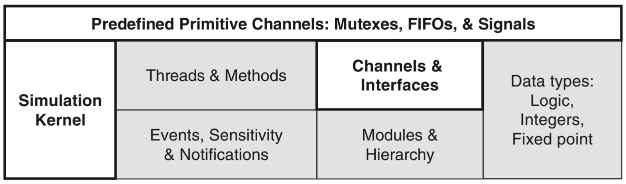
Chapter 9 Evaluate-Update Channels



Predefined Primitive Channels: Mutexes, FIFOs, & Signals
Simulation Kernel
Threads & Methods
Channels & Interfaces
Data types: Logic, Integers, Fixed point
Events, Sensitivity & Notifications
Modules & Hierarchy

SC_SIGNALS

The preceding chapter considered high-level synchronization mechanisms. This chapter delves into electronic hardware[1].

Electronic signals behave in a manner approaching instantaneous activity. Generally, electronic signals have a single source (producer), but multiple sinks (consumer). It is quite important that all sinks "see" a signal update at the same time.

The easiest way to understand this concept is to consider the common hardware shift register. This model has a number of registers or memory elements as indicated in the diagram (Fig. 9.1) below.

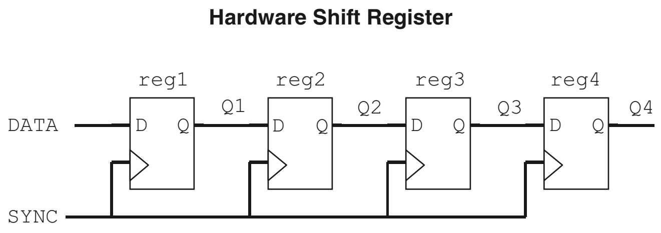


Fig. 9.1 Shift register

Data moves from left to right synchronous to the clock labeled `SYNC`. In software (e.g., C/C++), this flow would be modeled with four ordinary assignments (Fig. 9.2):

```
Q4 = Q3;
Q3 = Q2;
Q2 = Q1;
Q1 = DATA;
```

Fig. 9.2 Example of modeling a software-only shift register

[1]It is unclear whether the concepts discussed here have any application outside of electrical signals.

DOI 10.1007/978-0-387-69958-5_9, © Springer Science+Business Media, LLC 2010

For this register to work, ordering is very important. In hardware, things are more difficult. Each register (reg1…reg4) is an independent concurrent process. Recall that the simulator places no order requirements on the processes. Below (Fig. 9.3) is a diagram from Chapter 6 Concurrency. Consider each process to represent a register from the preceding design.

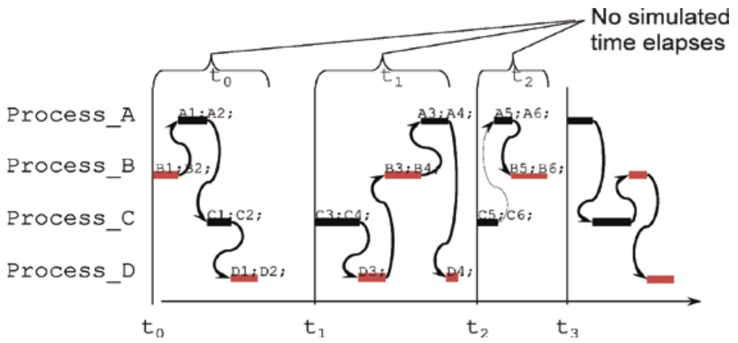


Fig. 9.3 Simulated activity with four concurrent processes

Since there is no guarantee that one assignment will take place before the other, we need to find some other solution. One idea might be to use events to force an ordering. This design would have `Process_A` wait for an event from `Process_B` before assigning its register, `Process_B` wait for an event from `Process_C` before assigning its register, and so on. This design requires coding both a **`wait`** and **`notify`** for each register, and it can become quite tiresome.

Another solution involves representing each register as a two-deep FIFO. This approach seems unnecessarily complex requiring two storage locations for the data and two pointers/counters to manage the state of the FIFO.

9.1 Completed Simulation Engine

To solve this problem, simulators have a feature known as the evaluate-update paradigm. The next diagram (Fig. 9.4) is the complete SystemC simulation kernel with the added update state. It is possible to go from evaluate to update and back. This cycle is known as the delta-cycle. Even when the simulator moves from evaluate to update to advance time, we say that at least one delta-cycle has occurred. Let's see how the delta-cycle is used.

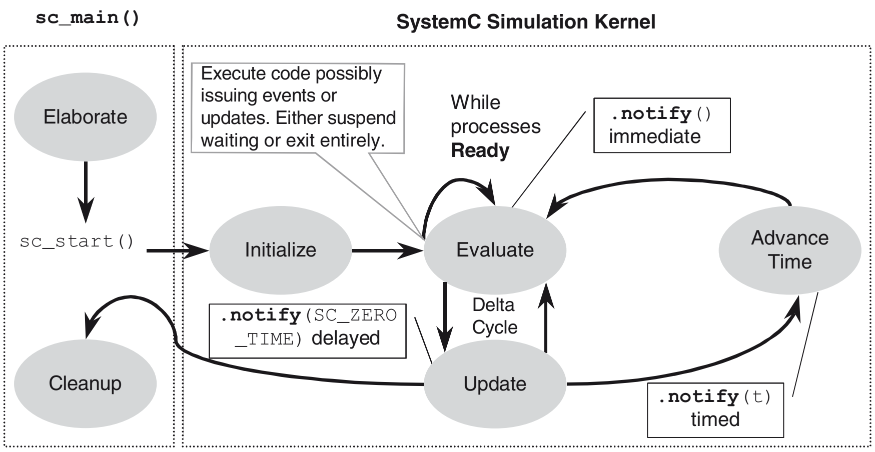

sc_main()
SystemC Simulation Kernel
Elaborate
sc_start()
Cleanup
Execute code possibly issuing events or updates. Either suspend waiting or exit entirely.
While processes Ready
.notify() immediate
Initialize
Evaluate
Advance Time
Delta Cycle
.notify(SC_ZERO_TIME) delayed
Update
.notify(t) timed

Fig. 9.4 Full SystemC simulation engine

Referring back to Chapter 6 on Concurrency, you'll recall that we discussed the notions of immediate notification, delayed notification, and timed notification. In the preceding diagram (Fig. 9.4) you will notice we've annotated the state bubble to indicate where each of these event notifications actually occur. It is important to realize that in the case of **notify(SC_ZERO_TIME)**, the notification occurs in the update phase after evaluation has completed. This means that processes waiting on events notified in this manner, will all see the event at the same instant. This contrasts starkly with immediate notification that uses the **notify(void)** syntax. Immediate notification may cause some processes to miss the event if they are not already waiting for it. These processes may miss the event because they have not run in the current evaluate phase.

Special channels, known as signal channels, use this update phase as a point of data synchronization. To accomplish this synchronization, every channel has two storage locations: the current value and the new value. Visually, there are two sets of data: new and current.

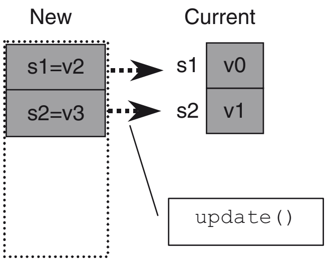


Fig. 9.5 Signal channel data storage

Referring to the preceding figure (Fig. 9.5), when a process writes to a signal channel, the process stores into the new value rather than into the current value. The process then calls **request_update()** to notify the simulation kernel. When the update phase occurs, the simulation kernel calls the **update()** methods of all channels that requested update during the preceding evaluate phase.

The **update()** method may do more than simply copy the new value into the current value. It may resolve contentions and, most importantly, notify **sc_event**'s (e.g., to indicate a change and wake up a process in the waiting state).

An important aspect of this paradigm is the current value remains unchanged until the end of the update phase. If a process writes to an evaluate-update channel and then immediately (i.e., without suspending) accesses the channel, it will find the current value unchanged.

This behavior is a frequent cause for confusion for simulation neophytes. Those familiar with HDLs should not be surprised. Shortly, we will discuss techniques to make this behavior less of a surprise.

Another consideration is that the new value will contain only the last value written to a signal channel during the evaluate phase. Thus, writing repeatedly overwrites the previous new value.

9.2 SystemC Signal Channels

The **sc_signal**<*T*> primitive channel and its close relative, **sc_buffer**<*T*>, both use the evaluate-update paradigm. Here (Fig. 9.6) is the syntax for declaration, reading, and writing:

The **sc_signal::write()** method contains the evaluate phase portion of the evaluate-update behavior. The **write()** method includes a call to the protected **sc_prim_channel::request_update()** method. The call to **sc_signal::update()** is hidden, and the call occurs during the update phase when the kernel calls it as a result of the **request_update()**.

```
  sc_signal<datatype> signame[, signamei]…;//define
…
  signame.write(newvalue);
  varname  = signame.read();
  wait(signame.value_changed_event()|...);
  wait(signame.default_event()|...);
  if (signame.event() == true) {
    // occurred in previous delta-cycle
```

Fig. 9.6 Syntax of **sc_signal**<*T*>

The **sc_signal::read()** method returns the current value. If the signal has been written to in the current evaluate phase, the new value is *not* reflected in the returned value. This behavior may come as a surprise to some folks (especially to non-hardware background folks).

The **sc_signal::value_changed_event()** method returns a reference to an sc_event. This event is notified any time the update causes a change in value. This behavior lets code wait on changes in the channel.

The **sc_signal::default_event()** method is simply an alias for the **via** method. This method is used by the **sensitive** class to allow use of the simplified syntax shown (Fig. 9.7) below (recall that this aliasing must be done during elaboration, normally in the constructor). Several other channels include this same aliasing to an event referencing. In fact, **`sc_event_queue`** returns this same method to allow it to be used in static sensitivity.

```
sensitive << signame;
```

Fig. 9.7 Static sensitivity to `sc_signal<T>`

The **`event()`** method is special. Normally, it is impossible to determine which event caused a return from **`wait()`**; however, for **`sc_signal`** channels (including other derivatives mentioned in this chapter), the **`event()`** method may be called to see if the channel issued an event in the immediately previous delta-cycle. This ability to determine which event caused the last return from **`wait()`** does not preclude the occurrence of other events in the previous cycle.

It should be noted that **`sc_signal`**<*T*> is essentially identical to VHDL's **`signal`**. For Verilog, the analogy is a **`reg`** that uses the Verilog non-blocking assignment operator exclusively.

```
// Declare variables
int                 count;
string              message_temp;
sc_signal<int>      count_sig;
sc_signal<string>   message_sig;

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

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

cout << "3rd delta cycle" << endl;
message_sig.write(message_temp = "Rev engines");
cout << "count is " << count << ", "
     << "count_sig is " << count_sig << endl
     << "message_temp is '" << message_temp << "', "
     << "message_sig is '" << message_sig << "'"
     << endl << endl << "Done" << endl;
```

Fig. 9.8 Example of **`sc_signal`**<*T*>

An example of usage and the slightly surprising results[2] are in order.

The example in Fig. 9.8 produces the result shown in Fig. 9.9. Notice how the current value of the signals, `count_sig` and `message_sig`, remain unchanged until a delta-cycle has occurred. On the other hand, the non-signal values, `count` and `message_temp`, get immediately updated.

```
Initialize during 1st delta cycle
count is 11, count_sig is 0
message_temp is 'Whoa', message_sig is ''
Waiting

2nd delta cycle
count is 20, count_sig is 10
message_temp is 'Whoa', message_sig is 'Hello'
Waiting

3rd delta cycle
count is 20, count_sig is 20
message_temp is 'Rev engines', message_sig is
'Hello'

Done
```

Fig. 9.9 Example of `sc_signal<T>` output

Because the code uses a naming convention (i.e., appended `_sig` to the signals), it is relatively easy to spot the evaluate-update signals and make the mental connection to the behavior. Without the naming convention, one might wonder if the identifiers represent some other channel (e.g., `sc_fifo<T>`).

In addition to the preceding syntax, SystemC has overloaded the assignment and copy operators to allow the following *dangerous* syntaxes:

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

Fig. 9.10 Syntax of **sc_signal**<*T*> (dangerous)

The reason we consider these syntaxes dangerous relates to the issue of the evaluate-update paradigm. Consider the following example, assuming that ***r*** is an **`sc_signal<`*`int`*`>`**:

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

Fig. 9.11 Dangerous **`sc_signal`**<*T*>

[2]This usage may surprise non-HDL experienced folks. HDL-experienced users should understand the VHDL or Verilog analogy.

Without sufficient context, the casual reader would be quite surprised at the results shown below. Assume on entry x=3, y=4, r=0.

```
Radius is 0
```

Fig. 9.12 Example of `sc_signal` output (dangerous)

Even when using what might be considered the safer syntax, you must be careful. We strongly suggest that you use a naming style.

One beneficial aspect of **`sc_signal`**<*T*> and **`sc_buffer`**<*T*> channels is a restriction that only a single process may write to a given signal during a specific delta-cycle. This restriction avoids the potential danger of two processes non-deterministically asserting a value and creating a race condition.

For the OSCI simulator, the run-time error is flagged in this situation if and only if you have defined the compile-time macro **`DEBUG_SYSTEMC`**. This macro is not part of the IEEE-1666 standard, but it was defined for the OSCI implementation to reduce simulation overhead. We recommend you define this macro early in the project, and only remove it once you are certain your code doesn't violate the signal process writer rule and need more performance. You can find an example of this danger in the `danger_ex` in the downloaded examples.

9.3 Resolved Signal Channels

There are times when it is appropriate to have multiple writers. One of these situations involves modeling buses that have the possibility of high impedance (i.e., Z) and contention (i.e., X).

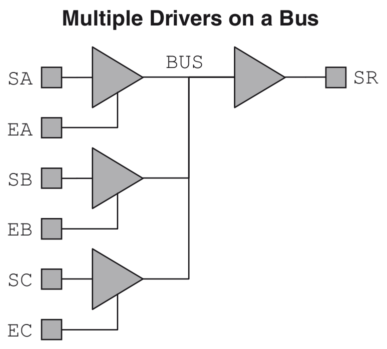


Fig. 9.13 Tri-state bus

SystemC provided the specialized channels **sc_signal_resolved** and **sc_signal_rv**<*T*> (Fig. 9.14). The _rv means resolved vector.

```
sc_signal_resolved name;
sc_signal_rv<WIDTH> name;
```

Fig. 9.14 Syntax of **sc_signal_resolved** and **sc_signal_rv**

The base functionality has identical semantics to **sc_signal**<***sc_logic***>; however, it allows for multiple writers and provides built-in resolution functionality as follows (Table 9.1):

Table 9.1 Resolution functionality for sc_signal_resolved

A\B	'0'	'1'	'X'	'Z'
'0'	'0'	'X'	'X'	'0'
'1'	'X'	'1'	'X'	'1'
'X'	'X'	'X'	'X'	'X'
'Z'	'0'	'1'	'X'	'Z'

One minor failing of SystemC is the lack of direct support for several common system-level bus concepts. Specifically, SystemC has no mechanisms for pull-ups, pull-downs, nor various open-source or open-drain variations.

For these, you have to create your own channels, which is not too difficult. The easiest way is to create a class derived from an existing class that almost works. Here is the resolution table (Table 9.2) for a pull-up functionality:

Table 9.2 Resolution functionality for eslx_pullup

A\B	'0'	'1'	'X'	'Z'
'0'	'0'	'X'	'X'	'0'
'1'	'X'	'1'	'X'	'1'
'X'	'X'	'X'	'X'	'X'
'Z'	'0'	'1'	'X'	'1'

Notice that there is only one difference in the table (shaded). The custom channel in Fig. 9.15 implements this resolution for a single-bit pull-up functionality.

```
class eslx_pullup
  : public sc_core::sc_signal_resolved {
public:
  // constructors
  eslx_pullup()
  : sc_signal_resolved(sc_gen_unique_name("pullup"))
  {}
  explicit eslx_pullup(const char* nm)
  : sc_signal_resolved(nm)
  {}
  const sc_dt::sc_logic& read() const {
    const sc_dt::sc_logic& result
      (sc_core::sc_signal_resolved::read());
    static const sc_dt::sc_logic
      ONE(sc_dt::SC_LOGIC_1);
    if (result == sc_dt::SC_LOGIC_Z) {
      return ONE;
    } else {
      return result;
    }//endif
  }
};
```

Fig. 9.15 Example of `eslx_signal_pullup`

9.4 Template Specializations of sc_signal Channels

SystemC has several template specializations that bear discussion. A template specialization occurs when a definition is provided for a specific template value. If there is more than one template variable involved, we call it a partial specialization.

For example, **`sc_signal`**<*T*> has a single template variable representing the **`typename`**. SystemC defines some additional behaviors for **`sc_signal<bool>`** that are not available for the general case. Thus, an **`sc_signal<char>`** does not support the concept of a **`posedge_event()`**.

The specialized templates **`sc_signal`**<**`bool`**> and **`sc_signal`**<**`sc_logic`**> have the following (Fig. 9.16) extensions:

```
sensitive << signame.posedge_event()
          << signame.negedge_event();
wait(signame.posedge_event()
    |signame.negedge_event());
if (signame.posedge_event()
   |signame.negedge_event()) {
```

Fig. 9.16 Syntax of specializations **posedge** and **negedge**

For **`sc_logic`**, a **`posedge_event`** occurs on any transition to **`SC_LOGIC_1`**, which includes **`SC_LOGIC_X`** and **`SC_LOGIC_Z`**. The same is true of transitions to **`SC_LOGIC_0`** and the **`negedge_event`**.

The Boolean **`posedge()`** and **`negedge()`** methods apply similarly to the **`event()`** method, and they only apply to the immediately previous delta-cycle.

It is notable that **`sc_buffer`** does *not* support these specializations.

9.5 Exercises

For the following exercises, use the samples provided in www.scftgu.com

Exercise 9.1: Examine, compile, and run `signal_ex`. Does this type of channel lend itself to higher level modeling? Why or why not?

Exercise 9.2: Examine, compile, and run `buffer_ex`. Contrast this with the previous example. In what situations might you prefer **`sc_buffer<`*`T`*`>`** over **`sc_signal<`*`T`*`>`** or visa versa?

Exercise 9.3: Examine, compile, and run `danger_ex`. Change the code to make it less dangerous.

Exercise 9.4: Examine, compile, and run `resolved_ex`. Observe the definition of `DEBUG_SYSTEMC` in `../Makefile.defs`. See if you can measure the performance difference.

Exercise 9.5: Examine the interactive simulation illustrating the simulation engine model in the downloaded examples. Notice how the **`update()`** method was used to extract information. Would you consider this an acceptable use for normal modeling? Why?