SystemC: Co-specification and Embedded System Modeling

EE8205: Embedded Computer Systems
http://www.ecb.torontomu.ca/~courses/ee8205/

Dr. Gul N. Khan
http://www.ecb.torontomu.ca/~gnkhan

Electrical and Computer Engineering
Toronto Metropolitan University

Overview:

- Hardware-Software Co-Specification
- SystemC and Co-specification
- Introduction to SystemC for Co-specification
- A SystemC Primer

Introductory Articles on Hardware-Software Codesign, part of SystemC: From the Ground Up related documents available at the course webpage
Hardware-Software Codesign

Co-design of Embedded Systems consists of the following parts:

- **Co-Specification**
  Developing system specification that describes hardware, software modules and relationship between the hardware and software

- **Co-Synthesis**
  Automatic and semi-automatic design of hardware and software modules to meet the specification

- **Co-Simulation and Co-verification**
  Simultaneous simulation of hardware and software
HW/SW Co-Specification

- Model the Embedded system functionality from an abstract level.
- No concept of hardware or software yet.
- Common environment
  SystemC: based on C++.
- Specification is analyzed to generate a task graph representation of the system functionality.
Co-Specification

• A system design language is needed to describe the functionality of both software and hardware.
• The system is first defined without making any assumptions about the implementation.
• A number of ways to define new specification standards grouped in three categories:
  ➢ SystemC An open source library in C++ that provides a modeling platform for systems with hardware and software components.
SystemC for Co-specification

Open SystemC Initiative (OSCI) 1999 by EDA venders including Synopsys, ARM, CoWare, Fujitsu, etc.

- A C++ based modeling environment containing a class library and a standard ANSI C++ compiler.
- SystemC provides a C++ based modeling platform for IP exchange and co-design of system-level intellectual property (SoC-IP) models.

- **SystemC is not an extension to C++**
  
  SystemC 1.0 and 2.1, 2.2 and 2.3 versions

  It has a new C++ class library
SystemC Library Classes

SystemC classes enable the user to
• Define modules and processes
• Add inter-process/module communication through ports and signals.

Modules/processes can handle a multitude of data types:
  Ranging from bits to bit-vectors, standard C++ types to user define types like structures

Modules and processes also introduce timing, concurrency and reactive behavior.
• Using SystemC requires knowledge of C/C++
SystemC 2.0 Language Architecture

- Standard Channels for Various MOC's
  - Kahn Process Networks
  - Static Dataflow, etc.
- Methodology-Specific Channels
  - Master/Slave Library, etc.

- Elementary Channels
  - Signal, Timer, Mutex, Semaphore, Fifo, etc.

- Core Language
  - Modules
  - Ports
  - Processes
  - Interfaces
  - Channels
  - Events

- Data Types
  - Logic Type (01XZ)
  - Logic Vectors
  - Bits and Bit Vectors
  - Arbitrary Precision Integers
  - Fixed Point Integers

- C++ Language Standard
SystemC 2.0 Language Architecture

- All of SystemC builds on C++
- Upper layers are cleanly built on top of the lower layers
- The SystemC core language provides a minimal set of modeling constructs for structural description, concurrency, communication, and synchronization.
- Data types are separate from the core language and user-defined data types are fully supported.
- Commonly used communication mechanisms such as signals and FIFOs can be built on top of the core language. The MOCs can also be built on top of the core language.
- If desired, lower layers can be used without needing the upper layers.
SystemC Benefits

SystemC 2.x allows the following tasks to be performed within a single language:

• Complex system specifications can be developed and simulated
• System specifications can be refined to mixed software and hardware implementations
• Hardware implementations can be accurately modeled at all the levels.
• Complex data types can be easily modeled, and a flexible fixed-point numeric type is supported
• The extensive knowledge, infrastructure and code base built around C and C++ can be leveraged
SystemC for Co-Specification

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.

Communication protocols:

- SystemC provides multi-level communication semantics that enable you to describe the system I/O protocols at different levels of abstraction.

Waveform tracing:

- SystemC supports tracing of waveforms in VCD, WIF, and ISDB formats.
SystemC Development Environment

your standard C/C++ development environment

header files
libraries

compiler
linker
debugger

"make"

"executable specification"
a.out

executable = simulator

class library and simulation kernel

source files for system and test benches

DSP Interface IP-Core ASIC
SystemC Features

Rich set of data types:
- to support multiple design domains and abstraction levels.
  - The fixed precision data types allow for fast simulation,
  - Arbitrary precision types can be used for computations with large numbers.
  - the fixed-point data types can be used for DSP applications.

Variety of port and signal types:
- To support modeling at different levels of abstraction, from the functional to the RTL.

Clocks:
- SystemC has the notion of clocks (as special signals).
- 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.
SystemC Data types

• SystemC supports:
  ▪ all C/C++ native types
  ▪ plus specific SystemC types

• SystemC types:
  ▪ Types for systems modeling
  ▪ 2 values (‘0’,’1’)
  ▪ 4 values (‘0’,’1’,’Z’,’X’)
  ▪ Arbitrary size integer (Signed/Unsigned)
  ▪ Fixed point types
SC_Logic, SC_int types

SC_Logic: More general than bool, 4 values:
   (‘0’ (false), ‘1’ (true), ‘X’ (undefined), ‘Z’ (high-impedance))

Assignment like bool
   my_logic = ‘0’;
   my_logic = ‘Z’;

Operators like bool but Simulation time bigger than bool

Declaration
   sc_logic my_logic;

Fixed precision Integer: Used when arithmetic operations need fixed size arithmetic operands
   • INT can be converted in UINT and vice-versa
   • 1-64 bits integer in SystemC
      sc_int<n> -- signed integer with n-bits
      sc_uint<n> -- unsigned integer with n-bits
Other SystemC types

Bit Vector

\textit{sc\_bv\langle n\rangle}

2-valued vector (0/1)
Not used in arithmetics operations
Faster simulation than \textit{sc\_lv}

Logic Vector

\textit{sc\_lv\langle n\rangle}

Vector of the 4-valued \textit{sc\_logic} type

Assignment operator (=)

\texttt{my\_vector = \textasciitilde XZ01}\n
Conversion between vector and integer (int or uint)
Assignment between \textit{sc\_bv} and \textit{sc\_lv}

Additional Operators:

<table>
<thead>
<tr>
<th>Reduction</th>
<th>\textit{and_reduction()}</th>
<th>\textit{or_reduction()}</th>
<th>\textit{xor_reduction()}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversion</td>
<td>\textit{to_string()}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## SystemC Data types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc_logic</td>
<td>Simple bit with 4 values(0/1/X/Z)</td>
</tr>
<tr>
<td>sc_int</td>
<td>Signed Integer from 1-64 bits</td>
</tr>
<tr>
<td>sc_uint</td>
<td>Unsigned Integer from 1-64 bits</td>
</tr>
<tr>
<td>sc_bigint</td>
<td>Arbitrary size signed integer</td>
</tr>
<tr>
<td>sc_biguint</td>
<td>Arbitrary size unsigned integer</td>
</tr>
<tr>
<td>sc_bv</td>
<td>Arbitrary size 2-values vector</td>
</tr>
<tr>
<td>sc_lv</td>
<td>Arbitrary size 4-values vector</td>
</tr>
<tr>
<td>sc_fixed</td>
<td>templated signed fixed point</td>
</tr>
<tr>
<td>sc_ufixed</td>
<td>templated unsigned fixed point</td>
</tr>
<tr>
<td>sc_fix</td>
<td>untemplated signed fixed point</td>
</tr>
<tr>
<td>sc_ufix</td>
<td>untemplated unsigned fixed point</td>
</tr>
</tbody>
</table>
# SystemC types

Operators of fixed precision types

<table>
<thead>
<tr>
<th>Bitwise</th>
<th>~</th>
<th>&amp;</th>
<th></th>
<th></th>
<th>^</th>
<th>&gt;&gt;</th>
<th>&lt;&lt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetics</td>
<td>+</td>
<td>-</td>
<td>*</td>
<td>/</td>
<td>%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignement</td>
<td>=</td>
<td>+=</td>
<td>-=</td>
<td>*=</td>
<td>/=</td>
<td>%=</td>
<td>&amp;=</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>!=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&lt;</td>
<td>&lt;=</td>
<td>&gt;</td>
<td>&gt;=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Auto-Inc/Dec</td>
<td>++</td>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit selection</td>
<td>[x]</td>
<td></td>
<td>e.g. mybit = myint[7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Part select</td>
<td>range()</td>
<td>e.g. myrange = myint.range(7,4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concatenation</td>
<td>(,)</td>
<td></td>
<td>e.g. intc = (inta, intb);</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Usage of SystemC types

sc_bit y, sc_bv<8> x;
y = x[6];

sc_bv<16> x, sc_bv<8> y;
y = x.range(0,7);

sc_bv<64> databus, sc_logic result;
result = databus.or_reduce();

sc_lv<32> bus2;
cout << "bus = " << bus2.to_string();
SystemC Specific Features

• Modules:
  ▪ A class called a module: A hierarchical entity that can have other modules or processes contained in it.

• Ports:
  ▪ Modules have ports through which they connect to other modules.
  ▪ Single-direction and bidirectional ports.

• Signals:
  ▪ SystemC supports resolved and unresolved signals.

• Processes:
  ▪ used to describe functionality.
  ▪ contained inside modules.
Modules

The basic building block in SystemC to partition a design.

- Modules are similar to, **entity** in VHDL
- Modules allow designers to hide internal data representation and algorithms from other modules.

Declaration

- Using the macro `SC_MODULE`
  ```
  SC_MODULE(modulename) {
  ```
- Using typical C++ struct or class declaration:
  ```
  struct modulename : sc_module {
  ```

Elements:

Ports, local signals, local data, other modules, processes, and constructors
**SystemC Constructor**

**Constructor:** Each module should include a constructor that identifies processes as methods using the SC_METHOD macro. SC_METHOD ( funct ) ; Identifies the function or process funct Methods are called similar to C++ as:

```
function_type module_name::function_name(data_type var_name) { … }
```

- SC_METHOD process is triggered by events and executes all the statements in it before returning control to the SystemC kernel.
- A Method needs to be made sensitive to some internal or external signal. e.g., sensitive_pos << clock or sensitive_neg << clock
- Process and threads get executed automatically in the constructor even if an event in sensitivity list does not occur. To prevent this un-intentional execution, dont_initialize() function is used.
SystemC Module

```cpp
SC_MODULE(module_name) {
// Ports declaration
// Signals declaration
// Module constructor : SC_CTOR
// Process constructors and sensibility list
//    SC_METHOD
// Sub-Modules creation and port mappings
// Signals initialization
}
```
Signals and Ports

**Ports** of a module are the external interfaces that pass information to and from a module.

```cpp
sc_inout<data_type> port_name;
```
- Create an input-output port of ‘data_type’ with name ‘port_name’.
- `sc_in` and `sc_out` create input and output ports respectively.

**Signals** are used to connect module ports allowing modules to communicate.

```cpp
sc_signal<data_type> sig_name;
```
- Create a signal of type ‘data_type’ and name it ‘sig_name’.
- hardware module has its own *input* and *output ports* to which these signals are mapped or bound.

For example:
```cpp
in_tmp = in.read();
out.write(out_temp);
```
2-to-1 Mux Modules

Module constructor – SC_CTOR is Similar to an “architecture“ in VHDL

```systemc
SC_MODULE( Mux21 ) {
  sc_in< sc_uint<8> >   in1;
  sc_in< sc_uint<8> >   in2;
  sc_in< bool >         selection;
  sc_out< sc_uint<8> >  out;

  void MuxImplement( void );
  SC_CTOR( Mux21 ) {
    SC_METHOD( MuxImplement );
    sensitive << selection;
    sensitive << in1;
    sensitive << in2;
  }
}
```
SystemC Counter Code

```c
struct counter : sc_module {
    // the counter module
    sc_inout<int> in;  // the input/output port of int type
    sc_in<bool> clk;  // Boolean input port for clock

    void counter_fn();  // counter module function

    SC_CTOR(counter) {
        SC_METHOD(counter_fn);  // declare the counter_fn as a method
        dont_initialize();  // don’t run it at first execution
        sensitive_pos << clk;  // make it sensitive to +ve clock edge
    }
}

// software block that check/reset the counter value, part of sc_main
void check_for_10(int *counted) {
    if (*counted == 10) {
        printf("Max count (10) reached ... Reset count to Zero\n");
        *counted = 0;
    }
}
```
void check_for_10(int *counted);

int sc_main(int argc, char *argv[]) {
    sc_signal<int> counting; // the signal for the counting variable
    sc_clock clock("clock", 20, 0.5); // clock period = 20 duty cycle = 50%
    int counted; // internal variable, to store the value in counting signal
    counting.write(0); // reset the counting signal to zero at start
    counter COUNT("counter"); // call counter module
    COUNT.in(counting); // map the ports by name
    COUNT.clk(clock);       // map the ports by name
    for (unsigned char i = 0; i < 21; i++) {
        counted = counting.read(); // copy the signal onto the variable
        check_for_10(&counted); // call the software block & check for 10
        counting.write(counted);  // copy the variable onto the signal
        sc_start(20);             // run the clock for one period
    }
    return 0;
}

BCD Counter Example Main Code
Counter Main Code with Tracing

```c
int sc_main(int argc, char *argv[]) {
    sc_signal<int> counting; // the signal for the counting variable
    sc_clock clock("clock", 20, 0.5); // clock; time period = 20 duty cycle = 50%
    int counted; // internal variable, to stores the value in counting signal
    // create the trace-file by the name of "counter_tracefile.vcd"
    sc_trace_file *tf = sc_create_vcd_trace_file("counter_tracefile");
    // trace the clock and the counting signals
    sc_trace(tf, clock.signal(), "clock");
    sc_trace(tf, counting, "counting");
    counting.write(0); // reset the counting signal to zero at start
    counter COUNT("counter"); // call counter module. COUNT is just a temp var
    COUNT.in(counting); // map the ports by name
    COUNT.clk(clock); // map the ports by name
    for (unsigned char i = 0; i < 21; i++) {
        ....
    }
    sc_close_vcd_trace_file(tf); // close the tracefile
    return 0;
}
```
#include "systemc.h"
#define COUNTER
struct counter : sc_module {  // the counter module
    sc_inout<int> in;       // the input/output port of int type
    sc_in<bool> clk;       // Boolean input port for clock
    void counter_fn();     // counter module function
    SC_CTOR( counter ) { // counter constructor
        SC_METHOD( counter_fn ); // declare the counter_fn as a method
        dont_initialize();      // don’t run it at first execution
        sensitive_pos << clk;   // make it sensitive to +ve clock edge
    }
} ;
void counter :: counter_fn() {
    in.write(in.read() + 1);
    printf("in=%d\n", in.read());
}
Module Instantiation

• Instantiate module
  Module_type Inst_module ("label");

• Instantiate module as a pointer
  Module_type *pInst_module;

  // Instantiate at the module constructor SC_CTOR
  pInst_module = new module_type ("label");

  Inst_module.a(s);
  Inst_module.b(c);
  Inst_module.q(q);

  pInst_module -> a(s);
  pInst_module -> b(c);
  pInst_module -> q(q);
Signals

\texttt{sc\_signal<type > q, s, c;}

- Positional Connection
- Named Connection
SC_MODULE(filter) {
    // Sub-modules: "components"
    sample *s1;
    coeff *c1;
    mult *m1;
    sc_signal<sc_uint<32>> q,s,c;
    // Constructor: "architecture"
    SC_CTOR(filter) {
        // Sub-modules instantiation/mapping
        s1 = new sample ("s1");
        s1->d(in(q)); // named mapping
        s1->d(out(s));
        c1 = new coeff("c1");
        c1->d(out(c)); // named mapping
        m1 = new mult ("m1");
        (*m1)(s, c, q) // positional mapping
    }
}
Communication and Synchronization

• SystemC 2.0 and higher has general-purpose
  • **Channel**
    ▪ A mechanism for communication and synchronization
    ▪ They implement one or more *interfaces*
  • **Interface**
    ▪ Specify a set of access methods to the channel
      But it does not implement those methods
  • **Event**
    ▪ Flexible, low-level synchronization primitive
    ▪ Used to construct other forms of synchronization
Communication and Synchronization

Module1

Interfaces

Channel

Events

Ports to Interfaces

Module2
Channels

- Channel implements an interface
  - It must implement all of its defined methods.
- Channels are used for communication between processes inside of modules and between modules.
- Inside of a module a process may directly access a channel.
- If a channel is connected to a port of a module, the process accesses the channel through the port.
Channels

Module

Ports

Module instance

Channel

Module instance

Process

Channel

Process

Process

Module body
Module instances
concurrent processes
Channels

Two types of Channels: Primitive and Hierarchical

- **Primitive Channels:**
  - They have no visible structure and no processes
  - They cannot directly access other primitive channels.
    - sc_signal
    - sc_signal_rv
    - sc_fifo
    - sc_mutex
    - sc_semaphore
    - sc_buffer

- **Hierarchical Channels:**
  - These are modules themselves,
  - may contain processes, other modules etc.
  - may directly access other hierarchical channels.
Channel Usage

Use Primitive Channels:

• when you need to use the request-update semantics.
• when channels are atomic and cannot reasonably be chopped into smaller pieces.
• when speed is absolutely crucial.
  ➢ Using primitive channels can often reduce the number of delta cycles.
• when it doesn't make any sense i.e. trying to build a channel out of processes and other channels such as a semaphore or a mutex.

Use Hierarchical Channels:

• when you would want to be able to explore the underlying structure,
• when channels contain processes or ports,
• when channels contain other channels.
A Communication Modeling  
FIFO Example

Problem definition: FIFO communication channel with blocking read and write operation
Source available in SystemC installation, under “examples\systemc” subdirectory
Processes are functions identified to the SystemC kernel and called if a signal of the sensitivity list changes.

- Processes implement the functionality of modules.
- Similar to C++ functions or methods

Three types of Processes: Methods, Threads and Cthreads

- **Methods**: When activated, executes and returns
  \[ \text{SC\_METHOD(process\_name)} \]

- **Threads**: can be suspended and reactivated
  - \text{wait( )} -> suspends
  - one sensitivity list event -> activates
  \[ \text{SC\_THREAD(process\_name)} \]

- **Cthreads**: are activated by the clock pulse
  \[ \text{SC\_CTHREAD(process\_name, clock value)}; \]
# Processes

<table>
<thead>
<tr>
<th>Type</th>
<th>SC_METHOD</th>
<th>SC_THREAD</th>
<th>SC_CTHREAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activates Exec.</td>
<td>Event in sensit. list</td>
<td>Event in sensit. List</td>
<td>Clock pulse</td>
</tr>
<tr>
<td>Suspends Exec.</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Infinite Loop</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>suspended/reactivated by</td>
<td>N.D.</td>
<td>wait()</td>
<td>wait() wait_until()</td>
</tr>
</tbody>
</table>
Sensitivity List of a Process

• **sensitive** with the ( ) operator
  Takes a single port or signal as argument
  ```plaintext
  sensitive(s1); sensitive(s2); sensitive(s3)
  ```

• **sensitive** with the stream notation
  Takes an arbitrary number of arguments
  ```plaintext
  sensitive << s1 << s2 << s3;
  ```

• **sensitive_pos** with either ( ) or << operator
  Defines sensitivity to positive edge of Boolean signal or clock
  ```plaintext
  sensitive_pos << clk;
  ```

• **sensitive_neg** with either ( ) or << operator
  Defines sensitivity to negative edge of Boolean signal or clock
  ```plaintext
  sensitive_neg << clk;
  ```
Multiple Process Example

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(); // process-1
    void ramwrite(); // process-2
    SC_CTOR(ram) {
        SC_METHOD(ramread);
        sensitive << addr << rwb;
        SC_METHOD(ramwrite);
        sensitive << addr << datain << rwb;
        for (i=0; i++ < 64) {
            memdata[i] = 0;
        }
    }
};
Thread Process and wait() function

- `wait()` may be used in both `SC_THREAD` and `SC_CTHREAD` processes but not in `SC_METHOD` process block.
- `wait()` suspends execution of the process until the process is invoked again.
- `wait(<pos_int>)` may be used to wait for a certain number of cycles (`SC_CTHREAD` only).

In Synchronous process (`SC_CTHREAD`):
- Statements before the `wait()` are executed in one cycle.
- Statements after the `wait()` are executed in the next cycle.

In Asynchronous process (`SC_THREAD`):
- Statements before the `wait()` are executed in the last event.
- Statements after the `wait()` are executed in the next event.
Thread Process and wait() function

```c
void do_count() {
    while(1) {
        if(reset) {
            value = 0;
        }
        else if (count) {
            value++;
            q.write(value);
        }
        wait();  // wait till next event!
    }
}
```
Thread Example

```cpp
SC_MODULE(my_module) {
    sc_in<bool> id;
    sc_in<bool> clock;
    sc_in<sc_uint<3>> in_a;
    sc_in<sc_uint<3>> in_b;
    sc_out<sc_uint<3>> out_c;

    void my_thread();

    SC_CTOR(my_module) {
        SC_THREAD(my_thread);
        sensitive << clock.pos();
    }
};
```

Thread Implementation

```cpp
//my_module.cpp
void my_module::my_thread()
{
    while(true){
        if (id.read())
            out_c.write(in_a.read());
        else
            out_c.write(in_b.read());
        wait();
    }
}
```
CThread

- Almost identical to `SC_THREAD`, but implements “clocked threads”
- Sensitive only to one edge of one and only one clock
- It is not triggered if inputs other than the clock change

- Models the behavior of unregistered inputs and registered outputs
- Useful for high level simulations, where the clock is used as the only synchronization device
- Adds `wait_until()` and `watching()` semantics for easy deployment.
Another Example

SC_MODULE(countsub) {
    sc_in<double>  in1;
    sc_in<double>  in2;
    sc_out<double> sum;
    sc_out<double> diff;
    sc_in<bool>    clk;
    void addsub();

    // Constructor:  
    SC_CTOR(countsub) {
        // declare addsub as SC_METHOD 
        SC_METHOD(addsub);
        // make it sensitive to 
        // positive clock 
        sensitive_pos << clk;
    }
}

// addsub method 
void countsub::addsub() {
    double a;
    double b;
    a = in1.read();
    b = in2.read();
    sum.write(a+b);
    diff.write(a-b);
}

adder subtractor
in1  in2  clk
sum  diff
The top level is a special function called `sc_main`.

- It is in a file named main.cpp or main.c.
- `sc_main()` is called by SystemC and is the entry point for your code.
- The execution of `sc_main()` until the `sc_start()` function is called.

```c
int sc_main (int argc, char *argv []) {
    // body of function
    sc_start(arg) ;
    return 0 ;
}
```

- `sc_start(arg)` has an optional argument:
  It specifies the number of time units to simulate.
  If it is a null argument the simulation will run forever.
Clocks

• Special object
• How to create?

\[
\text{sc\_clock \ clock\_name ("clock\_label", period, duty\_ratio, offset, initial\_value );}
\]

• Clock connection

\[
f1.\text{clk( clk\_signal );} \quad //\text{where f1 is a module}
\]

• Clock example:

\[
\text{sc\_clock clock1 ("clock1", 20, 0.5, 2, true);}
\]
**sc_time**

*sc_time* data type to measure time. Time is expressed in two parts: a numeric magnitude and a time unit e.g. SC_MS, SC_NS, SC_PS, SC_SEC, etc.

```plaintext
sc_time t(20, SC_NS);
//var t of type sc_time with value of 20ns
```

More Examples:

```plaintext
sc_time t_PERIOD(5, SC_NS);
sc_time t_TIMEOUT (100, SC_MS);
sc_time t_MEASURE, t_CURRENT, t_LAST_CLOCK;
t_MEASURE = (t_CURRENT-t_LAST_CLOCK);
if (t_MEASURE > t_HOLD) { error ("Setup violated") }
```
Time representation in SystemC

**Set Time Resolution:**

```c
sc_set_time_resolution (10, SC_PS) ;
```

- Any time value smaller than this is rounded off
- default; 1 Peco-Second

```c
sc_time t2(3.1416, SC_NS); // t2 gets 3140 PSEC
```

**To Control Simulation:**

```c
sc_start( ) ;
sc_stop( ) ;
```

**To Report Time Information:**

```c
sc_time_stamp( ) // returns the current simulation time
```

```c
cout << sc_time_stamp( ) << endl ;
```

```c
sc_simulation_time( )
```

Returns a value of type double with the current simulation time in the current default time unit
Event

- Something that happens at a specific point in time.
- Has no value or duration

sc_event:

- A class to model an event
  - Can be triggered and caught.

Important

(the source of a few coding errors):

- Events have no duration → you must be watching to catch it
  - If an event occurs, and no processes are waiting to catch it, the event goes unnoticed.
You can perform only two actions with an `sc_event`:

- wait for it
  - `wait(ev1)`
  - `SC_THREAD(my_thread_proc);`
  - `sensitive << ev_1; // or`
  - `sensitive(ev_1)`

- cause it to occur
  - `notify(ev1)`

Common misunderstanding:

- `if (event1) do_something`
  - Events have no value
  - You can test a Boolean that is set by the process that caused an event;
  - However, it is problematic to clear it properly.
To Trigger an Event:

```c
event_name.notify(args);
event_name.notify_delayed(args);
notify(args, event_name);
```

**Immediate Notification:**
causes processes which are sensitive to the event to be made ready to run in the current evaluate phase of the current delta-cycle.

**Delayed Notification:**
causes processes which are sensitive to the event to be made ready to run in the evaluate phase of the next delta-cycle.

**Timed Notification:**
causes processes which are sensitive to the event to be made ready to run at a specified time in the future.
notify( ) Examples

sc_event my_event; // event
sc_time t_zero (0, SC_NS); // variable t_zero of type sc_time
sc_time t(10, SC_MS); // variable t of type sc_time

**Immediate**

my_event.notify();
notify(my_event); // current delta cycle

**Delayed**

my_event.notify_delayed();
my_event.notify(t_zero);
notify(t_zero, my_event); // next delta cycle

**Timed**

my_event.notify(t);
notify(t, my_event);
my_event.notify_delayed(t); // 10 ms delay
cancel ( )

Cancels pending notifications for an event.
• It is supported for delayed and timed notifications.
• not supported for immediate notifications.

Given:

sc_event a, b, c;   // events
sc_time t_zero (0,SC_NS);   // variable t_zero of type sc_time
sc_time t(10, SC_MS);   // variable t of type sc_time
...

a.notify();   // current delta cycle
notify(t_zero, b);   // next delta cycle
notify(t, c);   // 10 ms delay

Cancel of Event Notification:

a.cancel();   // Error! Can't cancel immediate notification
b.cancel();   // cancel notification on event b
c.cancel();   // cancel notification on event c
Time & Execution Interaction

Simulated Execution Activity

Process_A() { //@ t_0
    stmt_{A1};
    stmt_{A2};
    wait(t_1);
    stmt_{A3};
    stmt_{A4};
    wait(t_2);
    stmt_{A5};
    stmt_{A6};
    wait(t_3);
}

Process_B() { //@ t_0
    stmt_{B1};
    stmt_{B2};
    wait(t_1);
    stmt_{B3};
    stmt_{B4};
    wait(t_2);
    stmt_{B5};
    stmt_{B6};
    wait(t_3);
}

Process_C() { //@ t_0
    stmt_{C1};
    stmt_{C2};
    wait(t_1);
    stmt_{C3};
    stmt_{C4};
    wait(t_2);
    stmt_{C5};
    stmt_{C6};
    wait(t_3);
}

Process_D() { //@ t_0
    stmt_{D1};
    stmt_{D2};
    wait(t_1);
    stmt_{D3};
    wait(SC_ZERO_TIME);
    stmt_{D4};
    wait(t_3);
}
wait() and watching()

Legacy SystemC code for Clocked Thread

wait(N); // delay N clock edges
wait_until(delay_expr); // until expr true @ clock

Same as
For (i=0; i!=N; i++)
    wait(); //similar as wait(N)
do wait () while (!expr) ; // same as
    // wait_until(delay_expr)

Previous versions of SystemC also included other
constructs to watch signals such as watching(),
Traffic Light Controller

Highway
- Normally has a green light.

Sensor:
- A car on the East-West side road triggers the sensor
  - The highway light: green => yellow => red,
  - Side road light: red => green.

SystemC Model:
- Uses two different time delays:
  - green to yellow delay >= yellow to red delay
    (to represent the way that a real traffic light works).
Traffic Controller Example

// 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
        sensitive << roadsensor;
        sensitive << clock.pos();
    }
};
Traffic Controller Example

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

