Ports & Signals

- **Port and Signal Binding**
  - Ports and signals to be bound need to have the same type
  - A signal connects two ports
  - A port is bound to one signal (port-to-signal) or to one sub-module port (port-to-port)
- **Resolution**
  - SystemC supports resolved ports and signals
  - Resolved ports/signals have 4-valued logic type (0,1,Z,X)
  - Resolved ports/signals allow multiple drivers
  - Resolved vector ports/vector signals

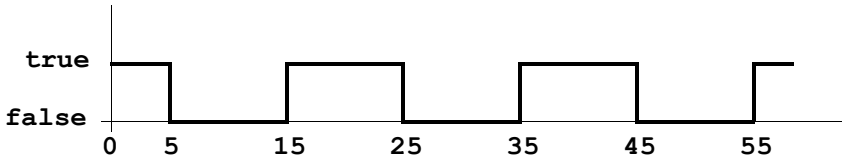
```
sc_in_rv< n > x;       // n bits wide resolved input port
sc_signal_rv< n> y;    // n bits wide resolved signal
```


Clocks

- **SystemC provides a special object *sc_clock***
  - Clocks generate timing signals to synchronize events
  - Multiple clocks with arbitrary phase relations are supported
  - Clock generation:

```
sc_clock clock_name("label",period,duty_ratio,offset,start_value);
sc_clock my_clk ("CLK", 20, SC_NS, 0.5, 5, SC_NS, true);
```

    - time units recommended for *period* & *offset*



  - Clock binding:

```
my_module.clk( my_clk );
```

Data Types

- **Native C/C++ types**
  - **integer types: char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long**
  - **floating point types: float, double, long double**
- **SystemC types**
  - **2-value (’0’, ’1’) logic / logic vector**
  - **4-value (’0’, ’1’, ’Z’, ’X’) logic / logic vector**
  - **Arbitrary sized integer (signed/unsigned)**
  - **Fixed point types (signed/unsigned, templated/untemplated)**
- **User-defined types**

SystemC Data Types

Type	Description
~~sc_bit~~	~~2-value single bit~~ *use bool*
sc_logic	4-value single bit
sc_int	1 to 64 bit signed integer
sc_uint	1 to 64 bit unsigned integer
sc_bigint	arbitrary sized signed integer
sc_biguint	arbitrary sized unsigned integer
sc_bv	arbitrary length 2-value vector
sc_lv	arbitrary length 4-value vector
sc_fixed	templated signed fixed point
sc_ufixed	templated unsigned fixed point
sc_fix	untemplated signed fixed point
sc_ufix	untemplated unsigned fixed point

~~sc_bit~~ / sc_logic

- ~~2-value single bit type: *sc_bit*~~ *use of type bool recommended*
  - ~~'0'=false, '1'=true~~
- 4-value single bit type: *sc_logic*
  - '0'=false, '1'=true, 'X'=unknown/indeterminate value, 'Z'=high-impedance/floating value
- Features:
  - Mixed use of operand types sc_bit and sc_logic
  - Use of character literals for constant assignments

- ~~sc_bit~~ / sc_logic operators

bitwise	& (and)	| (or)	^ (xor)	~ (not)
assignment	=	&=	|=	^=
equality	==	!=		

sc_int / sc_uint / sc_bigint / sc_biguint

- Fixed precision integer types
  - signed: sc_int<*n*> (*n*: word length, $1 \leq n \leq 64$)
  - unsigned: sc_uint<*n*> (*n*: word length, $1 \leq n \leq 64$)
- Arbitrary precision integer types
  - signed: sc_bigint<*n*> (*n*: word length, $n > 64$)
  - unsigned: sc_biguint<*n*> (*n*: word length, $n > 64$)

- Features:
  - Mixed use of operand types sc_int, sc_uint, sc_bigint, sc_biguint and C++ integer types
  - Truncation and/or sign extension if required
  - 2's complement representation

sc_int / sc_uint / sc_bigint / sc_biguint

- sc_int / sc_uint / sc_bigint / sc_biguint operators

bitwise	&	|	^	~	>>	<<		
arithmetic	+	-	*	/	%			
assignment	=	+=	-=	*=	/=	%=	&=	|=	^=
equality	==	!=						
relational	<	<=	>	>=				
auto-inc/dec	++	--						
bit/part select	[]	range()						
concatenation	(,)							

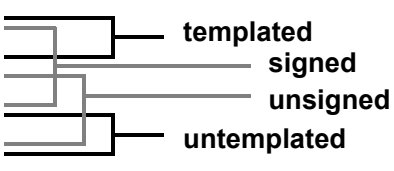
sc_bv / sc_lv

- Arbitrary length bit vector: sc_bv<*n*> (*n*: vector length)
- Arbitrary length logic vector: sc_lv<*n*> (*n*: vector length)
- Features:
  - Assignment between sc_bv and sc_lv
  - Use of string literals for vector constant assignments
  - Conversions between sc_bv/sc_lv and SystemC integer types
  - No arithmetic operation available
- sc_bv / sc_lv

bitwise	&	|	^	~	>>	<<
assignment	=	&=	|=	^=		
equality	==	!=				
bit/part select	[]	range()				
concatenation	(,)					
reduction	and_reduce()	or_reduce()	xor_reduce()			
conversion	to_string()					

sc_fixed / sc_ufixed / sc_fix / sc_ufix

- Fixed point types
  - sc_fixed
  - sc_ufixed
  - sc_fix
  - sc_ufix



  - templated - static arguments (to be known at compile time)
  - untemplated - nonstatic arguments (to be configured during runtime)
  - signed - 2's complement representation
  - unsigned
- Features:
  - Operations performed using arbitrary precision
  - Multiple quantization and overflow modes

sc_fixed / sc_ufixed / sc_fix / sc_ufix

- Templated signed fixed point type: sc_fixed
  - sc_fixed< *wl, iwl, q_mode, o_mode, n_bits* > *var_name* (*init_val*);
- Arguments:
  - wl - total number of bits
  - iwl - number of integer bits
  - q_mode - quantization mode (optional)
  - o_mode - overflow mode (optional)
  - n_bits - number of bits for overflow mode (optional)
- sc_fixed_fast, sc_ufixed_fast, sc_fix_fast, sc_ufix_fast
  - precision limited to 53 bits (C++ type *double* is used)

sc_fixed / sc_ufixed / sc_fix / sc_ufix

- **Example**
  - **sc_fixed< *8, 4* > *my_var* (*-1.75*);**
  - **(1.75)$_{10}$ = (0001.1100)$_2$**
  - **2's complement of (0001.1100)$_2$ = (1110.0100)$_2$**
- **Quantization and overflow modes**

Quantization Mode	
SC_RND	Round
SC_RND_ZERO	Round towards zero
SC_RND_MIN_INF	Rounding towards minus infinity
SC_RND_INF	Rounding towards infinity
SC_RND_CONV	Convergent rounding
SC_TRN	Truncate
SC_TRN_ZERO	Truncate towards Zero

Overflow Mode	
SC_SAT	Saturate
SC_SAT_ZERO	Saturate to zero
SC_SAT_SYM	Saturate symmetrically
SC_WRAP	Wrap-around
SC_WRAP_SYM	Wrap-around symmetrically

Operators

- **Comparison: == != > >= < <=**
- **Arithmetic: ++ -- * / % + -**
- **Bitwise: ~ & | ^**
- **Assignment: = &= |= ^= *= /= += -= <<= >>=**

- **Bit selection: bit(*idx*) [*idx*]**
- **Range selection: range(*high*,*low*) (*high*,*low*)**
- **Conversion: to_double() to_int() to_int64() to_long()**
  to_uint() to_uint64() to_ulong() to_string()
- **Testing: is_zero() is_neg() length()**
- **Bit reduction: and_reduce() nand_reduce() or_reduce() nor_reduce()**
  xor_reduce() xnor_reduce()

User-Defined Data Types

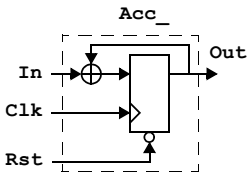
- User-defined data types can be used for ports and signals

```
class complex {
private:
  double re, im;
public:
  complex () { re=0.0; im=0.0; }
  complex (double r, double i) { re=r; im=i; }
  void set(double r, double i) { re=r; im=i; }
  double get_re() { return re; }
  double get_im() { return im; }
  int operator == (const complex &c) const {
    if ( ( re == c.re ) && ( im == c.im() ) ) return 1;
    else return 0;
  }
  // ...
};

sc_signal< complex > c;
```


Modules

- Module is a structural entity
  - helps to split complex designs
- Module can contain
  - ports
  - processes
  - user defined C++ functions
  - internal member data
  - constructor
  - modules and signals
- Valid for Hardware, Firmware/Software & System




```
SC_MODULE(accumulator) {
  // ports (input)
  sc_in<int> In;
  sc_in<bool> Rst;
  sc_in<bool> Clk;
  // ports (output)
  sc_out<int> Out;
  // local member functions
  void accumulate();
  void display(ostream& = cout);
  // local member data
  int Acc_;
  // constructor
  SC_CTOR(accumulator) {
    SC_METHOD(accumulate);
    sensitive_pos << Clk;
    Acc_ = 0 ;
  }
};
```

Module Ports

- Pass data between module-boundary and internal module-processes
- Direction: *in* (read), *out* (write), and *inout* (read & write)
- Type: *<int>* , *<bool>* , etc.
- Special: *clock*

Module Member functions

- Processes
  - functions registered with the SystemC kernel
- User defined functions
  - can be called within processes

Modules & Hierarchy

- Modules may contain sub-modules (hierarchical structure)
- In SC_MODULE:

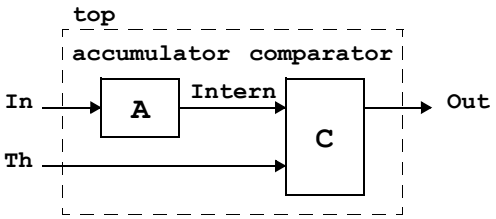
```
// sub-module declaration
module_type *my_module;
```

- In the module constructor of SC_MODULE:

```
// sub-module instantiation and port mapping
SC_CTOR( module_name ) {
  my_module = new module_type ("label");
  my_module -> in1 (sig1);
  my_module -> in2 (sig2);
  my_module -> out1 (sig3);
}
```

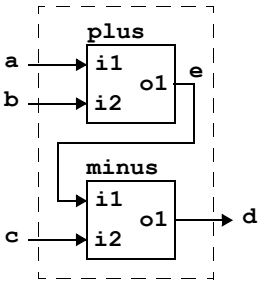
Modules & Hierarchy

- **Module instances**
- **Signals**




```
SC_MODULE(top) {
  // ports
  sc_in<int> In;
  ...
  // module instances
  accumulator A;
  comparator C;
  // signals
  sc_signal<int> Intern;
  // constructor
  SC_CTOR(top): A("A"), C("C") {
    A.In(In);
    A.Out(Intern);
    ...
  }
};
```


Modules & Hierarchy



d = (a + b) - c

```
SC_MODULE( plus ) {
  sc_in<int> i1;
  sc_in<int> i2;
  sc_out<int> o1;
  ...
};

SC_MODULE( minus ) {
  sc_in<int> i1;
  sc_in<int> i2;
  sc_out<int> o1;
  ...
};
```

```
SC_MODULE( alu ) {
  sc_in<int> a;
  sc_in<int> b;
  sc_in<int> c;
  sc_out<int> d;
  plus *p;
  minus *m;
  sc_signal<int> e;
  SC_CTOR( alu ) {
    p = new plus ( "PLUS" );
    p->i1 (a);
    p->i2 (b);
    p->o1 (e);
    m = new minus ( "MINUS" );
    (*m) (e,c,d);
  }
};
```

Processes

- Basic unit of concurrent execution
- Not hierarchical
- Contained inside modules
- Communication done via signals, module ports
- Encapsulates functionality

- SystemC takes care of scheduling the concurrent processes

- SystemC = Language (C++ Library) + Scheduler

Process Activation

- Processes have sensitivity lists
- Processes are triggered by events on sensitive signals

- Process Types
  - Method (SC_METHOD)
    - asynchronous block, like a combinational function
  - Thread (SC_THREAD)
    - asynchronous process
  - Clocked Thread (SC_CTHREAD)
    - synchronous process
    - limited suspend/resume control
  - Remote Procedure Call (RPC)

Process Declaration

- Declaration of member function (in SC_MODULE)

```
// process declaration
void my_process ();
```

- Instantiation (in module constructor of SC_MODULE)

```
// specification of process type and sensitivity
SC_CTOR( module_name ) {
  SC_METHOD( my_process );
  sensitive << sig1 << sig2;
}
```

- Definition of member function (in SC_MODULE or somewhere else)

```
// process specification
void module_name::my_process () {
  ...
}
```


Method – SC_METHOD

- Entire process executed once when a signal in its sensitivity list changes
  - static sensitivity – "sensitive << i1 << i2;"
  - dynamic sensitivity – "next_trigger(...);" overrides previous sensitivity list (timeout possible)
- Can not be suspended
- Local variables loose their values between successive calls
- Most similar to a usual C++ function
- Fastest

```
SC_MODULE( plus ) {
  sc_in<int> i1, i2;
  sc_out<int> o1;

  void do_plus();
  SC_CTOR( plus ) {
    SC_METHOD( do_plus );
    sensitive << i1 << i2;
  }
};
```

sensitivity list

```
void plus::do_plus() {
  int arg1, arg2, sum;

  arg1 = i1.read();
  arg2 = i2.read();
  sum = arg1 + arg2;
  o1.write(sum);
}
```

Thread – SC_THREAD

- Infinite loop limited by event boundaries:
  execution suspended by “wait()” statement(s)
  - static sensitivity – “sensitive << i1 << i2;” and “wait();”
  - dynamic sensitivity – “wait(*conditions*);” “wait_until(*delay_expr*)” – (timeout possible)
  - subject for interpretations...
- Activated (re-activated) when any of the signals in the sensitivity list changes
- Local variables are saved (similar to static variables in C++ functions)
- Slower than method with module data members to store the ‘state’

```
SC_MODULE( plus ) {
  sc_in<int> i1, i2;
  sc_out<int> o1;

  void do_plus();
  SC_CTOR( plus ) {
    SC_THREAD( do_plus );
    sensitive << i1 << i2;
  }
};
```

sensitivity list →

```
void plus::do_plus() {
  int arg1, arg2;
  int sum;
  while ( true ) {
    arg1 = i1.read();
    arg2 = i2.read();
    sum = arg1 + arg2;
    o1.write(sum);
    wait();
  }
}
```


Thread – SC_THREAD

- Sensitivity

```
wait(time);
wait(event);
wait(event1 | event2 ...);          // any of these
wait(event1 & event2 ...);          // all of these
wait(timeout,event);                // event with timeout
wait(timeout,event1|event2...);     // any event with timeout
wait(timeout,event1&event2...);     // all events with timeout
wait();                             // static sensitivity

// Example
sc_event ack_event, bus_error_event;
// ...
wait(t_MAX_DELAY, ack_event | bus_error_event);
if ( timed_out() ) break;    // deprecated
```

Clocked thread – SC_CTHREAD

- Particular thread: infinite loop, sensitive only to one edge of one clock
- Suspended by
  - "wait()" – waiting for the next clock edge
  - "wait(*N*)" – delay *N* clock cycles
- Only synchronous systems
- Slowest

```
SC_MODULE( plus ) {
  sc_in_clk clk;
  sc_in<int> i1, i2;
  sc_out<int> o1;

  void do_plus();
  SC_CTOR( plus ) {
    SC_CTHREAD (do_plus,clk.pos());
  }
};
```

```
void plus::do_plus() {
  int arg1, arg2;
  int sum;
  while ( true ) {
    arg1 = i1.read();
    arg2 = i2.read();
    sum = arg1 + arg2;
    o1.write(sum);
    wait();
  }
}
```


Processes

	SC_METHOD	SC_THREAD	SC_CTHREAD
triggered	by signal events	by signal events	by clock edge
infinite loop	no	yes	yes
execution suspended	no	yes	yes
suspend & resume	-	wait() wait_until()	wait()

construct & sensitize method	SC_METHOD(*p*);
	
	sensitive << *s*;
	sensitive << s.pos();
	sensitive << s.neg();
		SC_THREAD(*p*);
		
		sensitive << *s*;
		sensitive << s.pos();
		sensitive << s.neg();
			SC_CTHREAD (*p,clock*.pos());
			
			SC_CTHREAD (*p,clock*.neg());

modelling example (hardware)	combinational logic	sequential logic at RT level (asynchronous reset, etc.)	sequential logic at higher design levels

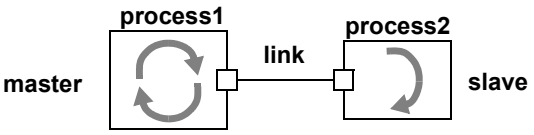
Remote Procedure Call – RPC

- Abstract communication and execution semantics for functional level
- Master/slave ports and processes
- Equivalent to function call but without function pointer
- Structure is key for re-use (split behavior of modules)
- RPC chain / concurrent RPC chains

```
SC_MODULE(M1) {
  sc_outmaster<int> Out;
  ...
  void process1();
  SC_CTOR(M1) {
    SC_METHOD(process1);
    sensitive << ...;
  }
};
```

```
SC_MODULE(M2) {
  sc_inslave<int> In;
  ...
  void process2();
  SC_CTOR(M2) {
    SC_SLAVE(process2, In);
  }
};
```

```
SC_MODULE(top) {
  sc_link_mp<int> link;
  ...
};
```



Time Control in SystemC

- "sc_time" – numeric magnitude + time unit
  - time units – SC_SEC, SC_MS, SC_US, SC_NS, SC_PS, SC_FS
    `sc_time t_PERIOD (5, SC_NS)`
- "sc_start()" – simulation phase = initialization + execution
  - "sc_start()" , ~~"sc_start(10000)"~~ , "sc_start(10, SC_SEC)"
- "sc_time_stamp()" – current simulation time
  - "cout << sc_time_stamp() << endl;"
- "wait()" – delaying for a certain time
  - "wait(2,SC_MS);"
- Waiting and Watching
  - Suspend / reactivate process execution (SC_THREAD, SC_CTHREAD)
    - Suspension: wait(); / Reactivation: event on a sensitive signal
  - Halt process execution until an event occurs (SC_CTHREAD only)
    - do { wait(); } while (*my_bool_sig*.read() != true);

Cycle-Accurate Simulation Scheduler

- (1) All clock signals that change their value at the current time are assigned their new value.
- (2) All *SC_METHOD* / *SC_THREAD* processes with inputs that have changed are executed. The entire bodies of *SC_METHOD* processes are executed. *SC_THREAD* processes are executed until the next *wait()* statement suspends execution. *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 are saved in a queue to be executed in step 5. All outputs of *SC_METHOD* / *SC_THREAD* processes that were executed in step 2 are also updated.
- (4) Step 2 and step 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.

TTÜ1918
Department of Computer Engineering

Example #1

```
// David C. Black, Jack Donovan. SystemC: From the Ground Up. Springer, 2004
#include <systemc.h>
#include <iostream>
SC_MODULE (HelloWorld) {                        // Declare the module class
  sc_in_clk iclk;                               // Define the clock port
  SC_CTOR (HelloWorld) {                        // Create a constructor
    SC_METHOD (main_method);                    // Register the main process
    sensitive << iclk.neg();                    // Specify clock sensitivity
    dont_initialize();                          // Skip initial call
  }
  void main_method (void)
    { std::cout << sc_time_stamp() << " Hello world!" << std::endl; }
};

int sc_main (int argc, char *argv[]) {
  const sc_time t_PERIOD(8,SC_NS);              // Declare a time constant
  sc_clock clk ("clk",t_PERIOD);                // Create periodic clock
  HelloWorld iHelloWorld ("iHelloWorld");       // Create an instance
  iHelloWorld.iclk(clk);                        // Connect the clock port and clock
  sc_start(10,SC_NS);                           // Invoke the simulator
  return 0;
}
```

Example #1

```
> g++ -I/cad/sysC/2.2/include -L/cad/sysC/2.2/lib hello.C -lsystemc
> a.out

            SystemC 2.2.0 --- Nov 25 2010 16:19:37
       Copyright (c) 1996-2006 by all Contributors
                  ALL RIGHTS RESERVED
4 ns Hello world!
```

- **Debugging?**
  - **VCD - Value Change Dump format**
  - **GTKWave Electronic Waveform Viewer**
  - **http://www.cs.manchester.ac.uk/apt/projects/tools/gtkwave/**

```
// Create a file for waveform
sc_trace_file *trcf = sc_create_vcd_trace_file("trace-it");
if (trcf==NULL) cout << "Sorry, no tracing..." << endl;
sc_trace(trcf,clk,"clk");

sc_start(10,SC_NS);                          // Invoke the simulator
sc_close_vcd_trace_file(trcf);
```


Example #2 (process_1)

```
// header file: process_1.h

SC_MODULE( process_1 ) {

  // Ports
  sc_in_clk clk;
  sc_in<int> a;
  sc_in<bool> ready_a;
  sc_out<int> b;
  sc_out<bool> ready_b;

  // Process functionality
  void do_process_1();

  // Constructor
  SC_CTOR( process_1 ) {
    SC_CTHREAD(do_process_1,clk.pos());
  }

};
```

```
// implementation file: process_1.cc

#include "systemc.h"
#include "process_1.h"

void process_1::do_process_1()
{
  int v;
  while ( true ) {
    do wait(); while(ready_a.read()!=true);
    v = a.read();
    v += 5;
    cout << "P1: v = " << v << endl;
    b.write( v );

    ready_b.write( true );
    wait();
    ready_b.write( false );
  }
}
```

Example #2 (process_2)

```
// header file: process_2.h

SC_MODULE( process_2 ) {

  // Ports
  sc_in_clk clk;
  sc_in<int> a;
  sc_in<bool> ready_a;
  sc_out<int> b;
  sc_out<bool> ready_b;

  // Process functionality
  void do_process_2();

  // Constructor
  SC_CTOR( process_2 ) {
    SC_CTHREAD(do_process_2,clk.pos());
  }

};
```

```
// implementation file: process_2.cc

#include "systemc.h"
#include "process_2.h"

void process_2::do_process_2()
{
  int v;
  while ( true ) {
    do wait(); while(ready_a.read()!=true);
    v = a.read();
    v += 3;
    cout << "P2: v = " << v << endl;
    b.write( v );

    ready_b.write( true );
    wait();
    ready_b.write( false );
  }
}
```


Example #2 (main)

```
// implementation file: main.cc
#include "systemc.h"
#include "process_1.h"
#include "process_2.h"

int sc_main (int ac,char *av[])
{
  sc_report_handler::set_actions
    ("/IEEE_Std_1666/deprecated",
     SC_DO_NOTHING);
  sc_signal<int> s1 ( "Signal-1" );
  sc_signal<int> s2 ( "Signal-2" );
  sc_signal<bool> ready_s1 ( "Ready-1" );
  sc_signal<bool> ready_s2 ( "Ready-2" );

  sc_clock clock("Clock", 20, 0.5, 0.0);

  process_1 p1 ( "P1" );
  p1.clk( clock );
  p1.a( s1 );
  p1.ready_a( ready_s1 );
  p1.b( s2 );
  p1.ready_b( ready_s2 );

  process_2 p2 ( "P2" );
  p2.clk( clock );
  p2.a( s2 );
  p2.ready_a( ready_s2 );
  p2.b( s1 );
  p2.ready_b( ready_s1 );

  s1.write(0);
  s2.write(0);
  ready_s1.write(true);
  ready_s2.write(false);

  sc_start(100000, SC_NS);

  return 0;
}
```

Example #2 (results)

```
        SystemC 2.2.0 --- Nov 25 2010 16:19:37
      Copyright (c) 1996-2006 by all Contributors
                  ALL RIGHTS RESERVED
P1: v = 5
P2: v = 8
P1: v = 13
P2: v = 16
P1: v = 21
P2: v = 24
P1: v = 29
P2: v = 32
P1: v = 37
P2: v = 40
...
P1: v = 19973
P2: v = 19976
P1: v = 19981
P2: v = 19984
P1: v = 19989
P2: v = 19992
P1: v = 19997
```


ModelSim SE & SystemC

- From “ModelSim SE User’s Manual”
- The following modifications are needed:
  - Replace “sc_main()” with an SC_MODULE, and potentially add a process to contain any testbench code
  - Replace “sc_start()” by using the “run” command in the GUI
  - Remove calls to “sc_initialize()”
  - Export the top level SystemC design unit(s) using the SC_MODULE_EXPORT macro
  - Verify that SystemC signal, ports and modules are explicitly named to avoid port binding and debugging errors. Disabling of automatic name binding may be needed.
    - SC_CTOR (or the SC_MTI_BIND_NAME) macro is used
    - use “-nonamebind” argument when compiling

ModelSim SE & SystemC

```
int sc_main(int argc, char* argv[])
{
  sc_signal<bool> mysig;
  mymod mod("mod");
  mod.outp(mysig);

  sc_start(100, SC_NS);
}
```

```
SC_MODULE(mytop)
{
  sc_signal<bool> mysig;
  mymod mod;

  SC_CTOR(mytop):
    mysig("mysig"),
    mod("mod")
  {
    mod.outp(mysig);
  }
};

SC_MODULE_EXPORT(mytop);
```


MUX example & GNU C/C++

```
// mux.h

#include "systemc.h"

SC_MODULE (mux) {
  int const static bit_number=8;
  sc_in <sc_bv <bit_number> > a_in;
  sc_in <sc_bv <bit_number> > b_in;
  sc_in <sc_bit> juht;
  sc_out <sc_bv <bit_number> > out_sig;

  SC_CTOR(mux):a_in("a_in"),b_in("b_in"),
    juht("juht"),out_sig("out_sig") {
    cout<<"mux constructor"<<endl;
    SC_METHOD( choose_out );
    sensitive<<juht<<a_in<<b_in;
  }

  void choose_out() {
    if (juht.read())
      out_sig.write(a_in.read());
    else out_sig.write(b_in.read());
  }
};
```

```
// t_mux.cpp

#include "systemc.h"
#include "mux.h"

SC_MODULE (t_mux) {
  int const static num_bits=8;
  sc_signal <sc_bv <num_bits> > t_a_in;
  sc_signal <sc_bv <num_bits> > t_b_in;
  sc_signal <sc_bit> t_juht;
  sc_signal <sc_bv <num_bits> > t_out_sig;
  mux* mux_instance;
  void stimulus();

  SC_CTOR(t_mux):t_a_in("t_a_in"),
      t_b_in("t_b_in"),t_juht("t_juht"),
      t_out_sig("t_out_sig") {
    cout<<"t_mux constructor"<<endl;
    mux_instance=new mux("mux");
    mux_instance->a_in(t_a_in);
    mux_instance->b_in(t_b_in);
    mux_instance->juht(t_juht);
    mux_instance->out_sig(t_out_sig);
    SC_THREAD(stimulus);
  }
};
```

MUX example & GNU C/C++

```
// t_mux.cpp ...

void t_mux::stimulus()
{
  t_a_in.write("11111111");
  t_b_in.write("00001111");
  t_juht.write((sc_bit)0);
  wait(1, SC_PS);

  t_juht.write((sc_bit)1);
  wait(10, SC_NS);

  t_juht.write((sc_bit)0);
  wait(10, SC_NS);

  t_a_in.write("11110000");
  t_b_in.write("10101010");
  t_juht.write((sc_bit)0);
  wait(10, SC_NS);

  t_juht.write((sc_bit)1);
  wait();
}
```

```
// t_mux.cpp ...

int sc_main(int args, char* argv[])
{
  t_mux testMux("t_mux");
  sc_trace_file *trcf=
    sc_create_vcd_trace_file("trace-it");
  if(trcf==NULL) cout<<"Sorry, no tracing..."<<endl;
  sc_trace(trcf, testMux.t_a_in, "t_a_in");
  sc_trace(trcf, testMux.t_b_in, "t_b_in");
  sc_trace(trcf, testMux.t_juht, "t_juht");
  sc_trace(trcf, testMux.t_out_sig, "t_out_sig");
  sc_start(40, SC_NS);
  sc_close_vcd_trace_file(trcf);
  return 0;
}
```

```
> g++ -I/cad/sysC/2.2/include -L/cad/sysC/2.2/lib
    t_mux.cpp -lsystemc -o mux
> ./mux
> gtkwave trace-it.vcd
```


MUX example & GTKWave