Plan

- Ch1 – Overview of SystemC
- Ch2 – Data Types
- Ch3 – Modules
- Ch4 – Notion of Time
- Ch5 – Concurrency
- Ch6 – Predefined Channels
- Ch7 – Structure
- Ch8 – Communication
- Ch9 – Custom Channels and Data
- Ch10 – Transaction Level Modeling

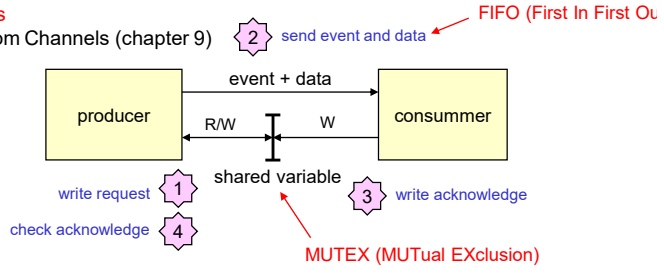
Predefined Channels

Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	Channels & Interfaces	Data types Logic, Integers, Fixed point
	Events, Sensitivity & Notification	Modules & Hierarchy	

- Introduction
- Basic Channels
- Evaluate-Update Channels
- Signal Tracing

Why Predefined Channels ?

- Communication between concurrent processes
  - using events
  - using module member data
  - using shared variables (more difficult)
- Events let us manage shared variables
  - careful coding because events may be missed !
- Built-in mechanisms
  - tedium of these chores
  - aid communications
  - encapsulate complex communication
- 2 types of channels
  - Primitive Channels
  - Hierarchical/Custom Channels (chapter 9)



Primitive Channels

- Base class of all primitive channel : sc_prim_channel
  - sc_mutex
  - sc_semaphore
  - sc_fifo
  - sc_signal
- Access to the update phase to the scheduler
  - update()
- Cannot be instantiate !

sc_prim_channel class

for SC_METHOD void **next_trigger**(…)

for SC_THREAD void **wait**(…)

Phases
virtual void **before_end_of_elaboration**()
virtual void **end_of_elaboration**()
virtual void **start_of_simulation**()
virtual void **end_of_simulation**()

Update phase

cause the scheduler to queue an update request

void **request_update**()

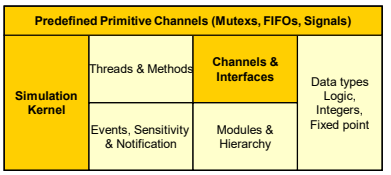
virtual void **update**()
{
…
}

called back by the scheduler during the update phase in response to a call to request_update()

Graphical notation Primitive Channels

bold line

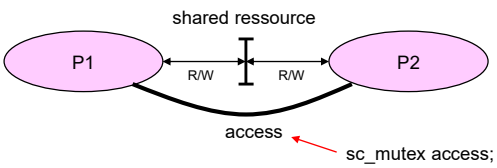
Predefined Channels



- Introduction
- Basic Channels
- Evaluate-Update Channels
- Signal Tracing

Mutex (Mutual Exclusion)

- Useful to model software part
- Multiprogram threads share a common resource
  - variables, tables, …
  - files
- SystemC : sc_mutex class
- Principle



return — true : success, false : already locked

virtual int **lock**(); → blocking style

virtual int **trylock**(); → Non-blocking style

virtual int **unlock**(); → unlock

1. P1 lock the mutex as possible to prevent P2 using resource
2. P1 uses the ressource
3. P1 unlock the mutex

Semaphore

- More than one copy or owner
- Example : parking space in a parking lot
- SystemC : sc_semaphore class

Constructors

```
explicit sc_semaphore( int sem_value)
sc_semaphore( const char*, int sem_value)
```

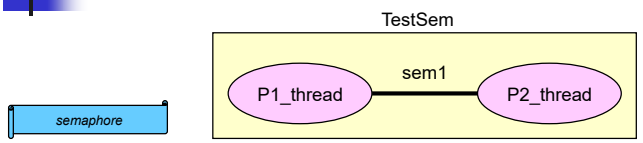
⇨

```
sc_semaphore sem1(5);  // 5 tokens
sc_semaphore sem2("SEM2", 3); // SEM2 has 3 tokens
```

Methods

virtual int **wait**();	blocking →	counter > 0 : decrement counter else wait a post()
virtual int **trywait**();	non-blocking →	counter > 0 : decrement counter else return false
virtual int **post**();	→	increment counter (return always value 0)
virtual int **get_value**() const;	→	return counter value

Semaphore Example




```
...
sc_semaphore sem;
SC_CTOR(TestSem) : sem(0)
{
    SC_THREAD(P1);
    SC_THREAD(P2);
}
```

```
void P1_thread()
  {
    while (true)
    {
      wait(10, SC_MS);
      cout << sc_time_stamp() << " : P1 = Post 1" << endl;
      sem.post();
      cout << sc_time_stamp() << " : P1 = Post 2" << endl;
      sem.post();
      cout << sc_time_stamp() << " : P1 = Post 3" << endl;
      sem.post();

      wait(20, SC_MS);
      cout << sc_time_stamp() << " : P1 = Post 4" << endl;
      sem.post();
    }
  }
```

```
0 s : P2 = Wait ...
10 ms : P1 = Post 1
10 ms : P1 = Post 2
10 ms : P1 = Post 3
10 ms : P2 = Write ...
12 ms : P2 = Wait ...
12 ms : P2 = Write ...
14 ms : P2 = Wait ...
14 ms : P2 = Write ...
16 ms : P2 = Wait ...
30 ms : P1 = Post 4
30 ms : P2 = Write ...
32 ms : P2 = Wait ...
40 ms : P1 = Post 1
40 ms : P1 = Post 2
40 ms : P1 = Post 3
40 ms : P2 = Write ...
```

```
void P2_thread()
{
  while (true)
  {
    cout << sc_time_stamp() << " : P2 = Wait ..." << endl;
    sem.wait();
    cout << sc_time_stamp() << " : P2 = Write ..." << endl;
    wait(2, SC_MS);
  }
}
```

FIFO (First In First Out)

- Most popular channel
  - Modeling at the architectural level (Khan process networks)
  - managing data flow
- By default, a FIFO has a depth of 16

Constructors

```
explicit sc_fifo( int size = 16 );
explicit sc_fifo( const char* name, int size = 16);
```

Methods

Read
```
virtual void read( T& );
virtual T read();
virtual bool nb_read( T& );
```

Write
```
virtual void write( const T& );
virtual bool nb_write( const T& );
sc_fifo<T>& operator= ( const T& );
```

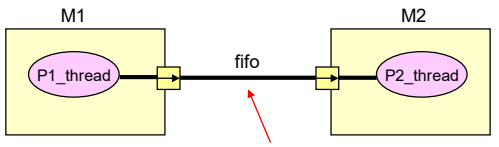
Wait
```
virtual const sc_event& data_written_event() const;
virtual const sc_event& data_read_event() const;
```

Return the number of values that are available for reading in the current delta cycle
```
virtual int num_available() const;
```

Return the number of empty slots that are free for writing in the current delta cycle
```
virtual int num_free() const;
```


FIFO – Input / Output

- Specialized port class
  - reading from a FIFO
  - writing to a FIFO




```
SC_MODULE(M1)
{
    sc_fifo_out<int> fifo_out;
    …

    void P1_thread()  // Process
    {
        …
        wait(10, SC_MS)
        fifo_out->write(value);
```

blocking if number of value > 16

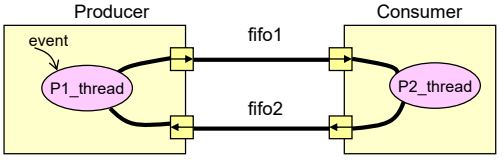
```
SC_MODULE(M2)
{
    sc_fifo_in<int> fifo_in;
    …

    void P2_thread()  // Process
    {
        …
        wait(5, SC_MS)
        value = fifo_in->read();
```

blocking

FIFO – Example (1/2)

period of event : 10 ms

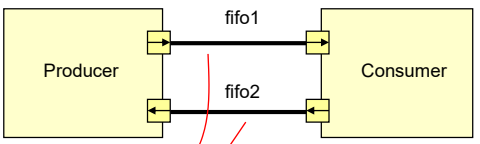
Type of FIFO : integer




```
SC_MODULE(Producer)
{
    sc_fifo_out<int> fifo_out;
    sc_fifo_in<int> fifo_in;

    SC_CTOR(Producer)
    {
        SC_THREAD(P1_thread);
    }
    void P1_thread()
    {
        int value = 1;
        while (true)
        {
            wait(10, SC_MS);
            cout << sc_time_stamp() << " : Producer => Writing to fifo1" << endl;
            fifo_out->write(value++);
            cout << sc_time_stamp() << " : Producer => Waiting on fifo2" << endl;
            int val = fifo_in->read();
            cout << sc_time_stamp() << " : Producer => Reading fifo2 = " << val ;
            cout << endl;
        }
    }
};
```

```
SC_MODULE(Consumer)
{
    sc_fifo_out<int> fifo_out;
    sc_fifo_in<int> fifo_in;

    SC_CTOR(Consumer)
    {
        SC_THREAD(P2_thread);
    }
    void P2_thread()
    {
        while (true)
        {
            cout << sc_time_stamp() << " : Consumer => Waiting on fifo1" << endl;
            int val = fifo_in->read();
            cout << sc_time_stamp() << " : Consumer => Reading fifo1 = " << val << endl;
            wait(4, SC_MS);
            cout << sc_time_stamp() << " : Consumer => Writing to fifo2" << endl;
            fifo_out->write(val*2);
        }
    }
};
```


FIFO – Example (2/2)

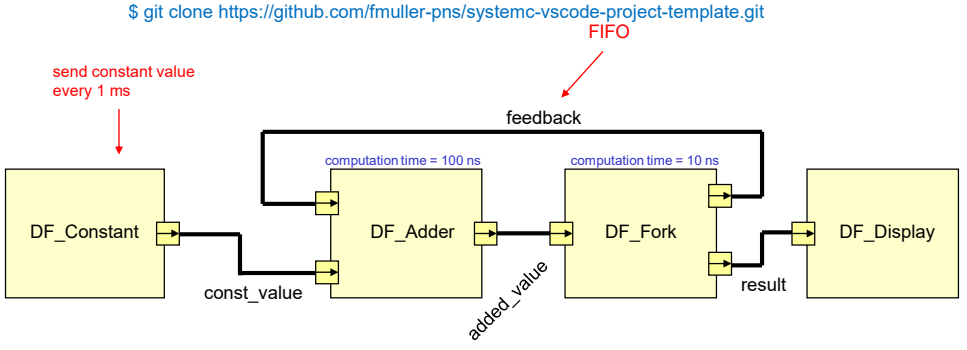



```
int sc_main(int argc, char *argv[])
{
    sc_fifo<int> fifo1, fifo2;

    Producer producer("Producer");
        producer.fifo_out(fifo1);
        producer.fifo_in(fifo2);
    Consumer consumer("Consumer");
        consumer.fifo_out(fifo2);
        consumer.fifo_in(fifo1);

    sc_start(100, SC_MS);

    return 0;
}
```

```
0 s : Consumer => Waiting on fifo1
10 ms : Producer => Writing to fifo1
10 ms : Producer => Waiting on fifo2
10 ms : Consumer => Reading fifo1 = 1
14 ms : Consumer => Writing to fifo2
14 ms : Consumer => Waiting on fifo1
14 ms : Producer => Reading fifo2 = 2
24 ms : Producer => Writing to fifo1
24 ms : Producer => Waiting on fifo2
24 ms : Consumer => Reading fifo1 = 2
28 ms : Consumer => Writing to fifo2
28 ms : Consumer => Waiting on fifo1
28 ms : Producer => Reading fifo2 = 4
38 ms : Producer => Writing to fifo1
38 ms : Producer => Waiting on fifo2
38 ms : Consumer => Reading fifo1 = 3
42 ms : Consumer => Writing to fifo2
42 ms : Consumer => Waiting on fifo1
42 ms : Producer => Reading fifo2 = 6
52 ms : Producer => Writing to fifo1
52 ms : Producer => Waiting on fifo2
…
```

Exercise : DataFlow

Using Visual Studio Code and template project:
$ git clone https://github.com/fmuller-pns/systemc-vscode-project-template.git



- const_value, feedback, added_value and result are FIFO
- The type is a double
- DF_Display displays N times

Write SystemC program

Predefined Channels

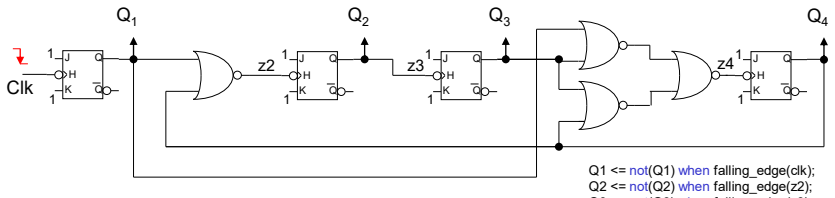
Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	Channels & Interfaces	Data types Logic, Integers, Fixed point
	Events, Sensitivity & Notification	Modules & Hierarchy	

- Introduction
- Basic Channels
- Evaluate-Update Channels
- Signal Tracing

Principle

- Modeling hardware part
- Behavior of a signal
  - instantaneous activity
  - single source / multiple sinks
  - all sinks “see” a signal update at the same time

Example : Asynchronous Decimal Counter




```
Q1 <= not(Q1) when falling_edge(clk);
Q2 <= not(Q2) when falling_edge(z2);
Q3 <= not(Q3) when falling_edge(z3);
Q4 <= not(Q4) when falling_edge(z4);
z2 <= Q1 nand Q4;
z3 <= Q2;
z4 <= (Q3 nand Q1) nand (Q3 nand Q4);
```

VHDL

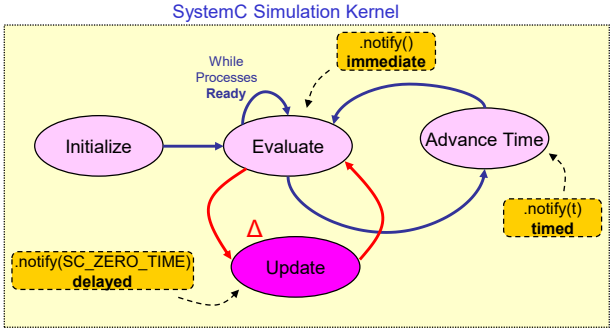
Principle
Evaluate – Update Paradigm

- Possible to go from evaluate to update and back
  - Time doesn’t advance
- Signal channels use this update phase as a point of data synchronization
- Two storage locations
  - current value
  - new value

signal

current	new

SystemC Simulation Kernel



Evaluation Phase

- process write value to a signal channel
- store the value in new value field
- calls request_update()

End of Evaluation Phase

there are no more processes in the ready state

Update Phase

kernel calls update() method for each channel that request an update
- copy new value into current value

Advance Time Phase

2 conditions:
- there are no more processes in the ready state
- no request update

Signal

- Intended to model the behavior of a single piece of wire carrying a digital electronic signal
- Use evaluate-Update paradigm
- sc_signal class is equal to
  - signal (VHDL)
  - reg that use non blocking assignments (<=) exclusively

Constructors

type

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

Methods

Read

```
virtual const T& read() const;
operator const T& () const;
```

call request_update()

Write

```
virtual void write( const T& );
sc_signal<T>& operator= ( const T& );
sc_signal<T>& operator= ( const sc_signal<T>& );
```

Event

```
virtual const sc_event& default_event() const;
virtual const sc_event& value_changed_event() const;
virtual bool event() const;
```

Example

```
sc_signal<bool> s1, s2;
bool a;

equivalent { a = s1.read();
             a = s1;

equivalent { s2.write(true);
             s2 = true;

equivalent { s1.write(s2.read());
             s1 = s2;

sensitive << s2.default_event();
wait(s1.default_event());
if (s2.event() == true)
    …
```


Signal
Dangerous syntax

- SystemC has overloaded the assignment and copy operator

```
int a;
signal<int> s;
```

→

```
a = s.read();
a = s;
s = 56;
```

- These syntaxes dangerous relates to the issue of the evaluate-update paradigm

```
// 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;
```

x,y,r are variables	x,y,r are sc_signal
x = 3, y = 4, r = 0	x = 3, y = 4, r = 0
↓	↓
?	?
Radius is 5	Radius is 0

Signal
Multiple writers / One readers

```
SC_MODULE(M)
{
   sc_signal<bool> sig; // Channel
   SC_CTOR(M)
   {
      SC_THREAD(writer);
      SC_THREAD(reader);
      SC_METHOD(writer2);
         sensitive << sig. posedge_event(); // Sensitive to the pos edge event
   }
   void writer()
   {
      wait(50, SC_NS);
      sig.write(false);
      sig.write(true);
      wait(50, SC_NS);
      sig = false; // Calls operator= ( const T& )
   }
   void reader()
   {
      wait(sig.value_changed_event());
      bool i = sig.read(); // Reads true
      wait(sig.value_changed_event());
      i = sig; // Calls operator const T& () which returns false
   }
   void writer2()
   {
      sig.write(!sig.read()); // An error. A signal shall not have multiple writers
   }
}
```

sc_signal<bool>
sc_signal<sc_logic>

```
sensitive    << sig.posedge_event()
             << sig.negedge_event();
wait(sig.posedge_event());
wait(sig.negedge_event());
if (sig.pos())
   …
if (sig.neg())
   …
```

sc_signal_resolved class

solution

ERROR !!

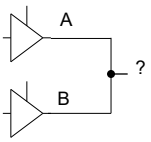
sc_signal_resolved / sc_signal_rv

- Channel derived from class sc_signal
- Resolved signal may be written by multiple processes
- conflicting values being resolved within the channel
  - Resolution table

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

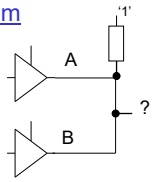
sc_signal_resolved name;
sc_signal_rv<5> name;

width



Example: Pullup mechanism

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




```
struct signal_pullup: public sc_signal_resolved
{
   signal_pullup()
   { }
   explicit signal_pullup(const char *nm) : sc_signal_resolved(nm)
   { }
   virtual void update()
   {
      sc_logic_resolve::resolve( m_new_val, m_val_vec );
      if (m_new_val == SC_LOGIC_Z)
         m_new_val = SC_LOGIC_1;
      base_type::update();
   }
```

sc_buffer

- Channel derived from class sc_signal

sc_signal => default_event()

A value-changed event is notified when the value of the signal is changed.

sc_buffer => default_event()

A value-changed event is notified whenever the buffer is written

Example

```
sc_signal<bool> a;

a.write(true);   → event
a.write(false);  → event
a.write(false);
```

```
sc_buffer<bool> a;

a.write(true);   → event
a.write(false);  → event
a.write(false);  → event
```


Predefined Channels

Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	Channels & Interfaces	Data types Logic, Integers, Fixed point
	Events, Sensitivity & Notification	Modules & Hierarchy	

- Introduction
- Basic Channels
- Evaluate-Update Channels
- Signal Tracing

Trace

- A trace file records a time-ordered sequence of value changes during simulation.
- VCD Format (Value Change Dump Format)

```
sc_clock clock("clk", 1, SC_NS);
sc_signal<int> shiftreg_in;
sc_signal<int> shiftreg_out;
…

sc_trace_file *tf = sc_create_vcd_trace_file("wave");

sc_write_comment(tf, "Simulation of Shift Reg at Elaboration Time Resolution");

sc_trace(tf,clock.signal(), "Clock");
sc_trace(tf,shiftreg_in,"shiftreg_in");
sc_trace(tf,shiftreg_out,"shiftreg_out");

sc_start(30, SC_NS);

sc_close_vcd_trace_file(tf);
```

filename

elaboration phase

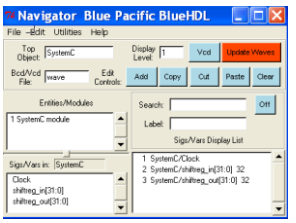
end of simulation

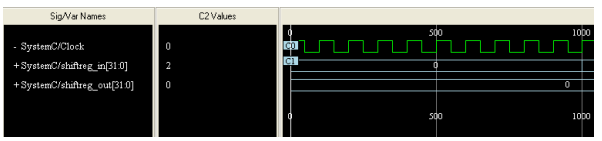
Trace

wave.vcd

```
$comment
Simulation of Shift Reg at Elaboration Time Resolution
$end

$date
   Mar 23, 2005      12:13:02
$end

$version
        SystemC 2.1_oct_12_04.beta --- Mar 16 2005 13:29:45
$end

$timescale
   Mar 23, 2005      12:13:02
$end

$scope module SystemC $end
$var wire   1  aaa  Clock      $end
$var wire  32  aab  shiftreg_in [31:0]  $end
$var wire  32  aac  shiftreg_out [31:0]  $end
$upscope $end
$enddefinitions  $end

$comment
All initial values are dumped below at time 0 sec = 0 timescale units.
$end

$dumpvars
1aaa
b0 aab
b0 aac
$end
```

```
#50
0aaa

#100
1aaa

#150
0aaa

#200
1aaa

#250
0aaa
```





Tracing Aggregate Types

```
const int MAXLEN = 8;

void sc_trace(sc_trace_file *tfile, bool *v, const sc_string& name, int arg_lengh)
{
    char mybuf[MAXLEN];

    for (int j=0; j < arg_length; j++)
    {
        sprintf(mybuf, "[%d]", j);
        sc_trace(tfile, v[j], name + mybuf);
    }
}
```

Using

```
const int MAX = 20;
bool v[MAX];
…
sc_trace(tfile, &v, "v", MAX);
```