations performed on fixed point data types are done using arbitrary precision. After the operation is complete the resulting operand is cast to fit the fixed point data type object. The casting operation will apply the quantization behavior of the target object to the new value and assign the new value to the target object. For instance in the example below the new value is calculated with 12 bits of precision, and 4 bits right of the binary point. When writing to the second fixed point object with only 2 bits to the right of the binary point, 2 bits will have to be removed. How these bits are removed is a function of the quantization mode.

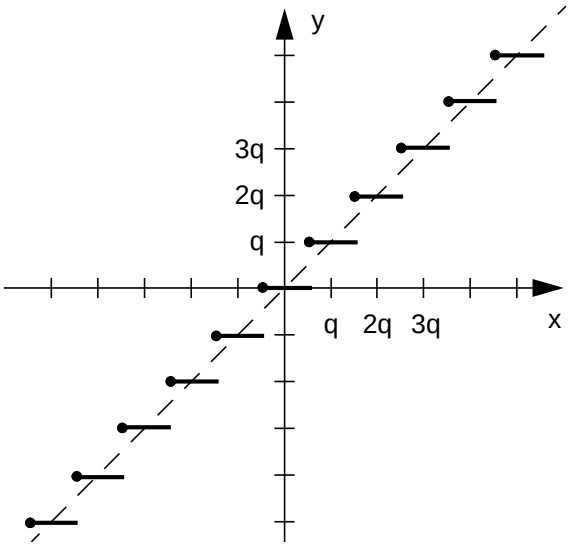
```
xxxxxxxx.xxxx   // 12 bits, 4 right of binary point
xxxxxxxx.xx     // 10 bits, 2 right of binary point
```

The next sections are going to describe each of the quantization modes in more detail.

## *SC_RND*

The SC_RND mode will round the value to the closest representable number. This is accomplished by adding the MSB of the removed bits to the remaining bits. The effect is to round towards plus infinity. A graph showing the effect of this rounding is shown below:

The x axis is the result of the previous arithmetic operation and the y axis is the value after quantization.




The diagonal line shows the ideal number representation given infinite bits. The small horizontal lines show the effect of the rounding. Any value within the range of the line will be rounded to the y value of the line. The graph is given in terms of q, which is the smallest quantization unit of the target object.

### SC_RND Examples

The first example will show the SC_RND quantization mode with a positive number. Two objects x and y are declared as sc_fixed types. A value is assigned to x. Then y is assigned the value x. However the value of x is outside the range of representation for y so quantization will occur.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_RND> y;

x = 1.25;
y = x;  // quantization will occur here

01.01  (1.25)  // representation of x value
01.1   (1.5)   // quantized y value
```

Value 1.25 is outside the range that can be exactly represented by the result fixed point type, sc_fixed<3,2,SC_RND>. Therefore quantization will occur.

When the MSB of the deleted bits is added to the remaining bits the result will be 1.5.

Here is another example using the same types, but a negative value.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_RND> y;

x = -1.25;
y = x;  // quantization will occur here

10.11  (-1.25)  // representation of x value
11.0   (-1)     // quantized y value
```

Again -1.25 is outside the representable range for the result type so quantization occurs. The MSB of the deleted bits is added to the remaining bits causing the result to be -1.

The last example shows the result with unsigned types.

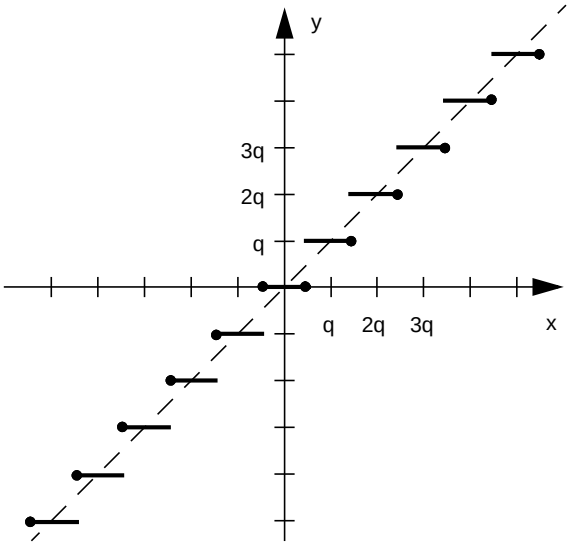
```
sc_ufixed<16,8> x;
sc_ufixed<12,8, SC_RND> y;

x = 38.30859375;
y = x;  // quantization will occur here

00100110.01001111  (38.30859375) // x value
00100110.0101      (38.3125)     // quantized y value
```

The MSB of the deleted bits is added to the remaining bits to return the result.

## *SC_RND_ZERO*

This quantization mode will perform an SC_RND operation if the two nearest representable numbers are not an equal distance apart. Otherwise rounding to zero will be performed. For positive numbers this means that the redundant bits are simply

deleted. For negative numbers the MSB of the deleted bits is added to the remaining bits. A graph showing this effect is shown below:




The diagonal line represents the ideal number representation given infinite bits. The small horizontal lines show the effect of the rounding. Any value within the range of the line will be converted to the value of the y axis.

### SC_RND_ZERO Examples

Two exampes are shown below. The first shows quantization of a positive number and the second the quantization of a negative number.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_RND_ZERO> y;

x = 1.25;
y = x;   // quantization occurs here

01.01  (1.25)  // value of x after assignment
01.0   (1)     // quantized value of y
```

Value 1.25 is outside the representation range of sc_fixed<3,2,SC_RND_ZERO> so quantization will be performed. Since this is a positive number the redundant bits are simply deleted. The next example shows the same types with a negative number.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_RND_ZERO> y;

x = 1.25;
y = x;   // quantization occurs here

10.11   (-1.25)  // value of x after assignment
11.0    (-1)     // quantized value of y
```

Value -1.25 is outside the representation range of the result type so quantization will occur. Since this value is a negative number the MSB of the deleted bits will be added to the remaining bits. Value -1.25 will be rounded to -1.

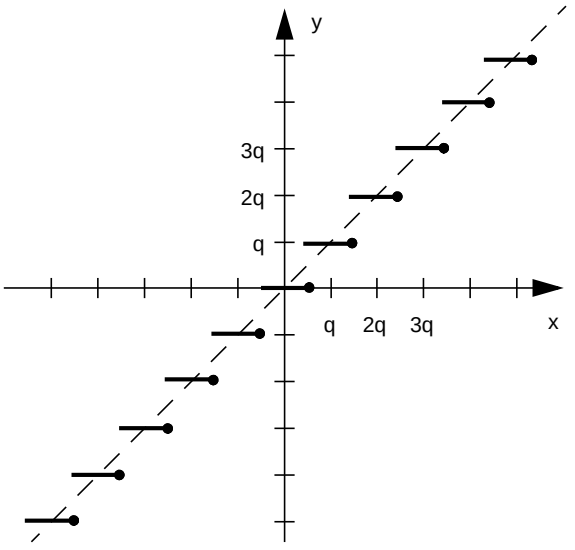
The last example shows an unsigned value.

```
sc_ufixed<14,9> x;
sc_ufixed<13,9,SC_RND_ZERO> y;

x = 38.28125;
y = x;  // quantization occurs here

000100110.01001   (38.28125) // x value after assign
000100110.0100    (38.25)    // quantized y value
```

The last example is a positive number by default so the redundant bits are deleted.

## *SC_RND_MIN_INF*

This quantization mode will also perform a check to see if the nearest 2 representable numbers are equal distance apart. If not the SC_RND quantization is performed. Otherwise this mode will round towards minus infinity by eliminating the redundant bits of the LSB of the number. A graph showing this effect is shown below:




The diagonal line represents the ideal number representation given infinite bits. The small horizontal lines show the effect of the rounding. Any value within the range of the line will be converted to the value of the y axis.

### SC_RND_MIN_INF Examples

The next two examples show the result of the SC_RND_MIN_INF quantization mode with a positive and a negative number signed number. The third example shows an unsigned number.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_RND_MIN_INF> y;

x = 1.25;
y = x;   // quantization occurs here

01.01  (1.25) // value of x after assignment
01.0   (1)    // value of y after quantization
```

Value 1.25 is outside the representable range of the result type so quantization will occur. For positive numbers the redundant bits are simply deleted resulting in the value 1. The next example uses the same types but a negative number.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_RND_MIN_INF> y;

x = -1.25;
y = x;   // quantization occurs here

10.11  (-1.25)  // value of x after assignment
10.1   (-1.5)   // value of y after quantization
```

Value -1.25 is outside the representable range for the result type, so quantization occurs. The result number is rounded towards minus infinity by removing the redundant bits. This produces the result -1.5.

The last example uses an unsigned number.

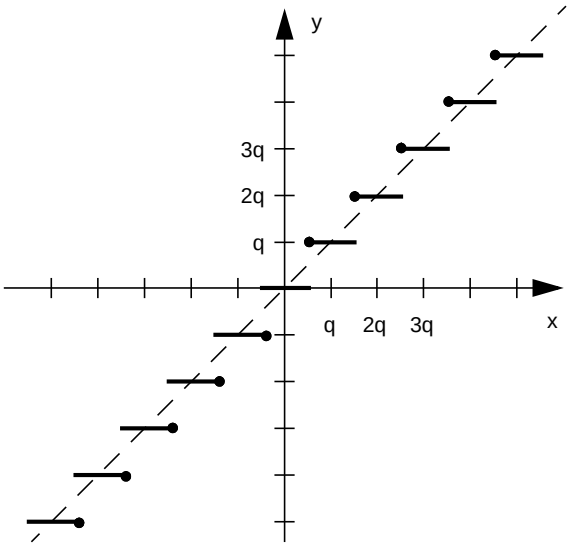
```
sc_ufixed<14,9> x;
sc_ufixed<13,9,SC_RND_ZERO> y;

x = 38.28125;
y = x;  // quantization occurs here

000100110.01001  (38.28125) // x after assign
000100110.0100   (38.25)    // y after quantization
```

For unsigned types the redundant bits are simply deleted.

## *SC_RND_INF*

This quantization mode also checks to see that the two nearest representable numbers are equal distance apart. If not, SC_RND quantization mode is applied. Otherwise the number is rounded towards plus infinity if positive or minus infinity if negative. For positive numbers the MSB of the deleted bits is added to the remaining bits. For negative numbers the redundant bits are deleted.

A graph showing this behavior is shown below:




The diagonal line represents the ideal number representation given infinite bits. The small horizontal lines show the effect of the rounding. Any value within the range of the line will be converted to the value of the y axis.

### SC_RND_INF Examples

Three examples will be shown. The first two use signed numbers and the last one an unsigned number. The first example shows quantization of a positive number and the second quantization of a negative number.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_RND_INF> y;

x = 1.25;
y = x;   // quantization occurs here

01.01   (1.25)  // value of x after assignment
01.1    (1.5)   // value of y after quantization
```

Value 1.25 is outside the representable range for the result type so quantization will occur. Since this is a positive number the MSB of the deleted bits is added to the remaining bits resulting in the value 1.5.

Here's the same quantization mode with a negative number.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_RND_INF> y;

x = -1.25;
y = x;   // quantization occurs here

10.11   (-1.25)  // value of x after assignment
10.1    (-1.5)   // value of y after quantization
```

Value -1.25 is outside the representable range for the result type so quantization will occur. Since this is a negative number the redundant bits will be deleted returning the value -1.5.

The last example shows the SC_RND_INF quantization mode with an unsigned number.

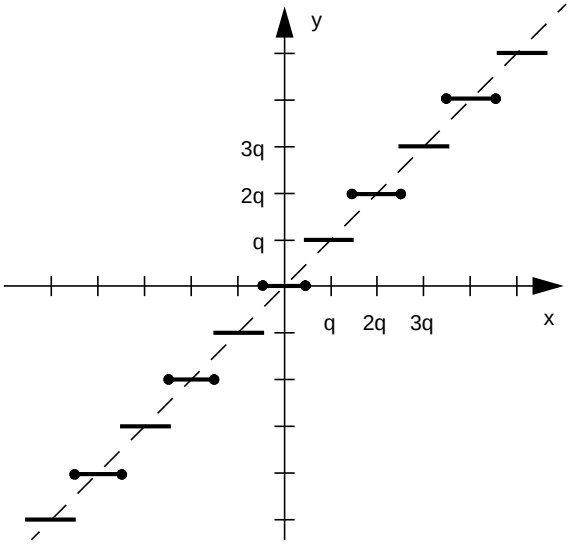
```
sc_ufixed<14,9> x;
sc_ufixed<13,9,SC_RND_ZERO> y;

x = 38.28125;
y = x;  // quantization occurs here

000100110.01001  (38.28125) // x after assignment
000100110.0101   (38.3125)  // y after quantization
```

For unsigned values the MSB of the deleted bits is added to the remaining bits.

## *SC_RND_CONV*

This quantization mode will check to see if the two closest representable numbers are equal distance apart. If not the SC_RND quantization mode is applied. Otherwise this mode checks the LSB of the remaining bits. If the LSB is 1 this mode will round towards plus infinity. If the LSB is 0 this mode will round towards minus infinity.

This behavior is shown by the graph below:




The diagonal line represents the ideal number representation given infinite bits. The small horizontal lines show the effect of the rounding. Any value within the range of the line will be converted to the value of the y axis.

### SC_RND_CONV Examples

Four examples will be shown. The first two use signed numbers and the last two unsigned numbers. The first example shows quantization of a positive number and the second quantization of a negative number.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_RND_CONV> y;

x = .75;
y = x;   // quantization occurs here

00.11   (.75)   // value of x after assignment
01.0    (1)     // value of y after quantization
```

Value .75 is outside the representable range for the result type so quantization will occur. The redundant bits are removed and the result is rounded towards plus infinity because the LSB of the remaining bits is 1.

The next example uses the same types and a negative number.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_RND_CONV> y;

x = -1.25;
y = x;   // quantization occurs here

10.11  (-1.25)   // value of x after assignment
11.0   (-1)      // value of y after quantization
```

Value -1.25 is outside the representable range for the result type so quantization will be performed. The LSB of the remaining bits is 1 so the result is rounded towards plus infinity.

The final examples shows the same quantization mode with an unsigned type.

```
sc_ufixed<14,9> x;
sc_ufixed<13,9,SC_RND_CONV> y;

x = 38.28125;
y = x;  // quantization occurs here

000100110.01001  (38.28125)  // LSB 0
000100110.0100   (38.25)     // minus infinity
```

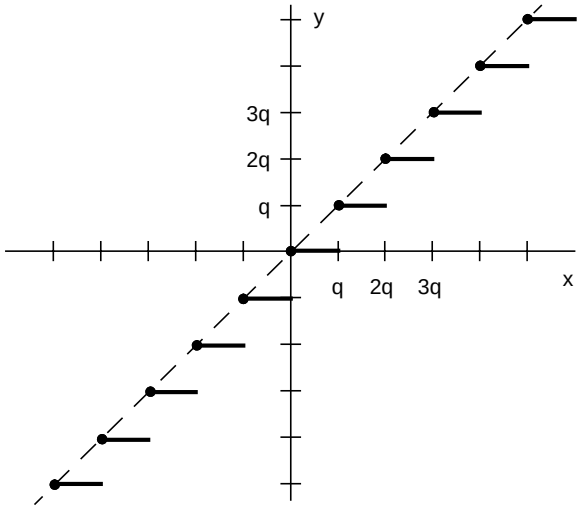
Here is an unsigned type with a different LSB value.

```
sc_ufixed<14,9> x;
sc_ufixed<13,9,SC_RND_CONV> y;

x = 38.34375;
y = x;  // quantization occurs here

000100110.01011  (38.34375)  // LSB 1
000100110.0110   (38.375)    // plus infinity
```

## *SC_TRN*

This quantization mode is the default for fixed point types and will be used if no other value is specified. The result is always rounded towards minus infinity. The redundant bits are always deleted no matter whether the number is positive or negative. The result value is the first representable number lower than the original value.

This is shown by the graph below:




The diagonal line represents the ideal number representation given infinite bits. The small horizontal lines show the effect of the rounding. Any value within the range of the line will be converted to the value of the y axis.

### SC_TRN Examples

The first two examples use an arithmetic precision of sc_fixed<4,2> with a result value of sc_fixed<3,2,SC_TRN>. Notice that the specification of SC_TRN is not required, as it is the default, but makes it quite clear which quantization mode is being used. The first example shows a positive number.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_TRN> y;
```

```
x = 1.25;
y = x;   // quantization occurs here

01.01  (1.25)  // value of x after assignment
01.0   (1)     // value of y after quantization
```

Value 1.25 is outside the representable range for the result type so quantization will be performed. The quantization simply truncates the redundant bits before assignment. In this case the LSB is removed to create the necessary result. The next example uses a negative number.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_TRN> y;

x = -1.25;
y = x;   // quantization occurs here

10.11  (-1.25)  // value of x after assignment
10.1   (-1.5)   // value of y after quantization
```

Value -1.25 is outside the representable range for the result type so quantization will occur. The LSB is simply removed creating the value -1.5.

The next example shows the same quantization mode with an unsigned value.

```
sc_ufixed<16,8> x;
sc_ufixed<12,8,SC_TRN> y;

x = 38.30859375;
y = x;  // quantization occurs here

00100110.01001111  (38.30859375)
00100110.0100      (38.25)
```

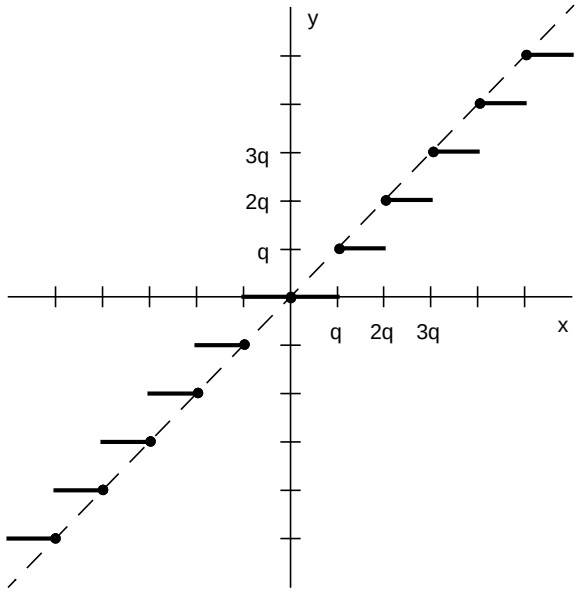
The 4 LSBs are simply removed to create the new value.

## *SC_TRN_ZERO*

For positive numbers this quantization mode is exactly the same as SC_TRN. For negative numbers the result is rounded towards zero. The result is the first repre-

sentable number lower in absolute value than the starting value. This is accomplished by deleting the redundant bits on the right side and adding the sign bit to the LSBs of the remaining bits. However this only occurs if at least one of the deleted bits is nonzero.

A graph showing this quantization mode is shown below:




The diagonal line represents the ideal number representation given infinite bits. The small horizontal lines show the effect of the rounding. Any value within the range of the line will be converted to the value of the y axis.

### SC_TRN_ZERO Examples

Two examples will be shown. The first one uses a signed number and the last one an unsigned number. The first example shows quantization of a negative number and the second quantization of an unsigned number.

```
sc_fixed<4,2> x;
sc_fixed<3,2,SC_TRN_ZERO> y;
```

```
x = -1.25;
y = x;   // quantization occurs here

10.11  (-1.25)  // value of x after assignment
11.0   (-1)     // value of y after quantization
```

Value -1.25 is outside the range of values of the result type so quantization will be performed. The LSB of the starting value is removed and the sign bit added to the LSBs. This occurs because the starting number was negative. If the starting value had been positive the result would have been a truncation of the redundant bits.

Here is another example using an unsigned type.

```
sc_ufixed<15,8> x;
sc_ufixed<12,8,SC_TRN_ZERO> y;

x = 38.30859375;
y = x;  // quantization occurs here

00100110.0100111  (38.30859375)
00100110.0100     (38.25)
```

This quantization mode for unsigned works the same as truncation because there are no negative values with unsigned numbers.

## *Overflow Modes*

In this section we will examine what happens when the result of an operation generates more bits on the MSB side of a number than are available for representation. Overflow occurs when the result of an operation is too large or too small for the available bit range. Overflow modes within the fixed point types of SystemC give the designer high level control over the result of an overflow condition.

Overflow modes are specified by the o_mode and n_bits parameters to a fixed point type. The supported overflow modes are listed in the table shown below:

| Overflow Mode | Name |
|---|---|
| Saturation | SC_SAT |
| Saturation to zero | SC_SAT_ZERO |
| Symmetrical saturation | SC_SAT_SYM |
| Wrap-around) | SC_WRAP |
| Sign magnitude wrap-around | SC_WRAP_SM |

## *MIN and MAX*

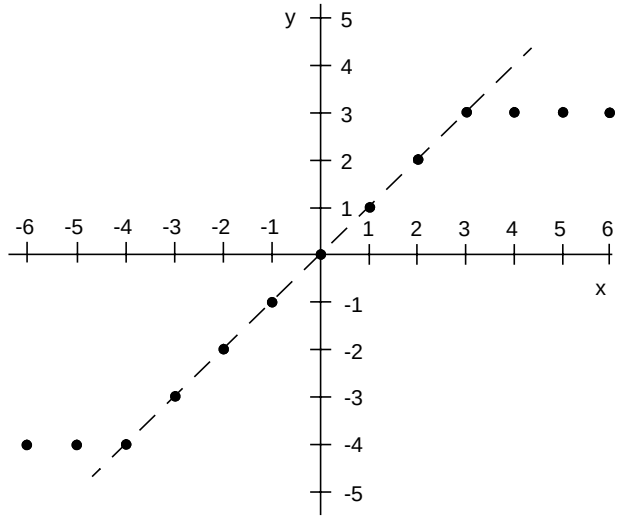
Throughout the discussion of overflow modes we will be using the terms MIN and MAX. MIN is the smallest negative number that can be represented and MAX is the largest positive number that can be represented with the available bit width.

The next few sections will discuss each of the overflow modes and their effect on the result of a cast operation.

## *SC_SAT*

This overflow mode will convert the specified value to MAX for an overflow or MIN for an underflow condition. The maximum and minimum values will be determined from the number of bits available. Value MAX will then be assigned to the result value for a positive overflow and MIN for a negative overflow condition.

A graph showing the behavior for a 3 bit type is shown below:




The diagonal line represents the ideal value if infinite bits are available for representation. The dots represent the values of the result. The X axis is the original value and the Y axis is the result. From this graph we can see that MAX = 3 and MIN = -4 for a 3 bit type.

### SC_SAT Examples

Assume that the arithmetic precision is sc_fixed<4,0> and the result is sc_fixed<3,0,SC_TRN, SC_SAT>. Then the example below will behave as shown.

```
sc_fixed<4,0> x;
sc_fixed<3,0, SC_TRN, SC_SAT> y;

x = 6;
y = x;  // overflow handling occurs here

0110  (6)  // value of x after assignment
011   (3)  // value of y after overflow handling
```

An overflow condition exists because 6 is outside the representation range for a signed 3 bit type. Therefore the value MAX (3) is assigned to the result. Below is the same types using a negative value.

```
sc_fixed<4,0> x;
sc_fixed<3,0, SC_TRN, SC_SAT> y;

x = -5;
y = x;  // overflow handling occurs here

1011  (-5)  // value of x after assignment
100   (-4)  // value of y after overflow handling
```

Value -5 is outside the range for a 3 bit signed type. The value MIN (-4) is assigned to the result.

For unsigned types the MAX value is always assigned as shown below:

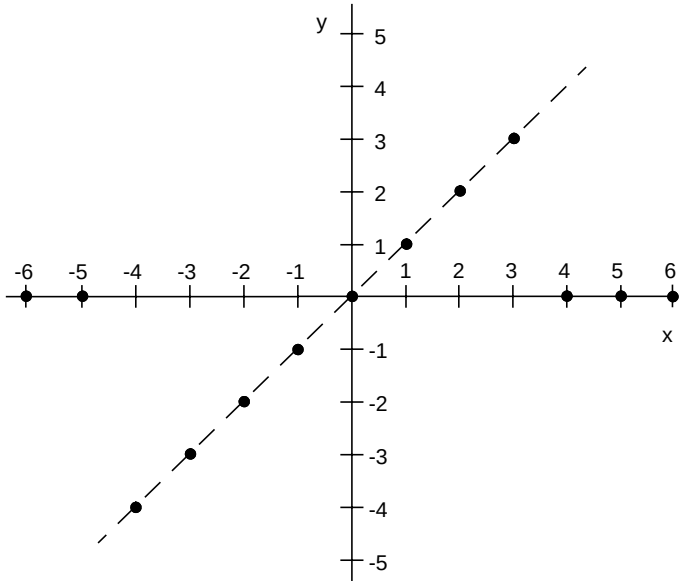
```
sc_ufixed<5,0> x;
sc_ufixed<3,0, SC_TRN, SC_SAT> y;

x = 14;
y = x;  // overflow processing occurs here

01110  (14)  // value of x after assignment
111    (7)   // value of y after overflow handling
```

Value 14 is outside the range of 3 bits unsigned, so MAX (7) is assigned to the result.

## *SC_SAT_ZERO*

This overflow mode will set the result to 0 for any input value that is outside the representable range of the fixed point type. If the result value is greater than MAX or smaller than MIN the result will be 0.

This is shown in the graph below:




The diagonal line represents the ideal value if infinite bits are available for representation. The dots represent the values of the result. The X axis is the original value and the Y axis is the result. From this graph we can see that MAX = 3 and MIN = -4 for a 3 bit type. Any value above MAX or below MIN is set to 0.

### SC_SAT-ZERO Examples

For these examples the arithmetic precision used is sc_fixed<4,0> and the result type is sc_fixed<3,0, SC_TRN, SC_SAT_ZERO>.

```
sc_fixed<4,0> x;
sc_fixed<3,0, SC_TRN, SC_SAT_ZERO> y;

x = 6;
y = x;  // overflow handling occurs here

0110  (6)  // value of x after assignment
000   (0)  // value of y after overflow handling
```

Value 6 is outside the representable range for the 3 bit result type specified so overflow processing will occur and return the value 0. Here is an example of a negative value.

```
sc_fixed<4,0> x;
sc_fixed<3,0, SC_TRN, SC_SAT_ZERO> y;

x = -5;
y = x;  // overflow handling occurs here

1011  (-5)  // value of x after assignment
000   (0)   // value of y after overflow handling
```

Value -5 is outside the representable range for the 3 bit type specified so the return value will be saturated to 0. This last example uses an unsigned type.

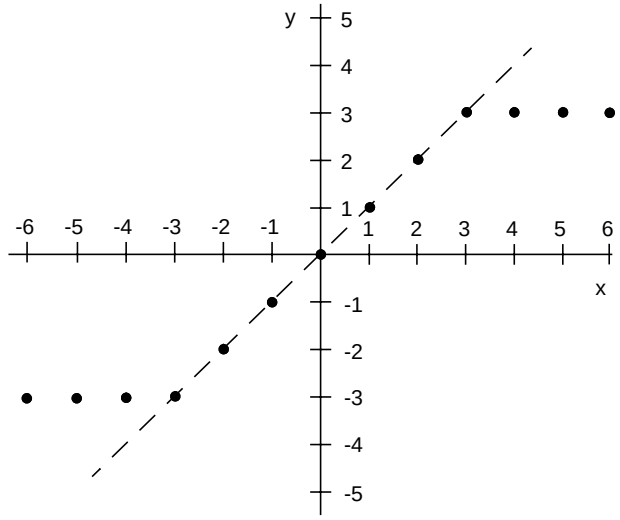
```
sc_ufixed<5,0> x;
sc_ufixed<3,0, SC_TRN, SC_SAT_ZERO> y;

x = 14;
y = x;  // overflow processing occurs here

01110  (14)  // value of x after assignment
000    (0)   // value of y after overflow handling
```

Value 14 is outside the range of 3 bits unsigned so overflow processing will occur and return the value 0.

## *SC_SAT_SYM*

In twos-complement notation one more negative value than positive value can be represented. When using SC_SAT overflow mode the absolute value of MIN is one more than MAX. Sometimes it is desirable to have the MIN and MAX value symmetrical around zero. The SC_SAT_SYM overflow mode will perform this function as required. Positive overflow will generate MAX and negative overflow will generate -MAX for signed numbers.

A graph showing this behavior is shown below:




The diagonal line represents the ideal value if infinite bits are available for representation. The dots represent the values of the result. The X axis is the original value and the Y axis is the result. From this graph we can see that MAX = 3 and MIN = -4 for a 3 bit type. An value above MAX is set to MAX for positive numbers. For negative numbers any value smaller than -MAX is set to -MAX.

### SC_SAT_SYM Examples

For the next two examples arithmetic precision is specified as sc_fixed<4,0> and the result precision is sc_fixed<3,0,SC_TRN, SC_SAT_SYM>.

```
sc_fixed<4,0> x;
sc_fixed<3,0, SC_TRN, SC_SAT_SYM> y;

x = 6;
y = x;  // overflow handling occurs here

0110  (6)  // value of x after assignment
011   (3)  // value of y after overflow handling
```

Value 6 is outside the range of values for a 3 bit signed value so the result is saturated to MAX (3). Here is a negative number example.

```
1011  (-5)
101   (-3)
```

Value -5 is outside the representable range for 3 bits so overflow processing will occur. The overflow mode will return -MAX (-3) as the result.

Here is an example using an unsigned type.

```
sc_ufixed<5,0> x;
sc_ufixed<3,0, SC_TRN, SC_SAT_SYM> y;

x = 14;
y = x;  // overflow processing occurs here

01110  (14)  // value of x after assignment
111    (7)   // value of y after overflow handling
```

Value 14 is outside the range for a 3 bit unsigned type so overflow mode will return MAX (7) as the result.

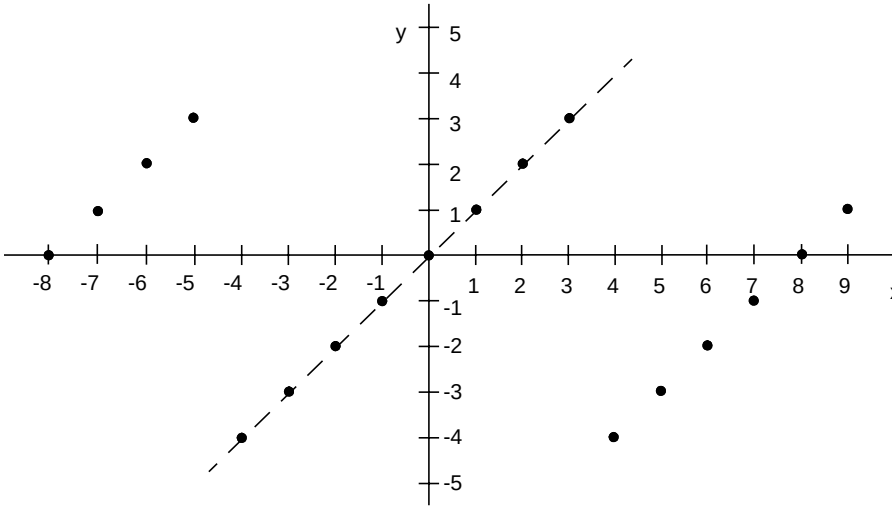
## *SC_WRAP*

With the wrap overflow modes the value of an arithmetic operand will wrap around from MAX to MIN as MAX is reached. The unsigned case is similar to the way a counter would work in hardware. When the MAX value is reached the counter would wrap around to 0 again.

There are two different cases within the SC_WRAP overflow mode. The first is with the n_bits parameter set to 0 or having a default value of 0. The second is when the n_bits parameter is a nonzero value.

## *SC_WRAP, n_bits = 0*

The first case is the default overflow mode. With this overflow mode any MSB bits outside the range of the target type are deleted. The graph below shows the behavior of this overflow mode.




The diagonal line represents the ideal value if infinite bits are available for representation. The dots represent the values of the result. The X axis is the starting value and the Y axis is the result. From this graph we can see that MAX = 3 and MIN = -4 for a 3 bit type. Notice that as the input value approaches the MAX value the next value is the MIN value. Also the next value smaller than MIN is MAX.

### SC_WRAP, n_bits = 0 Examples

The next two examples assume the original value is a signed 4 bit type and the result is a signed 3 bit type. Here is a positive number example.

```
sc_fixed<4,0> x;
sc_fixed<3,0, SC_TRN, SC_WRAP> y;

x = 4;
y = x;  // overflow handling occurs here

0100  (4)
100   (-4)
```

Value 4 is outside the representable range for 3 bits. The MSB is deleted resulting in the value -4. Here is a negative value example.

```
sc_fixed<4,0> x;
sc_fixed<3,0, SC_TRN, SC_WRAP> y;

x = -5;
y = x;  // overflow handling occurs here

1011  (-5)
011   (3)
```

Again -5 is outside the representable range for a 3 bit number, so the MSB is deleted resulting in the positive value 3.

Here is an unsigned type example.

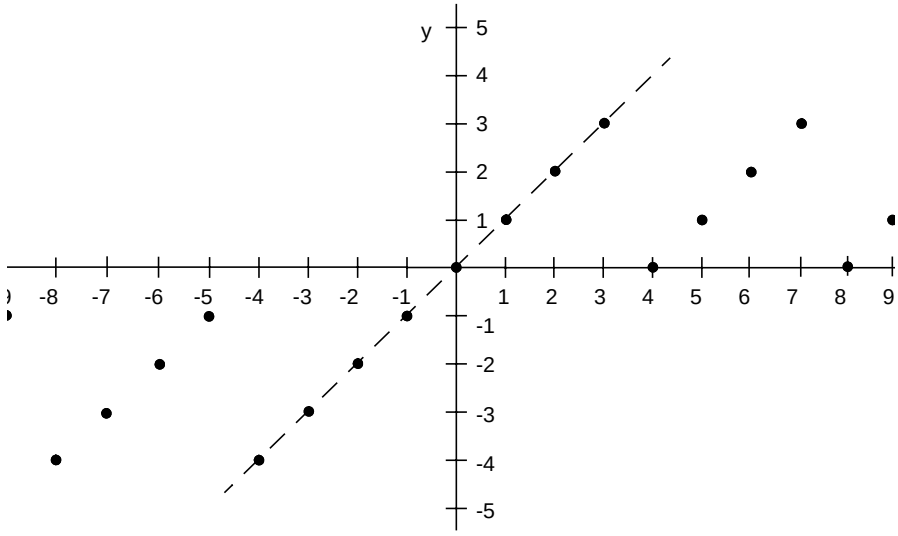
```
sc_ufixed<5,0> x;
sc_ufixed<3,0, SC_TRN, SC_WRAP> y;

x = 27;
y = x;  // overflow processing occurs here

11011  (27)
011    (3)
```

The two MSBs are deleted to fit the result into a 3 bit value.

## *SC_WRAP, n_bits > 0*

When n_bits is greater than 0 the designer is specifying that n_bit MSB bits are to be saturated or set to 1. The sign bit is retained so that positive numbers remain positive and negative numbers remain negative. The bits that are not saturated are simply copied from the original value to the result value.

A graph showing this behavior for 3 bits with n_bits = 1 is shown below:




The diagonal line represents the ideal value if infinite bits are available for representation. The dots represent the values of the result. The X axis is the starting value and the Y axis is the result. From this graph we can see that MAX = 3 and MIN = -4 for a 3 bit type. Values outside the positive representable range remain positive. Values outside the negative representable range remain negative. Notice that positive numbers wrap around to 0 while negative values wrap around to -1.

### SC_WRAP, n_bits>0 Examples

The original type for the next 2 examples is a signed 4 bit type. The result type is a signed 3 bit type. Parameter n_bits is set to 1 which will saturate 1 MSB bit.

```
sc_fixed<4,0> x;
sc_fixed<3,0, SC_TRN, SC_WRAP, 1> y;

x = 5;
y = x;  // overflow handling occurs here

0101  (5)  // value of x after assignment
001   (1)  // value of y after overflow handling
```

Value 5 is outside the representable range of 3 bits. Overflow will occur and the result wrapped to 1, still a positive number. The next example shows a negative number.

```
sc_fixed<4,0> x;
sc_fixed<3,0, SC_TRN, SC_WRAP, 1> y;

x = -5;
y = x;  // overflow handling occurs here

1011  (-5)  // value of x after assignment
111   (-1)  // vlaue of y after overflow handling
```

Value -5 is outside the range for 3 bits so overflow will occur. The sign bit will be retained and one bit saturated so the result will be -1.

The next example uses an unsigned type. This time n_bits is specified as 3.

```
sc_ufixed<7,0> x;
sc_ufixed<5,0, SC_TRN, SC_WRAP, 3> y;

x = 50;
y = x;  // overflow processing occurs here

0110010  (50) // value of x after assignment
11110    (30) // value of y after overflow handling
```

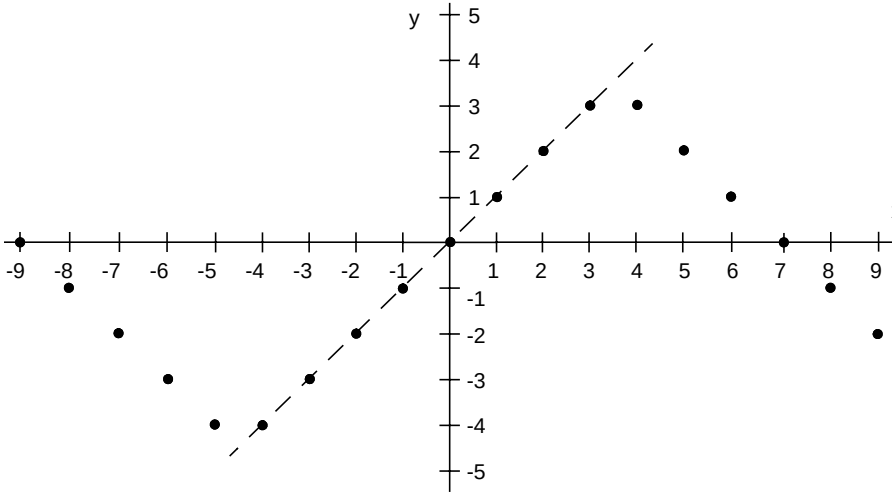
The 3 MSB bits are saturated to 1 as specified by n_bits. The other bits are copied starting from the LSB side of the starting value to the result value.

## *SC_WRAP_SM*

The SC_WRAP_SM overflow mode uses sign magnitude wrapping. This overflow mode behaves in two different styles depending on the value of parameter n_bits. When n_bits is 0 no bits are saturated. With n_bits greater than 0, n_bits MSB bits are saturated to 1.

## *SC_WRAP_SM, n_bits = 0*

This mode will first delete any MSB bits that are outside the result word length. The sign bit of the result is set to the value of the least significant deleted bit. If the most significant remaining bit is different from the original MSB then all the remaining bits are inverted. If the MSBs are the same the other bits are copied from the original value to the result value. A graph showing the result of this overflow mode is shown below:




The diagonal line represents the ideal value if infinite bits are available for representation. The dots represent the values of the result. The X axis is the starting value and the Y axis is the result. From this graph we can see that MAX = 3 and MIN = -4 for a 3 bit type. As the value of x increases, the value of y increases to MAX and then slowly starts to decrease until MIN is reached. The result is a sawtooth like waveform.

### SC_WRAP_SM, n_bits = 0 Examples

For the next few examples the starting value is a four bit representation of the value 4. If the target for this value is a 3 bit signed type the value 4 will overflow the type and overflow processing will occur. Here is the starting value:

```
sc_fixed<4,0> x;
```

```
sc_fixed<3,0,SC_TRN, SC_WRAP_SM> y;

x = 4;
y = x;  // overflow processing occurs here

0100  (4)
```

First the MSB is deleted to produce a 3 bit result.

```
100   (-4)
```

Next the new sign bit is calculated. The new sign bit is the least significant bit of the deleted bits. For this example only 1 bit was deleted and its value is 0. Therefore the new sign bit is 0. Now the sign bit of the new value (1) is compared with the calculated sign bit (0). If these bits are different, then the rest of the bits will be inverted. for this example the sign bits are different and the other bits will be inverted as shown below:

```
011   (3)
```

The sign magnitude wrap values with n_bits equal to 0 for 3 bit numbers are shown by the table below:

| **Original value in Decimal** | **Result value in Binary** |
|---|---|
| 8 | 111 |
| 7 | 000 |
| 6 | 001 |
| 5 | 010 |
| 4 | 011 |
| 3 | 011 |
| 2 | 010 |
| 1 | 001 |
| 0 | 000 |
| -1 | 111 |
| -2 | 110 |

| Original value in Decimal | Result value in Binary |
|---|---|
| -3 | 101 |
| -4 | 100 |
| -5 | 100 |
| -6 | 101 |
| -7 | 110 |

This table shows what happens when the original values in the left cell of the table are converted to result values in the table cells on the right.Notice that the original values are listed in decimal to show greater range.

## *SC_WRAP_SM, n_bits > 0*

The second overflow behavior within the SC_WRAP_SM overflow mode is the case when n_bits is greater than 0. A sign magnitude wrap will still be performed but now n_bits MSB bits will be saturated. In fact the first n_bits MSB bits on the MSB side of the result number will are saturated to MAX for positive numbers and to MIN for negative numbers. This means that all of the bits except for the sign bit will be saturated to a 1 for positive numbers and all of the bits will be saturated to 1 for negative numbers. Positive number remain positive and negative numbers remain negative.

When n_bits is equal to 1, one bit to the right of the sign bit is saturated and the remaining bits are copied. These remaining bits are xor-ed with the original and new value of the sign bit of the result number. If n_bits is greater than 1, the unsaturated bits are xor-ed with the original value of the least significant saturated bit and the inverse value of the original sign bit.

### SC_WRAP_SM, n_bits = 3 Examples

For this example the original number is a 9 bit number and the result will be 5 bits. Parameter n_bits is equal to 3. This will cause the first 3 MSBs of the new value to be saturated to MAX or MIN. Here’s the starting value.

```
sc_ufixed<9,0> x;
sc_ufixed<5,0,SC_TRN,SC_WRAP,3> y;
```

```
x = 234;
y = x;  // overflow processing occurs here

011101010  (234)
```

This value is first truncated to 5 bits.

```
01010  (10)
```

The original sign bit (0) is copied to the MSB of the new value. Next bits 4, 3, and 2 are converted to MAX because n_bits is equal to 3. The sign bit is not saturated to 1, because the sign does not change in this mode.

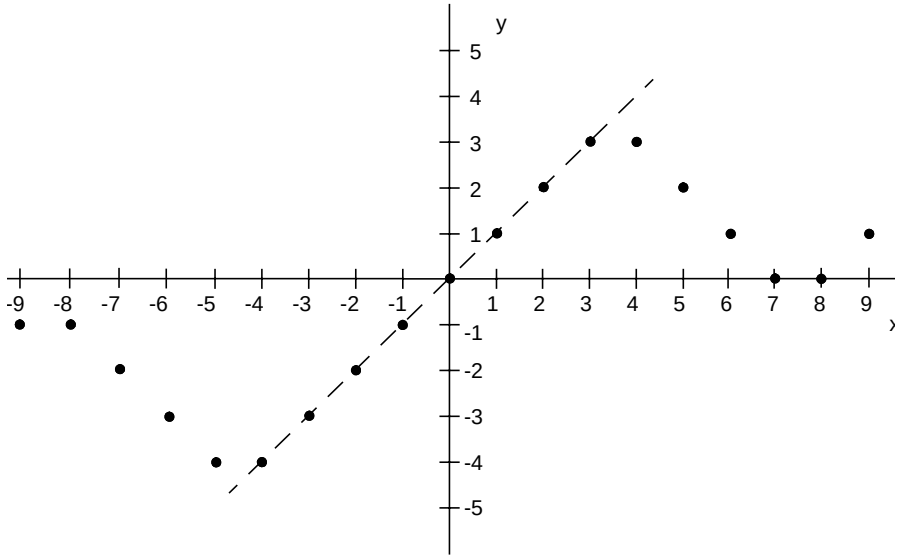
```
01110  (14)
```

The original value of the bit at position 2 (starting with 0 at right) was 0. The remaining bits at the LSB side (10) are xor-ed with this value and the inverse value of the original sign bit (01). The final result is shown below.

```
01101  (13)
```

## *SC_WRAP_SM, n_bits = 1*

This overflow mode behaves similarly to the mode where n_bits equals 0 except that positive numbers stay positive and negative number stay negative. The first bit on the MSB side of the new value will receive the sign bit of the original value. The other bits are copied and xor-ed with the original and the new value of the result sign bit. This behavior is shown in the graph below:




The diagonal line represents the ideal value if infinite bits are available for representation. The dots represent the values of the result. The X axis is the starting value and the Y axis is the result. From this graph we can see that MAX = 3 and MIN = -4 for a 3 bit type. Notice that while the graph looks somewhat like a sawtooth waveform, positive numbers do not dip below 0 and negative numbers do not cross -1.

### SC_WRAP_SM, n_bits = 1 Example

This example will cast a 5 bit representation of the number 12 to a 3 bit number using the SC_WRAP_SM overflow mode with n_bits equal to 1. Here's the original value.

```
sc_ufixed<5,0> x;
sc_ufixed<3,0,SC_TRN,SC_WRAP,1> y;

x = 12;
y = x;  // overflow processing occurs here

01100  (12)
```

This value is first truncated to 3 bits.

```
100  (4)
```

The original sign bit is copied to the MSB position.

```
000  (0)
```

The two remaining LSB bits are xor-ed with the original sign bit (1) and the new sign bit (0).

```
011  (3)
```

This algorithm can be applied to any number that cannot be exactly represented by 3 bits.

The table below summarizes the overflow behavior for 3 bits.

| Original value in Decimal | Result value in Binary |
|---|---|
| 9 | 001 |
| 8 | 000 |
| 7 | 000 |
| 6 | 001 |
| 5 | 010 |
| 4 | 011 |
| 3 | 011 |
| 2 | 010 |
| 1 | 001 |
| 0 | 000 |
| -1 | 111 |
| -2 | 110 |
| -3 | 101 |
| -4 | 100 |
| -5 | 100 |
| -6 | 101 |

| Original value in Decimal | Result value in Binary |
|---|---|
| -7 | 110 |
| -8 | 111 |
| -9 | 111 |

## *Fast Fixed Point Types*

The standard fixed point types described previously use arbitrary precision in calculations. This adds extra overhead that in most cases might not be needed. SystemC provides limited precision fixed point types to speed simulation when limited precision is all that's required.

With standard fixed point types the mantissa can be virtually any size. With limited precision fixed point types the mantissa is limited to 53 bits. Limited precision fixed point types are implemented with double precision floating point values. The range of representation of limited precision fixed point types is limited by the size of the double precision floating point value representation.

The 4 limited precision fixed point types are listed below:

- sc_fixed_fast
- sc_ufixed_fast
- sc_fix_fast
- sc_ufix_fast

The limited precision types have exactly the same interface as the arbitrary precision fixed point types. The same parameter names, types, and order are used to form both kinds of fixed point types. Also limited precision and arbitrary precision types can be mixed freely.

To get bit-true behavior for a design follow these guidelines:

- Make sure that the result of any operation with fast fixed point types does not generate a word length greater than 53 bits.

- When adding or subtracting two operands the result word length will be 1 more than the maximum aligned word length.
- When multiplying two operands the resulting bit length will be the sum of the word length of each operand.

Limited precision fixed point types should be used whenever possible to achieve the best simulation performance. Apply the guidelines from above to make sure that the limited precision types will be appropriate for your design.

## *Simple Examples*

Here are some simple examples to show how the fixed point types will be used. The first example is a simple adder with floating point inputs and output types.

```
// fxpadder.h

#include “systemc.h”

float adder(float a, float b)
{
  sc_fixed_fast<4,2,SC_RND, SC_WRAP> Inputa = a;
  sc_fixed_fast<6,3, SC_RND, SC_WRAP> Inputb = b;

  sc_fixed_fast<7,4,SC_RND, SC_WRAP> Output;

  Output = (Inputa + Inputb);
  return (Output);
}
```

This example is a simple adder with two floating point input argument and 1 floating point output return value. The declarations of Inputa and Inputb declare fixed point input types and conversions from floating point types. The declaration of Output specifies a fast fixed point type whose bit width is one greater than the biggest input operand. The assignment to variable Output performs the add operation and the return statement will assign the new result to the function output value. When the assignment is performed the fast fixed point type is converted back to a float.

This example allows the designer to easily change the bit widths, overflow modes, and quantization modes to get the desired adder behavior. The designer can simu-

late the behavior before implementation to see if the adder is functionally what is needed for the end product.

## *Type sc_fxtype_params*

Type sc_fxtype_params is used to configure the parameters of types sc_fix_fast, sc_ufix_fast, sc_fix, and sc_ufix. Remember these types do not need to have their parameters determined at compile time as do types sc_fixed, sc_ufixed, sc_fixed_fast, and sc_ufixed_fast. Therefore to set the parameters for these types declare an object of type sc_fxtype_params, initialize the parameter values as desired, and pass the sc_fxtype_params object as an argument to the sc_fix_fast, etc. declarations.

The sc_fxtype_params object has the same arguments passed to an object of type sc_fixed_fast. These include:

- wl - word length
- iwl - integer word length
- q_mode - quantization mode
- o_mode - overflow mode
- n_bits - saturated bits

These arguments are exactly as described in the last few sections. For instance a sc_fxtype_params object could be created as follows:

```
sc_fxtype_params small_add_params(8, 4, SC_RND,
SC_SAT);
```

This creates an object called small_add_params that contains the following parameter values:

- wl = 8
- iwl = 4
- q_mode = SC_RND
- o_mode = SC_SAT
- n_bits = 0 (default)

Any combination of arguments are allowed, but the order cannot be changed. A variable of type sc_fxtype_params can be initialized by another variable of type sc_fxtype_params. One variable of type sc_fxtype_params can also be assigned to another.

Individual argument values can be read and written using methods with the same name as the arguments shown above. Here's an example:

```
sc_fxtype_params small_add_params(8, 4, SC_RND,
SC_SAT);

x = small_add_params.wl();  // x = 8
small_add_params.iwl() = 4; // sets iwl to 4
```

The first statement will create a sc_fxtype_params object with wl = 8, iwl = 4, q_mode = SC_RND, and o_mode = SC_SAT. The second statement will read the value of wl, and the third statement will set the value of iwl.

## *Type sc_fxtype_context*

Type sc_fxtype_context is used to configure the default behavior of fixed point types. This type will set the default values for parameters to declaration of types sc_fix_fast, sc_ufix_fast, sc_fix, and sc_ufix. This type allows the designer to create a set of default parameter values and define when these values are used.

When a new sc_fxtype_context object is created the values specified as arguments become the new default values. The old default values are stored. When the new context goes out of scope the old default values are restored.

An example using both the sc_fxtype_params and sc_fxtype_context is shown below:

```
// fxpadder2.h
#include “systemc.h”

sc_fxtype_params myparams(SC_RND, SC_SAT);
sc_fxtype_context mycontext(myparams);

sc_fix_fast adder(sc_fix_fast a, sc_fix_fast b)
{
```

```
  sc_fix_fast Output(a.wl() +1, a.iwl() +1);
  // specify output wl and iwl to be one larger
  // than wl and iwl of a

  Output = a + b;
  return(Output);
}
```

This example uses the sc_fix_fast type in an adder. The first two declarations setup the quantization mode and overflow mode used in the description. The first statement will declare an sc_fxtype_params object (myparams) to specify the fixed point default parameter values. Notice that the wl and iwl parameters were not specified so the current default parameter values will be used.

The second statement creates a new sc_fxtype_context object and initializes the context with the default values of the sc_fxtype_params object created earlier. This context will now be active for all fixed point objects created in the scope of this declaration.

The declaration of adder specifies two input parameters and the output as sc_fix_fast types. When these types are declared they will pick up the overflow mode and quantization mode setup in context mycontext by default.

The declaration of Output specifies that the word length and integer word length will be one longer than the word length and integer word length of input a. Notice the use of methods wl() and iwl() to return the current values.

The last two statements will add a and b, assign the result to Output, and return the result. If any quantization or overflow handling is needed it will be performed when the assignment to Output takes place. The last statement assigns Output to the return value of the function. If needed more quantization and overflow handling could also occur when this statement executes.

### Complex Context Example

Contexts have the ability to be turned on and off. This can be very useful when using a number of different default values throughout your design. To declare a context but don’t use it right away use the SC_LATER argument. This is shown below:

```
sc_fxtype_params param1(12,3);
// not specified arguments are coming from
// the actual context.

sc_fxtype_params param2(32,3,SC_RND,SC_SAT);
sc_fxtype_params param3(16,16,SC_TRN,SC_SAT_ZERO);
```

First three sets of sc_fxtype_params objects have been created to hold the different values for the different contexts.

```
{
  ..............
  sc_fxtype_context c_1(param1,SC_LATER);
  /* only declaration of a context */
  sc_fxtype_context c_2(param2);
  /* declaration of a context and the
     parameter specified in param2 are
     the new default one */
  sc_fxtype_context c_3(param3, SC_LATER);
  /* only declaration of a context */
```

Next three contexts are created using each of the parameter sets created. The SC_LATER argument for parameter sets param1 and param3 mean that these parameter sets will not be currently active. These sets can be activated later by using a begin() method on variables c_1 and c_3. This will be shown below:

```
  sc_fix a;
  // is equivalent to sc_fix(32,3,SC_RND,SC_SAT) a;
  // because param2 is the default parameter set
  c_1.begin();
  // parameters specified in param1 are from now
 // on the new default ones. This is because param1 has
  // only word length and integer word length
  // speciifed, the quantization and overflow
  // modes are the built-in ones (SC_TRN, SC_WRAP)

  sc_fix b;
  // is equivalent to sc_fix(12,3,SC_TRN, SC_WRAP) b;
```

```
// because parameter set 1 is now active

c_3.begin();
// This will activate parameter set param3 making
// the default

sc_fix c;
// This declaration will use parameter set param3
// just activated so this declaration is equivalent
// to sc_fix(16,16,SC_TRN,SC_SAT_ZERO) c;

sc_fixed<13,5> zz;
// This declaration is equivalent to sc_fixed<13,5,
// SC_TRN, SC_WRAP> zz. The context has no influence
// for fixed point types sc_fixed and sc_ufixed, the
// built-in defaults are always used.

c_3.end();
// This will turn off the c_3 context so paramter set
// param3 is no longer valid. Parameter set param1
// will now be activated again.

sc_fix d;
// Parameter set param1 is used so this declaration is
// equivalent to sc_fix(12,3,SC_TRN, SC_WRAP) d;

c_1.end();
// This statement will turn off the c_1 context so
// parameter set param2 will be active again.

sc_fix e;
// Parameter set param2 is used so this declaration is
// equivalent to sc_fix(32,3,SC_RND,SC_SAT) e;

c_2.end();
// This statement will turn off the c_2 context so
// the built-in default values will now be used.

sc_fix f;
// This declaration uses the built-in default values
// so this declaration is equivalent to
```

```
// sc_fix(32,32,SC_TRN, SC_WRAP) f;
```

## *Operators*

There are a number of operators defined for fixed point types, as shown in the table below:

| Operator class | Operators in class |
|---|---|
| Bitwise | `~ & ^ \|` |
| Arithmetic | `* / + - << >> ++ --` |
| Equality | `== !=` |
| Relational | `< <= > >=` |
| Assignment | `= *= /= += -= <<= >>= &= ^= \|=` |

All of the normal arithmetic and equality operators are supported including an arithmetic shift left (<<) and arithmetic shift right (>>). The difference between the arithmetic shifts and the standard bit shifts are that the arithmetic shifts preserve the sign bit.

A small set of bitwise operators are defined for fixed point types. These operators are defined to work exclusively on signed or unsigned operands. No mixing of signed and unsigned operands is allowed. Also no mixing with any other type is allowed.

For the ~(not) operator the return type is the type of the operand. The bits in the two's complement mantissa are inverted to get the mantissa of the result. For binary operators the type of the result is the maximum aligned type (the longest width) of the two operands. The two operands are aligned by the binary point. The maximum word length and maximum fractional word length are taken. Both operands are converted to this type before performing the bitwise and, or, or xor operation.

## Bit Selection

As with other types that have already been discussed, bit selection is performed with []. The return type of this operation is type sc_fxnum_bitref which behaves like sc_bit. Bit selection can be used for reading and writing a single bit of a fixed point type.

## Part Selection

Part selection is performed with the range() method as with other types. The return type of the part selection is sc_fxnum_subref which behaves like sc_bv. Part selection can be performed on both sides of an assignment statement allowing both reading and writing of a part.

## Type Casting

Type casting is very important for fixed point types. Type casting is performed during initialization (if required) and assignment. Type casting will first use quantization to reduce then number of bits of the LSB side of the operand. Next overflow handling is performed to reduce the number of bits at the MSB side of the operand. Sign extension and zero fill are used in cases where the operand is not reduced but extended.

Type casting can be configured to be on or off. The default value of the cast switch is obtained from the current sc_fxtype_context object in use. Casting can be turned on or off through an argument during declaration, or by modifying or creating a new context. Here's an example:

```
sc_ufixed<16,16> d(SC_OFF);
```

This declaration specifies d as an unsigned 16 bit fixed point type in which casting is turned off. Values for the cast switch are SC_OFF and SC_ON. The default value is SC_ON.

Turning casting off will turn off fixed point handling of the operand. The operand will be treated as a large float value. The bit accurate behavior of the operand will not be available when casting is turned off.

## *Useful State Information*

There are some useful methods to query the state of a fixed point object. These methods return a boolean value depending on the value of a fixed point object. The following methods are supported:

- is_neg() - returns true if object has a negative value, otherwise returns false.
- is_zero() - returns true if object is zero value, otherwise returns false.
- overflow_flag() - returns true if last write to this object caused overflow to occur. Returns false if no overflow.
- quantization_flag() - returns true if last write to this object caused quantization to occur. Returns false if no quantization.

To use one of the methods append the method name to the variable name with a "." as shown below:

```
if (myvariable.is_zero()) { // do something
```

In this example if myvariable is 0 the if statement will be true.

## *Converting Fixed Point Types to Strings*

The value of a fixed point type can be converted to a character string with the to_string() method. This method takes two arguments. The first argument specifies the number representation of the result and the second specifies fixed or scientific format. The number representation argument is specified by one of the arguments from the table below:

| Value | Description | Prefix |
|---|---|---|
| SC_DEC | decimal, sign magnitude | |
| SC_BIN | binary, two's complement | 0b |
| SC_BIN_US | binary, unsigned | 0bus |
| SC_BIN_SM | binary, sign magnitude | 0bsm |
| SC_OCT | octal, two's complement | 0o |

| | | |
|---|---|---|
| SC_OCT_US | octal, unsigned | 0ous |
| SC_OCT_SM | octal, sign magnitude | 0osm |
| SC_HEX | hexadecimal, two's complement | 0x |
| SC_HEX_US | hexadecimal, unsigned | 0xus |
| SC_HEX_SM | hexadecimal, sign magnitude | 0xsm |
| SC_CSD | canonical signed digit | 0csd |

To specify how a number is represented use the following syntax:

```
varname.to_string(number representation, format);
```

Both arguments are optional. The default number representation is SC_DEC. The second argument (format) can be SC_F for fixed notation and SC_E for scientific notation. The default is SC_F for types sc_fixed, sc_ufixed, sc_fix and sc_ufix and the corresponding fast versions.

## *Arrays of Fixed Point Types*

When declaring a single fixed point object, each object can receive constructor arguments. For arrays of fixed point types, this is not the case. For arrays the default constructor will be used for each element. The constructor arguments are passed through the current context in use.

For sc_fix, sc_ufix and the corresponding fast types setup a context before the array declaration as shown below:

```
sc_fxtype_context c1(16, 1, SC_RND_CONV, SC_SAT_SYM);
sc_fix a[10];
```

This will create an array of 10 fixed point types that are 16 bits long, have 1 integer point, use SC_RND_CONV for quantization, and SC_SAT_SYM for overflow.

For sc_fixed, sc_ufixed and the corresponding fast types the arguments are passed as previously described. This is shown below:

```
sc_fixed<32,32> a[10];
sc_ufixed_fast<16,1,SC_RND_CONV,SC_SAT_SYM> b[4];
```

The first statement creates an array of 10 signed fixed point types 32 bits long with 32 bits left of the binary point. The second statement creates an array of 4 unsigned fixed point types 16 bits long, 1 bit to the left of the binary point, that uses the SC_RND_CONV quantization mode and the SC_SAT_SYM overflow mode.

For the sc_fixed and sc_ufixed types and the corresponding fast types the cast switch must be setup properly in the context as it cannot be passed as an argument.

### Larger Example

This example is a 17 coefficient FIR filter. This function takes one argument named Input of type sc_fixed<4,2,SC_RND, SC_WRAP> and returns a value of type sc_fixed<32,3,SC_RND,SC_WRAP>. The input value and the last 16 input values are successively multiplied by the 17 input coefficents. The input value is then stored in the state array to be used as one of the 16 values in the next calculation. As a new value is received the values in the state array are shifted to make room for the new value.

```
#include "ccfxd.h"

sc_fixed<32,3,SC_RND, SC_WRAP> fir_fx(sc_fixed<4,2,
      SC_RND, SC_WRAP> Input)
{
  const int NumberOfCoefficients = 17;
  static sc_fixed<4,2,SC_RND, SC_WRAP>
               state[NumberOfCoefficients-1];

  static sc_fixed<32,0,SC_RND, SC_WRAP>
               coeff[NumberOfCoefficients] = {
        1.05162989348173e-02,
        3.84160084649920e-03,
       -1.86606831848621e-02,
       -3.90706136822701e-02,
       -2.64619290828705e-02,
        3.91649864614010e-02,
        1.44576489925385e-01,
        2.5e-01,
        2.84146755933762e-01,
        2.43584483861923e-01,
        1.44576489925385e-01,
        3.91649864614010e-02,
```

```
        -2.64619290828705e-02,
        -3.90706136822701e-02,
        -1.86606831848621e-02,
         3.84160084649920e-03,
    1.05162989348173e-02};

  sc_fixed<32,3,SC_RND, SC_WRAP> Output;
  sc_fixed<4,2,SC_RND, SC_WRAP> * pstate;
  sc_fixed<32,0,SC_RND, SC_WRAP> * pcoeff;
  sc_fixed<32,3,SC_RND, SC_WRAP> sum;
  int i;

   /* FIR filter output */
  pcoeff = &coeff[0];
  pstate = &state[0];
  sum = ((*pcoeff++ ) * (Input));
  for (i = 0;i < (NumberOfCoefficients - 1);i++)
   {
      sum = (sum + ((*pcoeff++ ) * (*pstate++ )));
   }
  Output = sum;
   /* shift state */
  pstate = &state[(NumberOfCoefficients - 2)];
  pcoeff = (pstate - 1);
  for  (i = 0; i < (NumberOfCoefficients - 2); i++)
   {
      *pstate--  = *pcoeff-- ;
   }
  *pstate = Input;
  return(Output);
}
```

CHAPTER 9

# *Simulation and Debugging Using SystemC*

After you write a system description in SystemC, you typically want to simulate it as the next step in the design flow. This chapter describes the simulation control facilities provided by SystemC to start and stop a simulation, query the current time, and understand the order in which various processes are executed.

Writing a system description in SystemC gives you the advantage of using standard C++ development tools for compiling and debugging. This chapter describes the additional facilities that can help you debug SystemC programs.

## *Advanced Topic: SystemC Scheduler*

SystemC simulation is cycle-based: processes are executed and signals are updated at clock transitions. The SystemC library includes a cycle-based scheduler that handles all events on signals, and it schedules processes when the appropriate events happen at their inputs. SystemC simulation follows the evaluate-update paradigm where all processes that are ready to be executed are executed, and only then are their output signals updated.

The scheduler in SystemC executes the following steps during simulation.

1. All clock signals that change their value at the current time are assigned their new values.
2. All SC_METHOD/SC_THREAD processes with inputs that have changed are executed. The entire body of SC_METHOD function processes are executed, while SC_THREAD processes are executed until the next `wait()` statement suspends execution of the process. SC_METHOD/SC_THREAD processes are not executed in a fixed order.
3. All SC_CTHREAD processes that are triggered have their outputs updated, and they are saved in a queue to be executed later in step 5. All outputs of SC_METHOD/SC_THREAD processes that were executed in step 1 are also updated.
4. Steps 2 and 3 are repeated until no signal changes its value.
5. All SC_CTHREAD processes that were triggered and queued in step 3 are executed. There is no fixed execution order of these processes. Their outputs are updated at the next active edge (when step 3 is executed), and therefore are saved internally.
6. Simulation time is advanced to the next clock edge and the scheduler goes back to step 1.

If processes communicate using signals, the process execution order should not affect the simulation results. However, if global variables and pointers are used, process execution order affects the simulation results. Note that these simulation semantics are similar to Verilog simulation semantics with deferred signal assignments and VHDL simulation semantics.

## *Simulation Control*

You can only start simulation after you instantiate and properly connect all modules and signals. In SystemC, simulation starts by calling `sc_start()` from the top level, namely the `sc_main()` routine. The `sc_start()` function takes a variable of type `double` as an argument and simulates the system for as many time units as the value of the variable. If you want the simulation to continue indefinitely, provide a negative value for the argument to this function. This routine generates all the clock signals at the appropriate times and calls the SystemC scheduler.

Simulation can be stopped anytime (from within any process) by calling `sc_stop()`. The function does not take arguments. Simulation is stopped either at step 4 or at step 6 in the scheduler loop.

You can determine the current time during simulation by calling `sc_time_stamp()`. This function returns the current simulation time in a variable of type `double`.

To aid in debugging during simulation, variables, ports, and signal values can be read and printed. The printed value of a port or a signal is the current value of the port or signal, not a value just written to it.

### Advanced Simulation Control Techniques

You have the option to use a different method to generate clocks and control simulation than using `sc_start()`. To do that, you have to first call `sc_initialize()` to initialize the SystemC scheduler. Then you can set signals to values by writing to them, and calling the routine `sc_cycle()` to simulate the result of setting the signals. This function takes a variable of type `double` as an argument. It calls the SystemC scheduler, executes steps 2 through 5 of the scheduler loop (simulates until the current effects of the signal writes are propagated throughout the system). It then advances simulation time by the amount given as the argument to the function. For example, if the time unit is nS, `sc_cycle(10)` advances the simulation time by 10nS.

For examples, assume you have defined a clock as:

```
sc_clock clock(“my clock”, 20, 0.5);
```

You can simulate the generation of clocks for 200 time units by calling

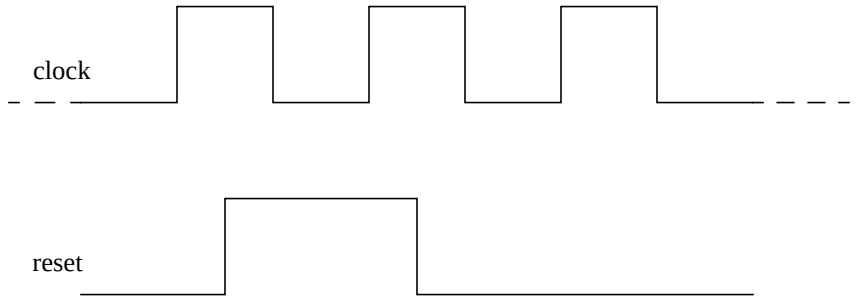
```
sc_start(200);
```

On the other hand, you can generate the clock yourself by doing the following:

```
sc_initialize();
for (int i = 0; i <= 200; i++)
   clock = 1;
   sc_cycle(10);
   clock = 0;
   sc_cycle(10);
}
```

Using this capability, you can inject events asynchronously with respect to the clock into the system, as shown in the following drawing.

**FIGURE 18. Signal Asynchronous to Clock**




To implement this, you can write the following in `sc_main()`:

```
sc_initialize();
// Let the clock run for 10 cycles
for (int i = 0; i <= 200; i++)
   clock = 1;
   sc_cycle(10);
   clock = 0;
   sc_cycle(10);
}

// Inject asynchronous reset
clock = 1;
sc_cycle(5);
reset = 1;
sc_cycle(5);
clock = 0;
sc_cycle(10);
clock = 1;
sc_cycle(5);
reset = 0;
```

```
sc_cycle(5);
clock = 0;
sc_cycle(10);

// Now let the clock run indefinitely
for (;;)
   clock = 1;
   sc_cycle(10);
   clock = 0;
   sc_cycle(10);
}
```

Note that `sc_cycle()` can only be called from the top level similar to `sc_start()`.

## *Tracing Waveforms*

SystemC provides functions that let you create a VCD (Value Change Dump), ASCII WIF (Waveform Intermediate Format), or ISDB (Integrated Signal Data Base) file that contains the values of variables and signals as they change during simulation. The waveforms defined in these files can be viewed using standard waveform viewers that support the VCD, WIF, or ISDB formats.

In generating waveforms, note the following:

- Only variables that are in scope during the entire simulation can be traced. This means all signals and data members of modules can be traced. Variables local to a function cannot be traced.
- Variables and signals of scalar, array and aggregate types can be traced.
- Different types of trace files can be created during the same simulation run.
- A signal or variable can be traced any number of times in different trace formats.

### Creating the Trace File

The first step in tracing waveforms is creating 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.

To create a WIF file, the `sc_create_wif_trace_file()` function needs to be called. For example,

```
sc_trace_file *trace_file;
my_trace_file = sc_create_wif_file(“my_trace”);
```

creates the WIF file named `my_trace.awif` (the `.awif` extension is automatically added). Similarly, an ISDB trace file can be created.

At the end of simulation the trace files need to be closed or errors can result. Close the trace files with one of the following functions.

```
sc_close_isdb_trace_file(my_trace_file);
sc_close_wif_trace_file(my_trace_file);
sc_close_vcd_trace_file(my_trace_file);
```

Call the function appropriate to the type of file that was created. Call this function just before the return statement in your sc_main routine.

### Tracing Scalar Variable and Signals

SystemC provides tracing functions for scalar variables and signals. All tracing functions have the following in common:

- The function is named `sc_trace()`.
- Their first argument is a pointer to the trace file data structure `sc_trace_file`.
- Their second argument is a reference or a pointer to a variable being traced.
- Their third argument is a reference to a string.

For example, the following illustrates how a signal of type `int` and a variable of type `float` are traced.

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

sc_trace(trace_file, a, “MyA”);
sc_trace(trace_file, b, “B”);
```

In this example, `trace_file` is a pointer of type `sc_trace_file`, that was created earlier. “`MyA`” is the name of the `int` variable as it would appear in the waveform viewer, and “`B`” is the name of the `float` variable.

The trace function registers (creates a list of) the signals and variables to be traced. The actual tracing happens during simulation and is handled by the SystemC scheduler. Note that calls to the `sc_trace()` functions are made only after the processes and signals are instantiated and after the trace file is opened.

## Tracing Variables and Signals of Aggregate Type

The trace functions defined in SystemC can accept signals or variables of scalar types only. To trace variables of aggregate type, you need to define special trace functions for variables of these types using the basic trace functions that are provide in SystemC.

For example, consider the structure

```
struct bus {
   unsigned address;
   bool read_write;
   unsigned data;
};
```

You need to define a trace function for this structure as follows:

```
void sc_trace(sc_trace_file *tf, const bus& v, const
sc_string& NAME)
{
   sc_trace(tf, v.address, NAME + “.address”);
   sc_trace(tf, v.read_write, NAME + “.rw”);
   sc_trace(tf, v.data, NAME + “.data”);
}
```

When called, this trace function traces the data structure by tracing individual fields of the structure. Note that each individual field of the structure is given a unique name by appending the field name to the structure name.

### Tracing Variable and Signal Arrays

To trace a variable or signal array, you need to define a specialized trace function using the basic data or signal trace functions SystemC provides. For example, the trace function for arrays of type `sc_signal<int>` are

```
void sc_trace(sc_trace_file *tf, sc_signal<int> *v,
const sc_string& NAME, int len)
{
   char stbuf[20];
   for (int i = 0; i< len; i++) {
      sprintf(stbuf, “[%d]”, i);
      sc_trace(tf, v[i], NAME + stbuf);
   }
}
```

This trace function has one additional argument, which is the length of the array to be traced.

SystemC has predefined trace functions for all SystemC defined vector types (`sc_int<>`, `sc_uinit<>`, `sc_biginit<>`, `sc_bigunit<>`, `sc_lv<>`, and so forth).

## *Debugging SystemC*

Because each thread or clocked-thread process generates a new thread of execution, debugging the simulation can be more difficult than with a typical linearly executed C++ program. The execution threads in the simulation means the simulation proceeds in a nonlinear fashion. It may be difficult to determine the code that will be executed next.

You may want to debug only your code, not the SystemC class libraries. The easiest way to debug a design is to place a breakpoint at the beginning of a process that you are interested in debugging. When the simulation stops at one of these breakpoints, simulation will halt and you can debug the appropriate process as required.

Appendix A

# VHDL Designer's Guide

This section will focus on helping VHDL designers learn how to write different types of models in SystemC. This section will present several complete models in SystemC and VHDL so that the VHDL designer can compare and contrast these models and learn how to write better SystemC models.

## DFF Examples

D flip flops are one of the basic building blocks of RTL design. Here are a few examples of some VHDL D flip flops and the corresponding SystemC models for comparison.

### Synchronous D Flip Flop

Here is the VHDL Model for a standard RTL D flip flop.

```
library ieee;
use ieee.std_logic_1164.all;
entity dff is
  port(clock : in std_logic;
       din   : in std_logic;
       dout  : out std_logic);
```

```
end dff;

architecture rtl of dff is
begin
  process
  begin
    wait until clock'event and clock = '1';
    dout <= din;
  end process;
end rtl;
```

Here is a corresponding SystemC model:

### SystemC Implementation

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

SC_MODULE(dff) {
  sc_in<bool>  din;
  sc_in<bool>  clock;
  sc_out<bool> dout;

  void doit() {
    dout = din;
  };

  SC_CTOR(dff) {
    SC_METHOD(doit);
    sensitive_pos << clock;
  }
};
```

### D Flip Flop with Asynchronous Reset

One of the most common flip flops used in designs is the Dff with asynchronous reset. These flip flops help the designer get a design to start at a known state easily.

By providing an active reset signal at design power up the designer can reset the flip flops of the design to a known state.

Here is the VHDL for a D flip flop with an asynchronous reset input.

```
library ieee;
use ieee.std_logic_1164.all;
entity dffa is
  port( clock : in std_logic;
        reset : in std_logic;
        din   : in std_logic;
        dout  : out std_logic);
end dffa;

architecture rtl of dffa is
begin
  process(reset, clock)
  begin
    if reset = '1' then
      dout <= '0';
    elsif clock'event and clock = '1' then
      dout <= din;
    end if;
  end process;
end rtl;
```

The SystemC model looks similar to the normal D flip flop discussed in the last section, but now has the reset signal in the process sensitivity list. Positive edges on the clock input or changes in value of the reset signal will cause process do_ffa to activate.

The process first checks the value of reset. If reset is equal to 1 the flip flop output is set to 0. If reset is not active the process will look for a positive edge on input clock. This is accomplished by using the event() method on the clock input port. This method works just like the 'event method in VHDL. It will be true if an event has just occurred on input clock.

Here is the corresponding SystemC implementation.

```
// dffa.h
```

```
#include "systemc.h"

SC_MODULE(dffa) {
  sc_in<bool>  clock;
  sc_in<bool>  reset;
  sc_in<bool>  din;
  sc_out<bool> dout;

  void do_ffa(){
    if (reset) {
      dout = false;
    } else if (clock.event()) {
        dout = din;
    }
  };

 SC_CTOR(dffa) {
    SC_METHOD(do_ffa);
    sensitive(reset);
    sensitive_pos(clock);
  }
};
```

## *Shifter*

The next few examples add more complexity. This module implements a very basic 8 bit shifter block. the shifter can be loaded with a new value by placing a value on input din, setting input load to 1, and causing a positive edge to occur on input clk. The shifter will shift the data left or right depending on the value of input LR. If LR equals 0 the shifter will shift its contents right by 1 bit. If LR equals 1 the shifter will shift its contents left by 1 bit.

Here is the VHDL description:

```
library ieee;
use ieee.std_logic_1164.all;
entity shift is
```

```
  port( din   : in std_logic_vector(7 downto 0);
        clk   : in std_logic;
        load  : in std_logic;
        LR    : in std_logic;
        dout  : inout std_logic_vector(7 downto 0));
end shift;

architecture rtl of shift is
  signal shiftval : std_logic_vector(7 downto 0);
begin
  nxt: process(load, LR, din, dout)
  begin
    if load = '1' then
      shiftval <= din;

    elsif LR = '0' then
      shiftval(6 downto 0) <= dout(7 downto 1);
      shiftval(7) <= '0';

    elsif LR = '1' then
      shiftval(7 downto 1) <= dout(6 downto 0);
      shiftval(0) <= '0';

    end if;
  end process;
end rtl;
```

### SystemC Implementation

The SystemC implementation of the shifter uses process shifty to perform the shifting and loading operations. This process is an SC_METHOD process sensitive only to the positive edge of input clk. A designer could use an SC_CTHREAD process for this example and the behavior would be the same. However and SC_CTHREAD process is less efficient and the simulation will run slower.

Whenever the clock has a positive edge process shifty will activate and check the value of input load. If load is 1 the current value of din is assigned to shiftval, the local value of the shifter at all times. Local value shiftval is needed because the value of output ports cannot be read. Notice that at the end of the process shiftval is assigned to dout.

If load is not active the process will check the value of input LR and perform the appropriate action based on the value of LR. To perform the actual shifting operation notice that process shifty uses the range() method.

Here is the SystemC implementation:

```
// shift.h

#include “systemc.h”

SC_MODULE(shift) {

  sc_in_bv<8>     din;
  sc_in<bool> clk;
  sc_in<bool> load;
  sc_in<bool> LR;
  sc_out_bv<8>    dout;

  sc_bv<8> shiftval;

  void shifty();

  SC_CTOR(shift) {
    SC_METHOD(shifty);
    sensitive_pos (clk);
  }
};

// shift.cc
#include “shift.h”

void shift::shifty {

  if ( load == ‘1’) {
    shiftval = din;

  } else if (LR == ‘0’) {
    shiftval.range(0,6) = shiftval.range(1,7);
    shiftval[7] = ‘0’;

  } else if (LR == ‘1’) {
```

```
    shiftval.range(1,7) = shiftval.range(0,6);
    shiftval[0] = '0';
  }
  dout = shiftval;
}
```

## *Counter*

The next example is an 8 bit counter. This counter can be set to a value by setting the value of input load to 1 and placing the value to load on input din. The counter can be cleared by setting input clear to a 1. Below is the VHDL implementation.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity counter is
  port( clock : in std_logic;
        load  : in std_logic;
        clear : in std_logic;
        din   : in std_logic_vector(7 downto 0);
        dout  : inout std_logic_vector(7 downto 0));
end counter;

architecture rtl of counter is
  signal countval : std_logic_vector(7 downto 0);
begin
  process(load, clear, din, dout)
  begin
    if clear = '1' then
      countval <= "00000000";

    elsif load = '1' then
      countval <= din;

    else
      countval <= dout + "00000001";
    end if;

  end process;
```

```
  process
  begin
    wait until clock'event and clock = '1';
    dout <= countval;
  end process;

end rtl;
```

### SystemC Implementation

Here is the SystemC implementation of the counter. Input ports clock, load, and clear are of type bool. Ports din and dout are 8 bit vector ports. Internally an int named countval is used to hold the value of the counter. When clear is a 1 countval is set to 0. When load is a 1 countval is set to the value on port din. Notice the read() method used when the port is read. This method is used because an implicit type conversion is happening when din is assigned to countval. This method helps SystemC determine the type of the port easier so that the correct conversion function can be called.

```
// counter.h

#include "systemc.h"

SC_MODULE(counter) {
  sc_in<bool>          clock;
  sc_in<bool>          load;
  sc_in<bool>          clear;
  sc_in<sc_int<8> >    din;
  sc_out<sc_int<8> >   dout;

  int countval;

  void onetwothree ();

  SC_CTOR(counter) {
    SC_METHOD(onetwothree);
    sensitive_pos (clock);
  }
};
```

```
// counter.cc

#include "counter.h"

void counter::onetwothree () {
  if (clear == '1') {
    countval = 0;

  } else if (load == '1') {
    countval = din.read(); // use read when a type
                           // conversion is happening
                           // from an input port

  } else {
    countval++;
  }
  dout = countval;
}
```

## *State Machine*

The next example is a state machine. This example represents a state machine within a voicemail controller. The state machine will start in the main state and then transition to a send state or review state depending on user inputs. From the review or send states the user can go to other states such as repeat, erase, record, etc. Output signals play, recrd, erase, save and address are triggered as each of these states are entered thereby controlling the voicemail system.
Here is the VHDL implementation:

```
package vm_pack is
  type t_vm_state is (main_st, review_st, repeat_st,
                      save_st, erase_st, send_st,
                      address_st, record_st,
                      begin_rec_st, message_st);
  type t_key is (‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’,
                 ‘7’, ‘8’, ‘9’, ‘*’, ‘#’);
```

```
end vm_pack;

use work.vm_pack.all;
library ieee;
use ieee.std_logic_1164.all;
entity stmach is
  port( clk : in std_logic;
        key : in t_key;
        play, recrd, erase, save,
        address : out std_logic);
end stmach;

architecture rtl of stmach is
  signal next_state, current_state : t_vm_state;
begin
  process(current_state, key)
  begin

  play <= '0';
  save <= '0';
  erase <= '0';
  recrd <= '0';
  address <= '0';

  case current_state is
    when main_st =>
        if key = '1' then
          next_state <= review_st;
        elsif key = '2' then
          next_state <= send_st;
        else
          next_state <= main_st;
        end if;

      when review_st =>
        if key = '1' then
          next_state <= repeat_st;
        elsif key = '2' then
          next_state <= save_st;
        elsif key = '3' then
          next_state <= erase_st;
```

```
  elsif key = '#' then
    next_state <= main_st;
  else
    next_state <= review_st;
  end if;

when repeat_st =>
  play <= '1';
  next_state <= review_st;

when save_st =>
  save <= '1';
  next_state <= review_st;

when erase_st =>
  erase <= '1';
  next_state <= review_st;

when send_st =>
  next_state <= address_st;

when address_st =>
  address <= '1';
  if key = '#' then
    next_state <= record_st;
  else
    next_state <= address_st;
  end if;

when record_st =>
  if key = '5' then
    next_state <= begin_rec_st;
  else
    next_state <= record_st;
  end if;

when begin_rec_st =>
  recrd <= '1';
  next_state <= message_st;

when message_st =>
```

```
        recrd <= '1';
        if key = '#' then
          next_state <= send_st;
        else
          next_state <= message_st;
        end if;
    end case;
  end process;

  process
  begin
    wait until clk'event and clk = '1';
    current_state <= next_state;
  end process;

end rtl;
```

### SystemC State Machine

The SystemC implementation uses two enum types to represent the state of the state machine and the state of the key values passed to the state machine. The state machine implementation consists of two SC_METHOD processes. SC_METHOD processes are by far the most efficient processes and should be used where possible. Process getnextst calculates the new state of the state machine based on the current state and the input values. Process setstate copies the calculated next_state to the current_state every positive clock edge on input clk.

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

enum vm_state (main_st, review_st, repeat_st, save_st,
               erase_st, send_st, address_st,
               record_st, begin_rec_st, message_st);

enum key_t ('0', '1', '2', '3', '4', '5', '6', '7',
            '8', '9', '*', '#');

SC_MODULE(stmach) {
```

```
  sc_in<bool>   clk;
  sc_in<key_t>      key;
  sc_out<sc_logic>  play;
  sc_out<sc_logic>  recrd;
  sc_out<sc_logic>  erase;
  sc_out<sc_logic>  save;
  sc_out<sc_logic>  address;

  sc_signal<vm_state> next_state;
  sc_signal<vm_state> current_state

  void getnextst();
  void setstate();

  SC_CTOR(stmach) {
    SC_METHOD(getnextst);
    sensitive << key << current_state;

    SC_METHOD(setstate);
    sensitive_pos (clk);
  }
};

// statmach.cc
#include “stmach.h”

void getnextst() {
  play = ‘0’;
  recrd = ‘0’;
  erase = ‘0’;
  save = ‘0’;
  address = ‘0’;

  switch (current_state) {

    case main_st:
      if (key == ‘1’) {
        next_state = review_st;
      } else {
        if (key == 2) {
          next_state = send_st;
```

```
    } else {
      next_state = main_st;
    }
  }

case review_st:
  if (key == '1') {
    next_state = repeat_st;
  } else {
     if (key == '2') {
       next_state = save_st;
     } else {
       if (key == '3') {
         next_state = erase_st;
       } else {
         if (key == '#') {
           next_state = main_st;
         } else {
           next_state = review_st;
         }
       }
     }
   }

case repeat_st:
  play = '1';
  next_state = review_st;

case save_st:
  save = '1';
  next_state = review_st;

case erase_st:
  erase = '1';
  next_state = review_st;

case send_st:
  next_state = address_st;

case address_st:
  address = '1';
```

```
      if (key == '#') {
        next_state = record_st;
      } else {
        next_state = address_st;
      }

    case record_st:
      if (key == '5') {
        next_state = begin_rec_st;
      } else {
        next_state = record_st;
      }

    case begin_rec_st:
      recrd = '1';
      next_state = message_st;

    case message_st:
      recrd = '1';
      if (key == '#') {
        next_state = send_st;
      } else {
        next_state = message_st;
      }
    } // end switch
  } // end method

void setstate() {
  current_state = next_state;
}
```

## *Memory*

The last module is a very simple memory model. The memory device has an enable port to activate the device, and a readwr port to determine whether or not the device is being written to or read from. The memory module has a single data inout bus that either delivers the addressed item, or accepts data to write to a location.When the enable input is 0, the output of the ram device will be all 'Z' (hi impedance) and no read or write operations can be performed. To read a location set enable to '1',

readwr to ‘0’, and apply the appropriate address. To write a location set enable to ‘1’, readwr to ‘1’, addr to the appropriate location to write, and data to the data value to write.

The model is implemented in VHDL with a single process so that a variable can be used to store the memory data. Notice that the SystemC implementation uses two processes, one for read and one for write.

Here is the VHDL model:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ram is
  port(enable : in std_logic;
       readwr : in std_logic;
       addr   : in std_logic_vector(7 downto 0);
       data   : inout std_logic_vector(15 downto 0)
      );
end ram;

architecture rtl of ram is
begin
  process(addr, enable, readwr)
    subtype data16 is std_logic_vector(15 downto 0);
    type ramtype is array(0 to 255) of data16;
    variable ramdata : ramtype;
  begin
    if (enable = ‘1’) then
      if readwr = ‘0’ then
        data <= ramdata(conv_integer(addr));
      elsif readwr = ‘1’ then
        ramdata(conv_integer(addr)) := data;
      end if;
    else
      data <= “ZZZZZZZZZZZZZZZZ”;
    end if;
  end process;
end rtl;
```

### SystemC Implementation

The SystemC implementation has similar port types to the VHDL model, but optimized for SystemC. Notice that addr is an sc_int of 8 bits. This is the most efficient implementation for object of less than 64 bits. Also notice that port data is an sc_inout_rv type. The port needs to be inout, and needs the ability to tristate the output. A resolved vector type will allow the output to tristate and still be able to connect to tristate busses.

The ram module contains two SC_METHOD processes. One for reading the ram and one for writing the ram. Notice that the process that writes the ram also has to be sensitive to changes on input port data so that the proper value gets written into the ram.

```
// ram.h
SC_MODULE(ram) {
  sc_in<sc_int<8> >   addr;
    sc_in<bool>    enable;
    sc_in<bool>    readwr;
    sc_inout_rv<16>    data;

    void read_data();
    void write_data();

    sc_int ram_data[256];

    SC_CTOR(ram) {
      SC_METHOD(read_data);
      sensitive << addr << enable << readwr;

      SC_METHOD(write_data);
      sensitive << addr << enable << readwr << data;
    }
};

// ram.cc
#include “ram.h”

void ram::read_data () {
  if (enable == ‘1’) && (readwr == ‘0’) {
    data = ramdata[addr];
  } else {
```

```
    data = “ZZZZZZZZZZZZZZZZ”;
  }
}

void ram::write_data () {
  if (enable == ‘1’) && (readwr == ‘1’) {
    ramdata[addr] = data;
  }
}
```

Appendix B

# *Verilog Designers' Guide*

This section is for Verilog designers wanting to learn how to write good SystemC models. This section will present a number of Verilog models and then the SystemC models for the same design. The Verilog designer can then compare and contrast the models to get a better understanding of how to write SystemC models.

## *DFF Examples*

D flip flops are one of the basic building blocks of RTL design. Here are a few examples of some Verilog D flip flops and the corresponding SystemC models for comparison.

### Synchronous D Flip Flop

Here is the Verilog model for a standard RTL D flip flop

```
module dff(din, clock, dout);
input din;
input clock;
output dout;

reg dout;
```

```
always @(posedge clock)
  dout <= din;

endmodule
```

**SystemC Implementation**

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

SC_MODULE(dff) {
  sc_in<bool>  din;
  sc_in<bool>  clock;
  sc_out<bool> dout;

  void doit(){
    dout = din;
  };

  SC_CTOR(dff) {
    SC_METHOD(doit);
    sensitive_pos << clock;
  }
};
```

## *Asynchronous Reset D Flip Flop*

### D Flip Flop with Asynchronous Reset

One of the most common flip flops used in designs is the Dff with asynchronous reset. These flip flops help the designer get a design to start at a known state easily. By providing an active reset signal at design power up the designer can reset the flip flops of the design to a known state.

Here is the Verilog description for a D flip flop with an asynchronous reset input.

```
module dffa(clock, reset, din, dout);
input clock, reset, din;
output dout;

reg dout;

always @(posedge clock or reset)
begin
  if (reset)
    dout <= 1'b0;
  else
    dout = din;
end
endmodule
```

### SystemC Implementation

The SystemC model looks similar to the normal D flip flop discussed in the last section, but now has the reset signal in the process sensitivity list. Positive edges on the clock input or changes in value of the reset signal will cause process do_ffa to activate.

The process first checks the value of reset. If reset is equal to 1 the flip flop output is set to 0. If reset is not active the process will look for a positive edge on input clock. This is accomplished by using the event() method on the clock input port. This method works just like the ‘event method in VHDL. It will be true if an event has just occurred on input clock.

Here is the corresponding SystemC implementation.

```
// dffa.h

#include "systemc.h"

SC_MODULE(dffa) {
  sc_in<bool>  clock;
  sc_in<bool>  reset;
  sc_in<bool>  din;
```

```
  sc_out<bool> dout;

  void do_ffa() {
    if (reset) {
      dout = false;
    } else if (clock.event()) {
      dout = din;
    }
  };

 SC_CTOR(dffa) {
    SC_METHOD(do_ffa);
    sensitive(reset);
    sensitive_pos(clock);
  }
};
```

## *Shifter*

The next few examples add more complexity. This module implements a very basic 8 bit shifter block. the shifter can be loaded with a new value by placing a value on input din, setting input load to 1, and causing a positive edge to occur on input clk. The shifter will shift the data left or right depending on the value of input LR. If LR equals 0 the shifter will shift its contents right by 1 bit. If LR equals 1 the shifter will shift its contents left by 1 bit.

Here is the Verilog description:

```
module shift(din, clk, load, LR, dout);
input [0:7] din;
input clk, load, LR;
output [0:7] dout;

wire [0:7] dout;
reg [0:7] shiftval;

assign dout = shiftval;
always @(posedge clk)
```

```
begin
  if (load)
    shiftval = din;
  else if (LR)
    begin
      shiftval[0:6] = shiftval[1:7];
      shiftval[7] = 1'b0;
    end
  else
    begin
      shiftval[1:7] = shiftval[0:6];
      shiftval[0] = 1'b0;
    end
end
endmodule
```

### SystemC Implementation

The SystemC implementation of the shifter uses process shifty to perform the shifting and loading operations. This process is an SC_METHOD process sensitive only to the positive edge of input clk. A designer could use an SC_CTHREAD process for this example and the behavior would be the same. However and SC_CTHREAD process is less efficient and the simulation will run slower.

Whenever the clock has a positive edge process shifty will activate and check the value of input load. If load is 1 the current value of din is assigned to shiftval, the local value of the shifter at all times. Local value shiftval is needed because the value of output ports cannot be read. Notice that at the end of the process shiftval is assigned to dout.

If load is not active the process will check the value of input LR and perform the appropriate action based on the value of LR. To perform the actual shifting operation notice that process shifty uses the range() method.

Here is the SystemC implementation:

```
// shift.h

#include "systemc.h"
```

```
SC_MODULE(shift) {

  sc_in_bv<8>     din;
  sc_in<bool> clk;
  sc_in<bool> load;
  sc_in<bool> LR;
  sc_out_bv<8>    dout;

  sc_bv<8> shiftval;

  void shifty();

  SC_CTOR(shift) {
    SC_METHOD(shifty);
    sensitive_pos (clk);
  }
};

// shift.cc
#include “shift.h”

void shift::shifty {

  if ( load == ‘1’) {
    shiftval = din;

  } else if (LR == ‘0’) {
    shiftval.range(0,6) = shiftval.range(1,7);
    shiftval[7] = ‘0’;

  } else if (LR == ‘1’) {
    shiftval.range(1,7) = shiftval.range(0,6);
    shiftval[0] = ‘0’;
  }
  dout = shiftval;
}
```

## *Counter*

The next example is an 8 bit counter. This counter can be set to a value by setting the value of input load to 1 and placing the value to load on input din. The counter can be cleared by setting input clear to a 1. Below is the Verilog implementation.

```
module counter(clock, load, clear, din, dout);
input clock, load, clear;
input [0:7] din;
output [0:7] dout;

wire [0:7] dout;
reg [0:7] countval;

assign dout = countval;

always @(posedge clock)
begin
  if (clear)
    countval = 0;
  else if (load)
    countval = din;
  else
    countval = countval + 1;
end
endmodule
```

### SystemC Implementation

Here is the SystemC implementation of the counter. Input ports clock, load, and clear are of type bool. Ports din and dout are 8 bit vector ports. Internally an int named countval is used to hold the value of the counter. When clear is a 1 countval is set to 0. When load is a 1 countval is set to the value on port din. Notice the read() method used when the port is read. This method is used because an implicit type conversion is happening when din is assigned to countval. This method helps SystemC determine the type of the port easier so that the correct conversion function can be called.

```
// counter.h
```

```
#include "systemc.h"

SC_MODULE(counter) {
  sc_in<bool>          clock;
  sc_in<bool>          load;
  sc_in<bool>          clear;
  sc_in<sc_int<8> >    din;
  sc_out<sc_int<8> >   dout;

  int countval;

  void onetwothree ();

  SC_CTOR(counter) {
    SC_METHOD(onetwothree);
    sensitive_pos (clock);
  }
};

// counter.cc

#include "counter.h"

void counter::onetwothree () {
  if (clear == '1') {
    countval = 0;

  } else if (load == '1') {
    countval = din.read(); // use read when a type
                           // conversion is happening
                           // from an input port

  } else {
    countval++;
  }
  dout = countval;
}
```

## *State Machine*

The next example is a state machine. This example represents a state machine within a voicemail controller. The state machine will start in the main state and then transition to a send state or review state depending on user inputs. From the review or send states the user can go to other states such as repeat, erase, record, etc. Output signals play, recrd, erase, save and address are triggered as each of these states are entered thereby controlling the voicemail system.

Here is the Verilog implementation:

```
// def.v
parameter main_st      = 4'b0000,
          review_st    = 4'b0001,
          repeat_st    = 4'b0010,
          save_st      = 4'b0011,
          erase_st     = 4'b0100,
          send_st      = 4'b0101,
          address_st   = 4'b0110,
          record_st    = 4'b0111,
          begin_rec_st = 4'b1000,
          message_st   = 4'b1001;

parameter zero   = 4'b0000,
          one    = 4'b0001,
          two    = 4'b0010,
          three  = 4'b0011,
          four   = 4'b0100,
          five   = 4'b0101,
          six    = 4'b0110,
          seven  = 4'b0111,
          eight  = 4'b1000,
          nine   = 4'b1001,
          star   = 4'b1010,
          pound  = 4'b1011;

// statemach.v
module stmach(clk, key, play, recrd, erase, save,
              address);

'include "def.v"
```

```
input clk;
input [0:3] key;
output play, recrd, erase, save, address;

reg [0:3] next_state;
reg [0:3] current_state;
reg play, recrd, erase, save, address;

always @(posedge clk)
  current_state = next_state;

always @(key or current_state)
begin
  play = 1'b0;
  recrd = 1'b0;
  erase = 1'b0;
  save = 1'b0;
  address = 1'b0;

  case (current_state)
    main_st : begin
     if (key == one)
        next_state = review_st;
      else if (key == two)
          next_state = send_st;
        else
          next_state = main_st;
    end
    review_st:begin
      if (key == one)
        next_state = repeat_st;
      else if (key == two)
           next_state = save_st;
         else if (key == three)
             next_state = erase_st;
           else if (key == pound)
               next_state = main_st;
             else
               next_state = review_st;
    end
    repeat_st: begin
```

```
      play = 1'b1;
      next_state = review_st;
    end
    save_st:begin
      save = 1'b1;
      next_state = review_st;
    end
    erase_st:begin
      erase = 1'b1;
      next_state = review_st;
    end
    send_st:begin
      next_state = address_st;
    end
    address_st:begin
      address = 1'b1;
      if (key == pound)
        next_state = record_st;
      else
        next_state = address_st;
    end
    record_st: begin
      if (key == five)
        next_state = begin_rec_st;
      else
        next_state = record_st;
    end
    begin_rec_st: begin
      recrd = 1'b1;
      next_state = message_st;
    end
    message_st: begin
      recrd = 1'b1;
      if (key == pound)
        next_state = send_st;
      else
        next_state = message_st;
      end
    endcase
  end
endmodule
```

### SystemC State Machine

The SystemC implementation uses two enum types to represent the state of the state machine and the state of the key values passed to the state machine. The state machine implementation consists of two SC_METHOD processes. SC_METHOD processes are by far the most efficient processes and should be used where possible. Process getnextst calculates the new state of the state machine based on the current state and the input values. Process setstate copies the calculated next_state to the current_state every positive clock edge on input clk.

```
// statemach.h
#include “systemc.h”

enum vm_state (main_st, review_st, repeat_st, save_st,
               erase_st, send_st, address_st,
               record_st, begin_rec_st, message_st);

enum key_t (‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’,
            ‘8’, ‘9’, ‘*’, ‘#’);

SC_MODULE(stmach) {
  sc_in<bool>   clk;
  sc_in<key_t>      key;
  sc_out<sc_logic>  play;
  sc_out<sc_logic>  recrd;
  sc_out<sc_logic>  erase;
  sc_out<sc_logic>  save;
  sc_out<sc_logic>  address;

  sc_signal<vm_state> next_state;
  sc_signal<vm_state> current_state

  void getnextst();
  void setstate();

  SC_CTOR(stmach) {
    SC_METHOD(getnextst);
    sensitive << key << current_state;
```

```
    SC_METHOD(setstate);
    sensitive_pos (clk);
  }
};

// statmach.cc
#include “stmach.h”

void getnextst() {
  play = ‘0’;
  recrd = ‘0’;
  erase = ‘0’;
  save = ‘0’;
  address = ‘0’;

  switch (current_state) {

    case main_st:
      if (key == ‘1’) {
        next_state = review_st;
      } else {
        if (key == 2) {
          next_state = send_st;
        } else {
          next_state = main_st;
        }
      }

    case review_st:
      if (key == ‘1’) {
        next_state = repeat_st;
      } else {
         if (key == ‘2’) {
           next_state = save_st;
         } else {
           if (key == ‘3’) {
             next_state = erase_st;
           } else {
             if (key == ‘#’) {
               next_state = main_st;
```

```
          } else {
            next_state = review_st;
          }
        }
      }
    }

case repeat_st:
  play = '1';
  next_state = review_st;

case save_st:
  save = '1';
  next_state = review_st;

case erase_st:
  erase = '1';
  next_state = review_st;

case send_st:
  next_state = address_st;

case address_st:
  address = '1';
  if (key == '#') {
    next_state = record_st;
  } else {
    next_state = address_st;
  }

case record_st:
  if (key == '5') {
    next_state = begin_rec_st;
  } else {
    next_state = record_st;
  }

case begin_rec_st:
  recrd = '1';
  next_state = message_st;
```

```
    case message_st:
      recrd = '1';
      if (key == '#') {
        next_state = send_st;
      } else {
        next_state = message_st;
      }
    } // end switch
  } // end method

void setstate() {
  current_state = next_state;
}
```

## *Memory*

The last module is a very simple memory model. The memory device has an enable port to activate the device, and a readwr port to determine whether or not the device is being written to or read from. The memory module has a single data inout bus that either delivers the addressed item, or accepts data to write to a location.When the enable input is 0, the output of the ram device will be all 'Z' (hi impedance) and no read or write operations can be performed. To read a location set enable to '1', readwr to '0', and apply the appropriate address. To write a location set enable to '1', readwr to '1', addr to the appropriate location to write, and data to the data value to write.

Here is the Verilog model:

```
module ram(addr, enable, readwr, data);
input [0:7] addr;
input enable, readwr;
inout [0:15] data;

reg [0:15] ram_data [0:255];

assign data = (enable & !readwr) ?
     ramdata[addr] : 16'bz;
```

```
always @(addr or enable or readwr or data)
begin
  if (enable & readwr)
    ramdata[addr] = data;
end
endmodule
```

### SystemC Implementation

The SystemC implementation has similar port types to the VHDL model, but optimized for SystemC. Notice that addr is an sc_int of 8 bits. This is the most efficient implementation for object of less than 64 bits. Also notice that port data is an sc_inout_rv type. The port needs to be inout, and needs the ability to tristate the output. A resolved vector type will allow the output to tristate and still be able to connect to tristate busses.

The ram module contains two SC_METHOD processes. One for reading the ram and one for writing the ram. Notice that the process that writes the ram also has to be sensitive to changes on input port data so that the proper value gets written into the ram.

```
// ram.h
SC_MODULE(ram) {
  sc_in<sc_int<8> >   addr;
    sc_in<bool>    enable;
    sc_in<bool>    readwr;
    sc_inout_rv<16>    data;

    void read_data();
    void write_data();

    sc_int ram_data[256];

    SC_CTOR(ram) {
      SC_METHOD(read_data);
      sensitive << addr << enable << readwr;

      SC_METHOD(write_data);
      sensitive << addr << enable << readwr << data;
    }
```

```
};

// ram.cc
#include “ram.h”

void ram::read_data () {
  if (enable == ‘1’) && (readwr == ‘0’) {
    data = ramdata[addr];
  } else {
    data = “ZZZZZZZZZZZZZZZZ”;
  }
}

void ram::write_data () {
  if (enable == ‘1’) && (readwr == ‘1’) {
    ramdata[addr] = data;
  }
}
```

# *Index*