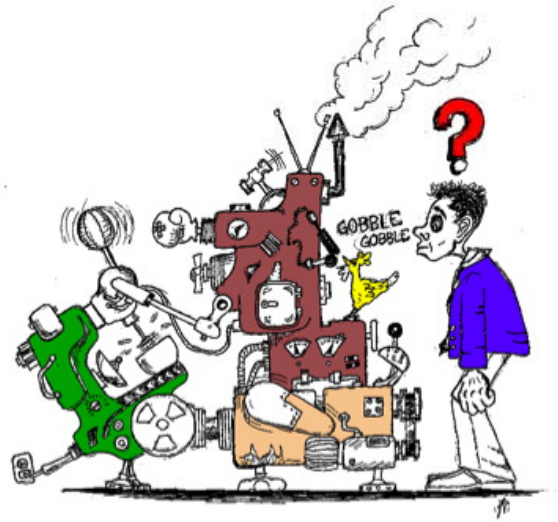
IP Modeling using SystemC
(adapted from an asic-world tutorial)



SystemC Model Development

- SystemC is based on C++, so we can
  - use standard C/C++ development tools to create, simulate, debug and explore algorithmic/architectural descriptions of the design
  - compile on a multitude of C++ compilers on several platforms
- The SystemC model is called executable specification, since we can compile and execute the SystemC model to understand system behavior

SystemC Versions

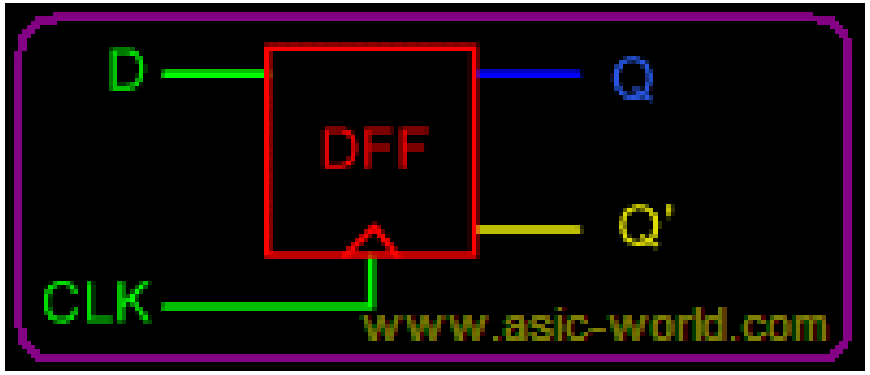
The SystemC class library is being released in stages.

- Release v1.0 (presently at version 1.0.2) provides
  - all the necessary hardware modeling facilities similar to those in HDLs, such as a VHDL
  - a simulation kernel
  - data types appropriate for fixed point arithmetic, communication channels which behave like pieces of wire (signals), and modules to break down a design into smaller parts.
- Release v2.0 (presently at version 2.3) provides an upgrade towards more powerful system-level design
  - signals built upon an underlying structure of channels, interfaces and ports
  - events as a primitive means of triggering behavior, together with a set of primitive channels, such as FIFO and mutex
  - dynamic processes
  - additional libraries to support a particular design methodology, e.g. Transaction-Level Modeling (TLM), SystemC Verification Library (SCV), Analog Mixed Signal (AMS)
- Release v3.0 will cover OS modeling to support development of embedded sw models

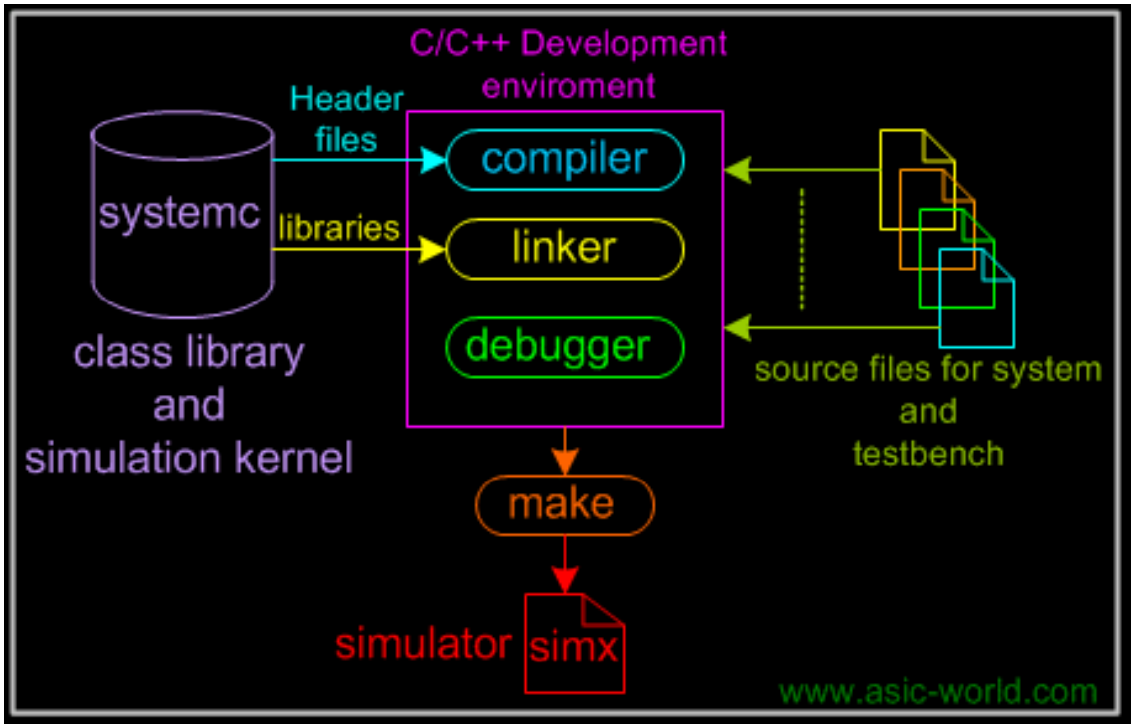
- Open SystemC Class Library supported by a group of companies forming the Open SystemC Initiative (OSCI).

SystemC Example: D-Flip Flop

```
1 // D-FF Code
2 #include "systemc.h"
3
4 SC_MODULE(d_ff) {
5   sc_in<bool> din;
6   sc_in<bool> clock;
7   sc_out<bool> dout;
8
9   void doit() {
10     dout = din;
11   };
12
13   SC_CTOR(d_ff) {
14     SC_METHOD(doit);
15     sensitive_pos << clock;
16   }
17 };
```



SystemC Design



Many engineers who want to learn SystemC, most often ask:

how much time it will take to learn SystemC?

The usual answer is:

It may not take more then one week, if you happen to know at least one programming language and a bit about OOPS

IP Modeling with SystemC

SystemC offers several hardware modeling constructs

- Modules and Hierarchy
- Hardware Data Types
- Methods and Threads
- Events, Sensitivity
- Interfaces and Channels

Modules and Hierarchy

- Modules are building blocks of SystemC designs, they are like modules in Verilog, class in C++
- Modules can be instantiated in another module
  - Each of the lower level modules instance can be referred through the complete hierarchy

Bit-Accurate SystemC Data Types

- SystemC supports all C++ data types
- Use SystemC data types (very close to Verilog HDL)
  - for hw modeling, e.g. bit-accuracy using **sc_logic** or **sc_lv**
  - consume more memory
  - slow down simulation
- **sc_fixed** and **sc_int** (and complex datatypes) allow modeling of complex math in DSP custom hardware or processor models without full floating-point capability.

SystemC Processes: Methods & Threads

- SC_METHOD and SC_THREADS form the backbone for hardware modeling
- SC_METHOD
  - are like functions in Verilog
  - do not consume time when they execute
- SC_THREAD
  - runs forever
  - consume time, like waiting for event (e.g. posedge of clock)

Events

- Events and sensitivity lists allow SystemC power to emulate hardware (concurrency)
- Events are implemented in SystemC using sc_event class
  - caused or triggered via sc_event member functions

Sensitivity Lists

SystemC has two types of sensitivity

- static sensitivity implemented by applying the SystemC sensitive command to an SC_METHOD, SC_THREAD, or SC_CTHREAD at elaboration time (within the constructor).
- Dynamic sensitivity allows a simulation process to change its sensitivity on the fly
  - SC_METHOD uses **next_trigger**, and
  - SC_THREAD uses a **next** command.

Interfaces, Channels and Ports

- In real hardware pins (ports) are used for communicating with external world.
- SystemC modules are usually interconnected using primitive or hierarchical channels.
- Both types of channels connect to modules via ports.

Hello World

(with a C++ member function)

```
1 // All systemc models include systemc.h header
2 #include "systemc.h"
3 // hello_world is the module name
4 SC_MODULE (hello_world) {
5   SC_CTOR (hello_world) {
6     // Nothing in constructor
7   }
8   void say_hello() {
9     //Print "Hello World" to the console.
10     cout << "Hello World.\n";
11   }
12 };
13 // sc_main  is top level function like C++ main
14 int sc_main(int argc, char* argv[]) {
15   hello_world hello("HELLO");
16   // Print the hello world
18   hello.say_hello();
18   return(0);
19 }
```

- systemc.h contains all SystemC macros & templates.
- We can use compiler preprocessor statements like #include, #define statements before module declaration
- Line 5 contains constructor SC_CTOR, which is similar to new in C++
- Line 8 contains function say_hello which prints the "Hello World" text to stdout when called.
- Alike C++, in SystemC if we have multiple lines within a block, we must use { and }.
- sc_main is top level function like main in C++.
- In sc_main we instantiate module hello_world.
- On line 17 we call the member function say_hello.

Executing Hello World

To compile and simulate the example in Linux

- g++ -I. -I$SYSTEMC_HOME/include -L. -L$SYSTEMC_HOME/lib-linux -o sim hello.cpp -lsystemc -lm
- ./sim

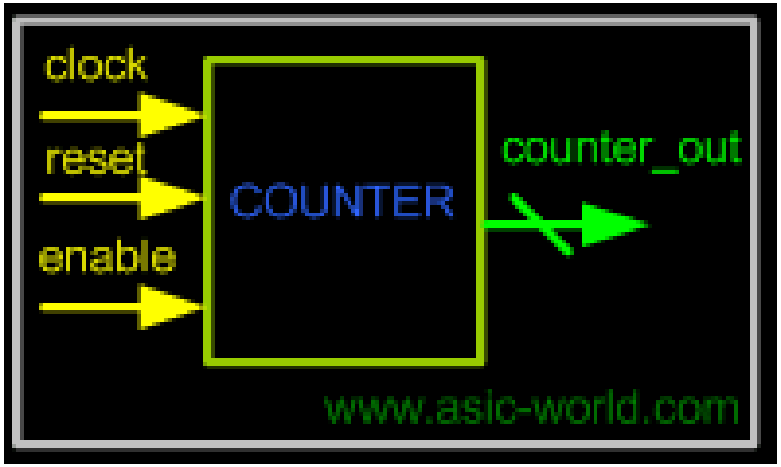
In the above command line options,

- $SYSTEMC_HOME is the SystemC installation directory

- Below is output of the execution of program hello.cpp

SystemC …
Hello World.

4-bit Counter



- 4-bit synchronous up counter
- Active high, asynchronous reset
- Active high enable

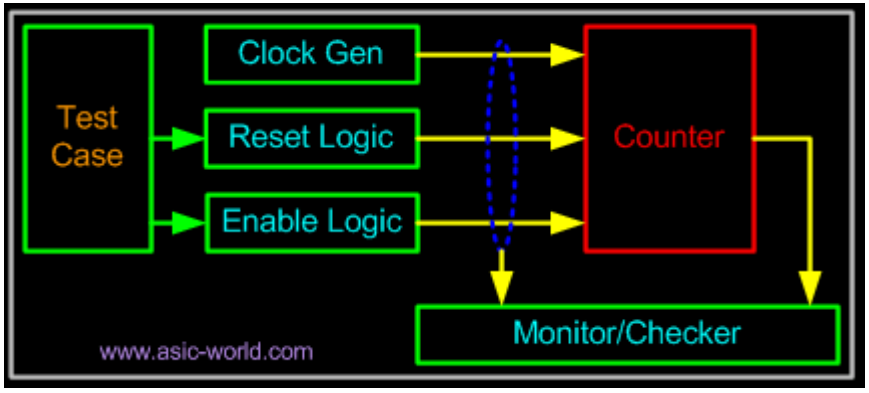
4-bit Counter

```
 9 #include "systemc.h"
10
11 SC_MODULE (first_counter) {
12   sc_in_clk     clock ;      // Clock input
13   sc_in<bool>   reset ;      // active high sync reset
       input
14   sc_in<bool>   enable;      // active high enable
15   sc_out<sc_uint<4> > counter_out; // 4 bit vector
16
17   //------------Local Variables Here---------------------
18   sc_uint<4>                 count;
19
20   //------------Code Starts Here-------------------------
21   // function implements actual counter logic
```

```
22   void incr_count () {
23     // At rising clock edge we check if reset is active
24     // If active, load the counter output with 4'b0000
25     if (reset.read() == 1) {
26       count =  0;
27       counter_out.write(count);
28     // If enable is active, then we increment the counter
29     } else if (enable.read() == 1) {
30       count = count + 1;
31       counter_out.write(count);
32       cout<<"@" << sc_time_stamp() <<" IncrCounter"
33         <<counter_out.read()<<endl;
34     }
35   } // End of function incr_count
```

```
41   SC_CTOR(first_counter) {
42     cout<<"Executing new"<<endl;
43     SC_METHOD(incr_count);
44     sensitive << reset;
45     sensitive << clock.pos();
46   }}
```

Counter Testbench



- Any digital circuit, no matter how complex, needs to be tested
- The same is done in SystemC
- Counter testbench consists of clock generator, reset control, enable control and monitor/checker logic
  - For the counter logic, we need to provide a clock and reset logic
  - Once the counter is out of reset, we toggle the enable input to the counter, and check the waveform to see if the counter is counting correctly

Example of Testbench Output

Below is the simple code of testbench without the monitor/checker logic.

```
SystemC
Executing new
@0 s Asserting reset

WARNING: Default time step is used for VCD tracing.
@10 ns De-Asserting reset

@40 ns Asserting Enable

@41 ns :: Incremented Counter 0
@43 ns :: Incremented Counter 1
@45 ns :: Incremented Counter 2
@47 ns :: Incremented Counter 3
@49 ns :: Incremented Counter 4
@51 ns :: Incremented Counter 5
@53 ns :: Incremented Counter 6
@55 ns :: Incremented Counter 7
@57 ns :: Incremented Counter 8
@59 ns :: Incremented Counter 9
@61 ns :: Incremented Counter 10
@63 ns :: Incremented Counter 11
@65 ns :: Incremented Counter 12
@67 ns :: Incremented Counter 13
@69 ns :: Incremented Counter 14
@71 ns :: Incremented Counter 15
@73 ns :: Incremented Counter 0
@75 ns :: Incremented Counter 1
@77 ns :: Incremented Counter 2
@79 ns :: Incremented Counter 3
@80 ns De-Asserting Enable

@80 ns Terminating simulation
```

Introduction to Data Types

- SystemC has many data types to support hw modeling, including fractional fixed-point
  - C++ Data Types
  - Bit Type
  - Logic Type
  - Arbitrary Width Bit Type
  - Arbitrary Width Logic Type
  - Signed Integer Type
  - Unsigned Integer Type
  - Arbitrary Precision Signed Integer Type
  - Arbitrary Precision Unsigned Integer Type
  - Resolved Types
  - User-defined Data Types
- Choice of data type depends on
  - synthesizability
  - level of abstraction & bit-accuracy requirements
  - simulation speed

Numeric Representation

SystemC provides a unified string based on C-like strings

- sc_string name ("0 base [sign] number e[+/-] exp]");

where

  - name : Name of the variable
  - base : **b** for binary, **o** for octal, **d** for decimal and **x** for hexa decimal
  - sign : **us** for unsigned, **sm** for signed magnitute, **csd** for canonical signed digit
  - number : Number is an integer in the indicated base
  - exp : The optional exponent exp is always specified using decimal

- Possible to convert between these formats

C++ Data Types

- SystemC supports all basic C++ data types listed below.
  - int
  - long int
  - unsigned int
  - unsigned long int
  - unsigned short int
  - short
  - float
  - char
  - bool
- Refer to C++ for these data types

sc_bit

Operator	Description	Usage
&	Bitwise AND	expr1 & expr2
|	Bitwise OR	expr1 | expr2
^	Bitwise XOR	expr1 ^ expr2
~	Bitwise NOT	~expr
&=	AND assignment	&= expr
\=	OR assignment	|= expr
^=	XOR assignment	^= expr
==	Equality	expr1 == expr2
!=	Inequality	expr1 != expr2

- sc_bit is the bit type data type, which can take two values '0' and '1‘; 1 represents true and 0 represents false
- Useful for modeling systems where Z (hi impedance) or X (unknown) values are not needed
- Several logical and comparison operators work on sc_bit

sc_bit

```
 1 #include <systemc.h>
 2
 3 int sc_main (int argc, char* argv[]) {
 4   // Declare the sc_bit
 5   sc_bit            enable;
 6   sc_bit            read_en;
 7   // Assign value to sc_bit
 8   enable = '1';
 9   cout <<"Value of enable  : " << enable << endl;
10   // Assign sc_bit to another sc_bit
11   read_en = enable;
12   cout <<"Value of read_en : " << read_en << endl;
13   return 1;
14 }
```

SystemC …
Value of enable : 1 Value of read_en : 1

sc_logic

Operator	Description	Usage
&	Bitwise AND	expr1 & expr2
|	Bitwise OR	expr1 | expr2
^	Bitwise XOR	expr1 ^ expr2
~	Bitwise NOT	~expr
&=	AND assignment	variable &= expr
|=	OR assignment	variable |= expr
^=	XOR assignment	variable ^= expr
=	Equality	expr1 = expr2
!=	Inequality	expr1 != expr2

- The sc_logic type can hold:
  - '0': false
  - '1': true
  - 'X' or 'x': unknown or indeterminate value
  - 'Z' or 'z': high-impedance or floating value
- There are a number of logical and comparison operators that work with sc_bit.
- sc_logic and sc_bit type can be **assigned or compared** to each other.
  - sc_logic has predefined cast values in such assignments.

sc_logic

```
1 #include <systemc.h>
3 int sc_main (int argc, char* argv[]) {
4   // Declare the sc_logic
5   sc_logic       read_en;
6   sc_logic       pad;
7   sc_logic       enable;
8   sc_bit         no_x_z;
9   // Assign values
10   pad = 'z';
11   cout <<"pad: " << pad << endl;
12   enable = '0';
13   cout <<"enable: " << enable << endl;
14   read_en = ~enable;
15   cout <<"read_en: " << read_en << endl;
16   // Logic operation
17   if (pad == '1') {
18     cout <<"Pad is 1"<< endl;
19   } else {
20     cout <<"Pad is not 1"<< endl;
21   }
22   // Assign to sc_bit type
23   no_x_z = enable; // Assign 0/1 value
24   cout <<"no_x_z : " << no_x_z  << endl;
25   no_x_z = pad; // Assign Z
26   cout <<"no_x_z : " << no_x_z  << endl;
27   // Assign with cast values
28   pad = sc_logic ('x');
29   cout <<"pad     : " << pad << endl;
31   return 1;
32 }
```

```
SystemC …
Value of pad     : Z
Value of enable  : 0
Value of read_en : 1
```

```
Pad is not 1
Value of no_x_z  : 0
Warning: (W211) sc_logic value 'Z' cannot be converted to bool
Value of no_x_z  : 1
Value of pad     : X
```

Assignments: sc_bit & sc_logic

```
1 #include <systemc.h>
 2
 3 int sc_main (int argc, char* argv[]) {
 4   // Declare the sc_bit
 5   sc_bit x;
 6   // Declare the sc_logic
 7   sc_logic y, z;
 8   // Assign values to x, y, z
 9   x = '1';
10    y = '1';
11    z = 'x';
12    // sc_bit and sc_logic
13    if (x == y) {
14      cout <<"x = y " << endl;
15    } else {
16      x = y; // Assign sc_logic to sc_bit
17      cout <<"x != y " << endl;
18      cout<<"New Value of x : "<< x << endl;
19    }
20   // sc_logic and sc_logic
21   if (y  ! = z) {
22     y = z; // Assign sc_bit to sc_logic
23     cout <<"y != z " << endl;
24     cout<<"New Value of y : "<< y << endl;
25   }
26   // sc_logic and character literal
27   if (y == 'x') {
28     cout<<"y equal to value 'X'"<< endl;
29   }
30
31   return 1;
32 }
```

SystemC …
x = y
y != z
New Value of y : X
y equal to value 'x'

Logical Operations on X & Z

- Logical operation on 4 states (x,z,0,1) on various types of operators has different effects.
  - If you know Verilog then you must already know how results differ when one of the inputs is X or Z.
- We consider the effect for the following operators
  - & - AND Operator
  - | - OR Operator
  - ~ - NOT Operator

Logical Operations on X & Z

AND Operator

For AND operator, when any of the inputs is 0, then output is 0. When both the inputs is 1 then output is 1, for other cases output is X.

& (AND)	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

OR Operator

For OR operator, when any of the inputs is 1, then output is 1. When both the inputs is 0 then output is 0, for other cases output is X.

| (OR)	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

NOT Operator

For NOT operator, when the input is 1, then output is 0. When the input is 0 then output is 1, for other cases output is X.

~ (NOT)	0	1	X	Z
	1	0	X	X

Arbitrary-Width Bit Type: sc_bv

- sc_bit is one bit wide
- SystemC defines sc_bv (Bit Vector) for multi-bit vectors
  - vector width is specified as an integer.
  - A width of M sets the vector size and direction to be M-1 down to 0 with the M^{th} bit being the most significant bit.
    - The rightmost index of the vector is 0 and is also the least significant bit.
- In sc_bv assignments, if the variable has
  - smaller size (width), the rest of the bits are extended with 0.
  - large size (width), then they extra bits are truncated.
- Also, sc_bv defines bit reduction & range selection operators
  - not available on port and signals.

Arbitrary Width Bit Type: sc_bv

Operator	Description	Usage
&	Bitwise AND	expr1 & expr2
|	Bitwise OR	expr1 | expr2
^	Bitwise XOR	expr1 ^ expr2
~	Bitwise NOT	~expr
&=	AND assignment	variable &= expr
|=	OR assignment	variable |= expr
^=	XOR assignment	variable ^= expr
=	Equality	expr1 = expr2
!=	Inequality	expr1 != expr2
[]	Bit selection	variable [index]
(,)	Concatenation	(expr1, expr2, expr3)

Methods in Arbitrary Width Bit Type

Methods	Description	Usage
range()	Range selection	variable.range(indexl, index2)
and_reduce()	Reduction AND	variable.and_reduce()
or_reduce()	Reduction OR	variable.or_reduce()
xor_reduce()	Reduction XOR	variable.xor_reduce()

Example of sc_bv

```
1 #include <systemc.h>
2
3 int sc_main (int argc, char* argv[]) {
4   sc_bv<8>  data_bus  ;
5   sc_bv<16> addr_bus  ;
6   sc_bit    parity    ;
7   // Assign value to sc_bv
8   data_bus = "00001011";
9   cout <<"data_bus : " << data_bus << endl;
10   // Use range operator
11   addr_bus.range(7,0) = data_bus;
12   cout <<"addr_bus : " << addr_bus << endl;
13   // Assign reverse to addr bus using range
     operator
14   addr_bus.range(0,7) = data_bus;
15   cout <<"addr_bus : " << addr_bus << endl;
16   // Use bit select to set the value
17   addr_bus[10] = "1";
18   cout <<"addr_bus : " << addr_bus << endl;
19   // Use reduction operator
20   parity  = data_bus.xor_reduce();
21   cout <<"parity   : " << parity << endl;
22
23   return 1;
24 }
```

```
SystemC …
data_bus : 00001011
addr_bus : 0000000000001011
```

```
addr_bus : 0000000011010000
addr_bus : 0000010011010000
parity   : 1
```

Arbitrary Width Logic Type: sc_lv

- sc_logic is one bit wide
- SystemC defines sc_lv (Logic Vector) for multi-bit vectors
  - vector width is specified as an integer.
  - A width of M sets the vector size and direction to be M-1 down to 0 with the M^{th} bit being the most significant bit.
    - The rightmost index of the vector is 0 and is also the least significant bit.
- In sc_lv assignments, if the variable has
  - smaller size (width), the rest of the bits are extended with 0.
  - large size (width), then they extra bits are truncated.
- Also, sc_lv defines bit reduction & range selection operators
  - not available on port and signals.
- No arithmetic operations on sc_lv types, instead they must be performed on sc_int or sc_uint and then assigned to sc_lv.

Arbitrary Width Logic Type: sc_lv

```
1 #include <systemc.h>
2
3 int sc_main (int argc, char* argv[]) {
4   sc_lv<8>  data_bus  (sc_logic ('z')); // All bits
    are Z
5   sc_lv<16> addr_bus  ; // All bits are X
6   sc_logic  parity    ;
7   // Print Default value of data_bus
8   cout <<"data_bus : " << data_bus << endl;
9   // Assign value to sc_bv
10   data_bus = "00001011";
11   cout <<"data_bus : " << data_bus << endl;
12   // Use range operator
13   addr_bus.range(7,0) = data_bus;
14   cout <<"addr_bus : " << addr_bus << endl;
15   // Assign reverse to addr bus using range
    operator
16   addr_bus.range(0,7) = data_bus;
17   cout <<"addr_bus : " << addr_bus << endl;
18   // Use bit select to set the value
19   addr_bus[10] = "1";
20   cout <<"addr_bus : " << addr_bus << endl;
21   // Use reduction operator
22   parity  = data_bus.xor_reduce();
23   cout <<"parity   : " << parity << endl;
24
25   return 1;
26 }
```

```
SystemC …
 Value of data_bus : ZZZZZZZZ
 Value of data_bus : 00001011
 Value of addr_bus : XXXXXXXX00001011
 Value of addr_bus : XXXXXXXX11010000
 Value of addr_bus : XXXXX1XX11010000
 Value of parity   : 1
```

sc_int

- sc_int is fixed precision signed integer of size 64 bits
  - sc_int values stored in 2's compliment
  - sign bit for a N-bit wide sc_int is stored at N-1 bit position
- Integer types have a very rich set of 64-bit operators
  - result size is determined at object declaration

Operations of sc_int

Basic Operators

Operator	Description	Usage
&	Bitwise AND	expr1 & expr2
1	Bitwise OR	expr1 | expr2
^	Bitwise XOR	expr1 ^ expr2
~	Bitwise NOT	~expr
+	Addition	expr1 + expr2
-	Minus	expr1 - expr2
*	Multiply	expr1 * expr2
/	Divide	expr1 / expr2
%	Modulus	expr1 % expr2
>>	Arithmetic right shift	expr >> constant
<<	Arithmetic left shift	expr << constant

Compound Assignment

Operator	Description	Usage
+=	+ assignment	variable += expr
-=	- assignment	variable -= expr
*=	* assignment	variable *= expr
\=	\ assignment	variable \= expr
%=	% assignment	variable %= expr
&=	AND assignment	variable &= expr
|=	OR assignment	variable |= expr
^=	XOR assignment	variable ^= expr

Logic Operators

Operator	Description	Usage
=	Equality	expr1 = expr2
!=	Inequality	expr1 != expr2
<	Less than	expr1 < expr2
<=	Less than or equal to	expr1 <= expr2
>	Greater than	expr1 > expr2
>=	Greater than or equal to	expr1 >= expr2
++	AutoIncrement	variable ++
--	AutoDecrement	variable --

Bit Operators

Operator	Description	Usage
[]	Bit selection	variable [index]
(,)	Concatenation	(expr1, expr2,...)
range()	Range Selection	variable.range (index1,index2)

- Bitwise operators work on operands
- not(~) operator will invert all bits
- shift operators will shift left(<<) or right(>>) an operand by the specified number of bits.
- sc_int can be used with C++ integer types without restriction

Example of sc_int

```
1 #include <systemc.h>
2
3 int sc_main (int argc, char* argv[]) {
4   sc_int<1>  bit_size    = 0;
5   sc_int<4>  nibble_size = 1;
6   sc_int<8>  byte_size   = 2;
7   sc_int<32> dword_size  = 3;
8   //sc_int<128> addr; sc_int cannot be > 64
9   // Perform auto increment
10   dword_size ++;
11   cout <<"dword_size : " << dword_size <<
     endl;
12   // Terse method addition
13   byte_size += nibble_size;
14   cout <<"byte_size  : " << byte_size << endl;
15   // Bit selection
16   bit_size = dword_size[2];
17   cout <<"bit_size   : " << bit_size << endl;
18   // Range: can not assign out of range
19   nibble_size = dword_size.range(4,1); //
20   cout <<"nibble_size: " << nibble_size <<
     endl;
21   // Concatenate
22   dword_size =
     (byte_size,byte_size,byte_size,byte_size);
23   cout <<"dword_size : " << dword_size <<
     endl;
24   return 1;
25 }
```

```
SystemC …
dword_size : 4
byte_size  : 3
```

```
bit_size   : -1
nibble_size: 2
dword_size : 50529027
```

sc_uint

- sc_uint is fixed precision unsigned integer of size 64 bits
  - sc_int values stored in 2's compliment
  - sign bit for a N-bit wide sc_int is stored at N-1 bit position
- Integer types have a very rich set of 64-bit operators
  - result size is determined at object declaration
- In assignments, sc_uint operands converted to/from sc_int
  - When assigning an integer to an unsigned operand, the integer value in 2's complement form is interpreted as an unsigned number.
  - When assigning an unsigned to a signed operand, the unsigned is expanded to a 64 bit unsigned number and then truncated to get the signed value.
- sc_uint has rich operators (alike sc_int)

Example of sc_uint

```
1 #include <systemc.h>
2
3 int sc_main (int argc, char* argv[]) {
4   sc_uint<1>  bit_size    = 0;
5   sc_uint<4>  nibble_size = 1;
6   sc_uint<8>  byte_size   = 2;
7   sc_uint<32> dword_size  = 3;
8   //sc_int<128> addr; sc_int can not be more
      then 64
9   // Perform auto increment
10   dword_size ++;
11   cout <<"dword_size : " << dword_size <<
      endl;
12   // Terse method addition
13   byte_size += nibble_size;
14   cout <<"byte_size  : " << byte_size << endl;
15   // Bit selection
16   bit_size = dword_size[2];
17   cout <<"bit_size   : " << bit_size << endl;
18   // Range: cannot assign out of range
19   nibble_size = dword_size.range(4,1);
20   cout <<"nibble_size: " << nibble_size << endl;
21   // Concatenated
22   dword_size =
     (byte_size,byte_size,byte_size,byte_size);
23   cout <<"dword_size : " << dword_size << endl;
25   return 1;
26 }
```

```
SystemC …
dword_size : 4
byte_size  : 3
bit_size   : 1
nibble_size: 2
dword_size : 50529027
```

sc_bigint

- sc_bigint for arbitrary sized, signed integers > 64 bits
  - sc_int does not work
  - sc_bigint is a 2's complement signed integer of any size
  - MAX_NBITS in sc_constants.h limits max size of bits to 512
  - may be limited only by underlying system
- Rich operators (arithmetic and other) use arbitrary precision
  - refer to sc_int for details

Example of sc_bigint

```
1 #include <systemc.h>
 2
 3 int sc_main (int argc, char* argv[]) {
 4   sc_bigint<128> large_size ;
 5   // Shift operator
 6   large_size =  1000 << 1;
 7   cout <<"large_size : " << large_size << endl;
 8
 9   return 1;
10 }
```

```
SystemC …
large_size : 2000
```

sc_biguint

- sc_biguint for arbitrary sized, unsigned integers > 64 bits
  - sc_uint does not work
  - sc_bigint is a 2's complement signed integer of any size
  - MAX_NBITS in sc_constants.h limits max size of bits to 512
  - may be limited only by underlying system
- Rich operators (arithmetic and other) use arbitrary precision
  - refer to sc_int for details

sc_biguint

```
1 #include <systemc.h>
2
3 int sc_main (int argc, char* argv[]) {
4   sc_biguint<128> a  = 30000;
5   sc_biguint<128> b  = 20000;
6   sc_biguint<256> c  = 0;
7   // multiplication operator
8   c = a * b;
9   cout <<"Value of c : " << c << endl;
10
11   return 1;
12 }
```

```
SystemC …
Value of c : 600000000
```

Modules

- Modules are basic, manageable building blocks
  - similar to module in Verilog and Entity in VHDL
  - help partition complex designs among different designers in a group
  - hide internal data representation and algorithms from other modules
  - use public interfaces to other modules, so easier to change/maintain the system
- A module contains
  - constructor
  - internal variables
  - ports
  - internal methods/functions to work on the ports

sc_module

Modules are easily declared with SystemC keyword sc_module:

SC_MODULE("module_name") {
// module body
}

where

- sc_module: macro or reserve word
- module_name : Any valid module name

Alternatively, in pure C++ syntax, we define a module as follows:

class module_name : sc_module {
// Module body
}

Example of sc_module

```
1 // All systemc code should include systemc.h file
2 #include "systemc.h"
3 // SC_MODULE is macro, hello_world is module name
4 SC_MODULE (hello_world) {
5   // Body of module hello_world
6 };
```

Module Ports

- Module ports pass data to/from module processes to the external world, as in Verilog and VHDL.
- Port has
  - Name as any valid variable name
  - Direction (syntax sc_direction type variable)
    - sc_in: Input Ports
    - sc_out : Output Ports
    - sc_inout : Bi-direction Ports
  - data type as any C++, SystemC, or user-defined data type

Module sc_port

```
1 #include "systemc.h"
2
3 SC_MODULE (first_counter) {
4   sc_in_clk     clock ;      // Clock input of the design
5   sc_in<bool>   reset ;      // active high, synchronous Reset input
6   sc_in<bool>   enable;      // Active high enable signal for counter
7   sc_out<sc_uint<4> > counter_out; // 4 bit vector output of counter
8
9   // Rest of body
10 }
```

Module Signals

- For communicating outside the module we use ports
- Signals are used
  - for communicating within a SystemC module we use signals
  - for connecting two module ports in parent module we use signals
    - If child modules A and B are used in parent module C, signals connect together ports of module A and B
- Signals are defined as: **sc_signal type variable**,

where

  - sc_signal : Reserved word
  - type : any legal data type
  - variable : a valid variable name

Signals are like wires in Verilog

Module Signals

```
1 #include "systemc.h"
2
3 SC_MODULE (counter) {
4   sc_signal <bool>        reset ;
5   sc_signal <bool>        enable;
6   sc_signal <sc_uint<4> > counter_out;
7
8   // Rest of body
9 }
```

Module Instantiation & Port Connection

- Instantiating modules in SystemC as in Verilog
- SystemC allows two ways of connecting ports
  - by Name
  - by Position

Module Instantiation: By Name

```
1 #include "systemc.h"
2 #include "first_counter.cpp"
3
4 int sc_main (int argc, char* argv[]) {
5   sc_signal<bool>   clock;
6   sc_signal<bool>   reset;
7   sc_signal<bool>   enable;
8   sc_signal<sc_uint<4> > counter_out;
9   int i = 0;
10   // Connect the DUT
11   first_counter counter("COUNTER");
12     // Here ports are connected by name
13     counter.enable(enable);
14     counter.reset(reset);
15     counter.clock(clock);
16     counter.counter_out(counter_out);
17
18   // Rest of the body
19
20   return 0;// Terminate simulation
21  }
```

Here

- order is not important
- top module names match with leaf module

Module Instantiation: By Position

```
1 #include "systemc.h"
2 #include "first_counter.cpp"
3
4 int sc_main (int argc, char* argv[]) {
5   sc_signal<bool>   clock;
6   sc_signal<bool>   reset;
7   sc_signal<bool>   enable;
8   sc_signal<sc_uint<4> > counter_out;
9   int i = 0;
10  // Connect the DUT
11  first_counter counter("cntr");
12    // ports connected by position
13  counter(clock,reset,enable,counter_out);
14  // Rest of the testbench …
15  return 0;// Terminate simulation
18 }
```

Here

- order should match correctly.
- not good idea to connect ports by position
- problem in debug to locate the port when port is added or deleted

Internal Variables

```
1 #include "systemc.h"
2
3 SC_MODULE (local_variable) {
4   sc_in_clk     clock ;     // Clock input of the
    design
5   sc_in<bool>   reset ;   // active high sync reset
6   sc_in<bool>   enable; // Active high enable
7   sc_out<sc_uint<4> > counter_out; // 4 bit
    vector output of the counter
8
9   //------------Local Variables Here----------------
    -----
10   sc_uint<4>           count;
11
12   //------------Code Starts Here--------------------
    -----
13   // Below function implements actual counter
    logic
14   void incr_count () {
15     // Body of the function
16   } // End of function incr_count
17
18   // Constructor for the counter
19   SC_CTOR(local_variable) {
20     cout<<"Executing new"<<endl;
21     SC_METHOD(incr_count);
22     sensitive << reset;
23     sensitive << clock.pos();
24   } // End of Constructor
25
26 }; // End of Module counter
```

Processes

- Actual functionality of SystemC module
  - same as **always block** of Verilog.
  - level sensitive to model combinational logic or
  - edge sensitive logic to module sequential logic
- Processes
  - can be made to run forever (alike Verilog) – call threads in SystemC
  - or run whenever an event occurs (e.g. posedge of clock) – call method

- **Note:** sensitive list signals which trigger processes are **ports** – not signals or local variables

Example of sc_method

```
1 #include "systemc.h"
2
3 SC_MODULE(dff) {
4   sc_in  <bool> din; // Data input of FF
5   sc_in  <bool> clk; // Clock input of FF
6   sc_out <bool> out; // Q output of FF
7
8   void implement();  // Process that implements DFF
9
10   SC_CTOR(dff) {
11     SC_METHOD(implement);
12     sensitive_pos << clk; // Call implements at every pos edge of clk
13   }
14 };
```

Constructor of sc_module

- The module constructor
  - creates/initializes the internal data structures
  - passes the module & hierarchy instance name to the constructor at instantiation (creation) time
    - this helps identify the module when errors occur or when reporting info
    - the only difference between C++ and SystemC constructor

Example of sc_module

```
1 #include "systemc.h"
 2
 3 SC_MODULE(dff) {
 4   sc_in  <bool> din; // Data input of FF
 5   sc_in  <bool> clk; // Clock input of FF
 6   sc_out <bool> dout;// Q output of FF
 7
 8   void implement() {  // Process that implements DFF
 9     dout = din;
10   }
11
12   SC_CTOR(dff) {
13     SC_METHOD(implement); // Contains a process named implement
14     sensitive_pos << clk; // Execute implements() at every posedge of clk
15   }
16 };
```

Processes

- There are three types of processes in SystemC.
- Processes
  - are not hierarchical, so no process will call another process directly.
  - can call methods and functions that are not processes
  - have sensitivity lists, i.e. a list of signals that cause the process to be invoked, whenever the value of a signal in this list changes
  - cause other processes to execute by assigning new values to signals in the sensitivity list of the other process

- Methods
  - Edge Sensitive
  - Level Sensitive
- Threads
- Clocked Threads

sc_method

- Methods behaves like a function, when called it gets started and executes and returns execution back to calling mechanism.
- A method is called whenever the event in the sensitivity list changes.
  - Like **always block** in Verilog.
- Triggering event in sensitive list can be either edge sensitive or level sensitive.

- **Note :** Sensitive list signals which trigger a process can be a signal or local variable or port.

- The input signals that cause the process to reactivate are specified by the sensitivity list. The sensitivity list is specified in the module constructor

Example of sc_method

```
 9 #include "systemc.h"
10
11 SC_MODULE (first_counter) {
12   sc_in_clk     clock ;      // Clock input of the design
13   sc_in<bool>   reset ;      // active high, synchronous Reset input
14   sc_in<bool>   enable;      // Active high enable signal for
       counter
15   sc_out<sc_uint<4> > counter_out; // 4 bit vector output of the
       counter
16
17   //------------Local Variables Here---------------------
18   sc_uint<4>count;
19
20   //------------Code Starts Here-------------------------
21   // Below function implements actual counter logic
22   void incr_count () {
23     // At every rising edge of clock we check if reset is active
24     // If active, we load the counter output with 4'b0000
25     if (reset.read() == 1) {
26       count =  0;
27       counter_out.write(count);
28     // If enable is active, then we increment the counter
29     } else if (enable.read() == 1) {
30       count = count + 1;
31       counter_out.write(count);
32     }
33   } // End of function incr_count
34
35   // Below functions prints value of count when ever it changes
36   void print_count () {
37     cout<<"@" << sc_time_stamp() <<
38       " :: Counter Value "<<counter_out.read()<<endl;
39   }
40
41   // Constructor for the counter
42   // Since this counter is a positive edge trigged one,
43   // We trigger the below block with respect to positive
44   // edge of the clock and also when ever reset changes state
45   SC_CTOR(first_counter) {
46     // Edge and level sensitive
47     SC_METHOD(incr_count);
48     sensitive << reset;
49     sensitive << clock.pos();
50     // Level Sensitive method
51     SC_METHOD(print_count);
52     sensitive << counter_out;
53   } }
```

sc_thread

- sc_thread keeps executing or waiting for some event to occur
- Conditions that enable this processes can be event triggered, like posedge of clock
- Thread Process can be suspended and reactivated
  - Thread Process can contain **wait()** functions that suspend process execution until an event occurs on one of the signals the process is sensitive to
  - An event will reactivate the thread process from the statement the process was last suspended
  - The process will continue to execute until the next **wait()**

Example of sc_thread

```
1 //------------------------------------------------------
2 // This is my second Systemc Example
3 // Design Name : first_counter
4 // File Name : first_counter.cpp
5 // Function : This is a 4 bit up-counter with
6 // Synchronous active high reset and
7 // with active high enable signal
8 //------------------------------------------------------
9 #include "systemc.h"
10
11 SC_MODULE (first_counter) {
12   sc_in_clk     clock ;      // Clock input of the design
13   sc_in<bool>   reset ;      // active high, synchronous Reset input
14   sc_in<bool>   enable;      // Active high enable signal for counter
15   sc_out<sc_uint<4> > counter_out; // 4 bit vector output of the counter
16
17   //------------Local Variables Here---------------------
18   sc_uint<4>              count;
19
20   //------------Code Starts Here-------------------------
21   // Below function implements actual counter logic
22   void incr_count () {
23     // For threads, we need to have while true loop
24     while (true) {
25       // Wait for the event in sensitivity list to occure
26       // In this example - positive edge of clock
27       wait();
28       if (reset.read() == 1) {
29         count =  0;
30         counter_out.write(count);
31       // If enable is active, then we increment the counter
32       } else if (enable.read() == 1) {
33         count = count + 1;
34         counter_out.write(count);
35       }
36     }
37   } // End of function incr_count
38
39   // Below functions prints value of count when ever it changes
40   void print_count () {
41     while (true) {
42       wait();
43       cout<<"@" << sc_time_stamp() <<
44         " :: Counter Value "<<counter_out.read()<<endl;
45     }
46   }
47
48   // Constructor for the counter
49   // Since this counter is a positive edge trigged one,
50   // We trigger the below block with respect to positive
51   // edge of the clock
52   SC_CTOR(first_counter) {
53     // Edge sensitive to clock
54     SC_THREAD(incr_count);
55     sensitive << clock.pos();
56     // Level Sensitive to change in counter output
57     SC_THREAD(print_count);
58     sensitive << counter_out;
59   } // End of Constructor
60
61 }; // End of Module counter
```

sc_cthread

- SC_CTHREAD
  - **does not have** a separate sensitivity list
  - it specifies a clock object that triggers the process, i.e. SC_CTHREAD process is activated whenever the specified clock edge occurs
- process incr_count will execute on every positive edge of the clock.
-
- **Note :**SC_CTHREAD can have only one bit wide ports as trigger.

Example of sc_cthread

```
 1 //------------------------------------------------------
 2 // This is my second Systemc Example
 3 // Design Name : first_counter
 4 // File Name : first_counter.cpp
 5 // Function : This is a 4 bit up-counter with
 6 // Synchronous active high reset and
 7 // with active high enable signal
 8 //------------------------------------------------------
 9 #include "systemc.h"
10
11 SC_MODULE (first_counter) {
12   sc_in_clk     clock ;      // Clock input of the design
13   sc_in<bool>   reset ;      // active high, synchronous Reset input
14   sc_in<bool>   enable;      // Active high enable signal for counter
15   sc_out<sc_uint<4> > counter_out; // 4 bit vector output of the counter
16
17   //------------Local Variables Here---------------------
18   sc_uint<4>              count;
19
20   //------------Code Starts Here-------------------------
21   // Below function implements actual counter logic
22   void incr_count () {
23     // For threads, we need to have while true loop
24     while (true) {
25       // Wait for the event in sensitivity list to occure
26       // In this example - positive edge of clock
27       wait();
28       if (reset.read() == 1) {
29         count =  0;
30         counter_out.write(count);
31       // If enable is active, then we increment the counter
32       } else if (enable.read() == 1) {
33         count = count + 1;
34         counter_out.write(count);
35       }
36     }
37   } // End of function incr_count
38
39   // Below functions prints value of count when ever it changes
40   void print_count () {
41     while (true) {
42       wait();
43       cout<<"@" << sc_time_stamp() <<
44         " :: Counter Value "<<counter_out.read()<<endl;
45     }
46   }
47
48   // Constructor for the counter
49   // Since this counter is a positive edge trigged one,
50   // We trigger the below block with respect to positive
51   // edge of the clock
52   SC_CTOR(first_counter) {
53     // cthreads require to have thread name and triggering
54     // event to passed as clock object
55     SC_CTHREAD(incr_count, clock.pos());
56     // Level Sensitive to change in counter output
57     SC_THREAD(print_count);
58       sensitive << counter_out;
59   } // End of Constructor
60
61 }; // End of Module counter
```

dont_initialize

- Process and threads (not cthreads) execute automatically in constructor, even if event in sensitivity list does not occur
- Use dont_initialize() to prevent unintentional execution

Example of dont_initialize

```
1 #include "systemc.h"
 2
 3 SC_MODULE (tff_sync_reset) {
 4   sc_in    <bool> data, clk, reset  ;
 5   sc_out   <bool> q;
 6
 7   bool q_l ;
 8
 9   void tff () {
10     if (reset.read()) {
11       q_l = 0;
12     } else if (data.read()) {
13       q_l =  ! q_l;
14     }
15     q.write(q_l);
16   }
17
18   SC_CTOR(tff_sync_reset) {
19     SC_METHOD (tff);
20       dont_initialize();
21       sensitive << clk.pos();
22   }
23
24 };
25
26 SC_MODULE (tff_tb) {
27   sc_in<bool> clk;
28
29   sc_signal <bool> data, reset  ;
30   sc_signal   <bool> q;
31   tff_sync_reset *dut;
32
```

```
33   void do_test() {
34     cout << "@" << sc_time_stamp() <<" Starting test"<<endl;
35     wait(); cout << "@" << sc_time_stamp() <<" Asserting reset"<<endl;
37     reset = true;  wait(4); cout << "@" << sc_time_stamp() <<" De-Asserting reset"<<endl;
40     reset = false;  wait(3); cout << "@" << sc_time_stamp() <<" Asserting Data input"<<endl;
43     data = true; wait(3); data = false;
46     cout << "@" << sc_time_stamp() <<" De-Asserting Data input"<<endl;
47     wait(3);
48     cout << "@" << sc_time_stamp() <<" Terminating simulation"<<endl;
49     sc_stop();
50   }
51
52   SC_CTOR(tff_tb) {
53     dut = new tff_sync_reset ("TFF");
54     dut->clk    (clk);
55     dut->reset  (reset);
56     dut->data   (data);
57     dut->q      (q);
58     SC_THREAD (do_test);
59       dont_initialize();
60       sensitive << clk.pos();
61   }
62 };
63
64 int sc_main (int argc, char* argv[]) {
65   sc_clock clock ("my_clock",1,0.5);
66
67   tff_tb  object("TFF_TB");
68     object.clk (clock.signal());
69   sc_trace_file *wf = sc_create_vcd_trace_file("tff");
70     sc_trace(wf, object.clk, "clk");
71     sc_trace(wf, object.reset, "reset");
72     sc_trace(wf, object.data, "data");
73     sc_trace(wf, object.q, "q");
74
75   sc_start(0);
76   sc_start();
77   sc_close_vcd_trace_file(wf);
78   return 0;// Terminate simulation
79 }
```

```
@0 s Starting test
WARNING: Default time step is used for VCD tracing.
@1 ns Asserting reset
@5 ns De-Asserting reset
@8 ns Asserting Data input
@11 ns De-Asserting Data input
@14 ns Terminating simulation
SystemC: simulation stopped by user.
```

Accessing Ports and Signals

- Last, we saw how to declare different types of ports
- Not a good idea to access ports directly for reading or writing.
- Access ports using
  - port_name.write('value') for writing value to port
  - port_name.read() for reading value from port
- It is not always possible to use same data types of ports or signals inside processes
  - write() and read() methods do automatic type conversion from other data types to port data types

Example of Accessing Ports and Signals

```
1 #include <systemc.h>
2
3 SC_MODULE(ports_access) {
4   sc_in<sc_bit> a;
5   sc_in<sc_bit> b;
6   sc_in<bool>   en;
7   sc_out<sc_lv<2> >  out;
8
9   // Method to manipulate output
10  void body () {
11    // Ports should use read() method to read values
12    if (en.read() == 1) {
13      // Should use write() method of write values
14      out.write(a.read() + b.read());
15    }
16  }
17  // Method to monitor ports
18  void monitor () {
19    cout << "@" << sc_time_stamp() <<" a : " << a
20      << " b : " << b << " en : " << " out : "
21      << out.read() <<endl;
22  }
23
24  SC_CTOR(ports_access) {
25    SC_METHOD(body);
26      sensitive << a << b << en;
27    SC_METHOD(monitor);
28      sensitive << a << b << en << out;
29  }
30
31 };
32
33 // Testbench to generate test vectors
34 int sc_main (int argc, char* argv[]) {
35   sc_signal <sc_bit> a;
36   sc_signal <sc_bit> b;
37   sc_signal <bool>   en;
38   sc_signal <sc_lv<2> >  out;
39
40   ports_access prt_ac("PORT_ACCESS");
41     prt_ac.a(a);
42     prt_ac.b(b);
43     prt_ac.en(en);
44     prt_ac.out(out);
45
46   sc_start(0);
47   // Open VCD file
48   sc_trace_file *wf = sc_create_vcd_trace_file("ports_access");
49     sc_trace(wf, a, "a");
50     sc_trace(wf, b, "b");
51     sc_trace(wf, en, "en");
52     sc_trace(wf, out, "out");
53   // Start the testing here
54   sc_start(1);
55   a = sc_bit('0');
56   b = sc_bit('0');
57   en = 1;
58   sc_start(1);
59   a = sc_bit('1');
60   sc_start(1);
61   b = sc_bit('1');
62   sc_start(2);
63
64   sc_close_vcd_trace_file(wf);
65   return 0;// Terminate simulation
66 }
```

```
SystemC …
@0 s a : 0 b : 0 en :  out : XX
 WARNING: Default time step is used for VCD tracing.
 @1 ns a : 0 b : 0 en :  out : XX
 @1 ns a : 0 b : 0 en :  out : 00
 @2 ns a : 1 b : 0 en :  out : 00
 @2 ns a : 1 b : 0 en :  out : 01
 @3 ns a : 1 b : 1 en :  out : 01
 @3 ns a : 1 b : 1 en :  out : 10
```

Array of Ports and Signals

- SystemC allows to declare array of ports and signals
- This is useful when we need several similar ports and signals

Example of Array of Ports and Signals

```
1 #include <systemc.h>
2
3 SC_MODULE(ports_arrays) {
4   sc_in<sc_uint<2> > a[4];
5   sc_in<sc_uint<2> > b[4];
6   sc_out<sc_uint<3> > o[4];
7
8   void body () {
9     int i;
10    for (i=0; i < 4; i ++) {
11      o[i].write(a[i].read() + b[i].read());
12    }
13  }
14
15  SC_CTOR(ports_arrays) {
16    int j;
17    SC_METHOD(body);
18      for (j=0; j<4; j++) {
19        sensitive << a[j] << b[j];
20      }
21  }
22 };
23
24 // Testbench to generate test vectors
25 int sc_main (int argc, char* argv[]) {
26   sc_signal<sc_uint<2> > a[4];
27   sc_signal<sc_uint<2> > b[4];
28   sc_signal<sc_uint<3> > o[4];
29
30   int z;
31
32   ports_arrays prt_ar("PORT_ARRAY");
33     for (z=0; z<4; z++) {
34       prt_ar.a[z](a[z]);
35       prt_ar.b[z](b[z]);
36       prt_ar.o[z](o[z]);
37     }
```

```
38   sc_start(0);
39   // Open VCD file
40   sc_trace_file *wf = sc_create_vcd_trace_file("ports_arrays");
41   ((vcd_trace_file*)wf)->sc_set_vcd_time_unit(-5);
42     for (z=0; z<4; z++) {
43       char str[3];
44       sprintf(str, "(%0d)",z);
45       sc_trace(wf,a[z],"a" + string(str));
46       sc_trace(wf,b[z],"b" + string(str));
47       sc_trace(wf,o[z],"o" + string(str));
48     }
49   // Start the testing here
50   sc_start(1);
51   for (z=0; z<4; z++) {
52     a[z] = rand();
53     b[z] = rand();
54     sc_start(1);
55     cout << "@" << sc_time_stamp() <<" a : " << a[z]
56       << " ,b : " << b[z] << " ,o : " << o[z] << endl;
57   }
58   sc_start(2);
59
60   sc_close_vcd_trace_file(wf);
61   return 0;// Terminate simulation
62 }
```

SystemC

WARNING: Default time step is used for VCD tracing.

@2 ns a : 3 ,b : 2 ,o : 5

@3 ns a : 1 ,b : 3 ,o : 4

@4 ns a : 1 ,b : 3 ,o : 4

@5 ns a : 2 ,b : 0 ,o : 2

Signal Binding

- Each port must be bound to a single signal
- When ports are bound to other signals or ports, both types must match
- When accessing a port, the variable assigned must have the same type as the port type

Example of Signal Binding

```
1 #include <systemc.h>
 2
 3 SC_MODULE (some_block) {
 4   sc_in<bool>   clock;
 5   sc_in<sc_bit> data;
 6   sc_in<sc_bit> reset;
 7   sc_in<sc_bit> inv;
 8   sc_out<sc_bit> out;
 9
10   void body () {
11     if (reset.read() == 1) {
12       out = sc_bit(0);
13     } else if (inv.read() == 1) {
14       out = ~out.read();
15     } else {
16       out.write(data.read());
17     }
18   }
19
20   SC_CTOR(some_block) {
21     SC_METHOD(body);
22       sensitive << clock.pos();
23   }
24 };
25
26
27 SC_MODULE (signal_bind) {
28   sc_in<bool> clock;
29
30   sc_signal<sc_bit> data;
31   sc_signal<sc_bit> reset;
32   sc_signal<sc_bit> inv;
33   sc_signal<sc_bit> out;
34   some_block *block;
35   int  done;
36
37   void do_test() {
38     while (true) {
39       wait();
40       if (done == 0) {
41         cout << "@" << sc_time_stamp() <<" Starting test"<<endl;
42         wait();
43         wait();
44         inv = sc_bit('0');
45         data = sc_bit('0');
46         cout << "@" << sc_time_stamp() <<" Driving 0 all inputs"<<endl;
47         // Assert reset
48         reset = sc_bit('1');
49         cout << "@" << sc_time_stamp() <<" Asserting reset"<<endl;
50         // Deassert reset
51         wait();
52         wait();
53         reset = sc_bit('0');
54         cout << "@" << sc_time_stamp() <<" De-Asserting reset"<<endl;
55         // Assert data
56         wait();
57         wait();
58         cout << "@" << sc_time_stamp() <<" Asserting data"<<endl;
59         data = sc_bit('1');
60         wait();
61         wait();
```

Example of Signal Binding

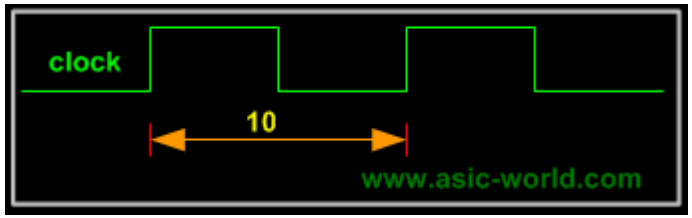
```
62        cout << "@" << sc_time_stamp() <<" Asserting inv"<<endl;
63        inv = sc_bit('1');
64        wait();
65        wait();
66        cout << "@" << sc_time_stamp() <<" De-Asserting inv"<<endl;
67        inv = sc_bit('0');
68        wait();
69        wait();
70        done = 1;
71      }
72    }
73  }
74
75  void monitor() {
76    cout << "@" << sc_time_stamp() << " data :" << data
77      << " reset :" << reset << " inv :"
78      << inv << " out :" << out <<endl;
79  }
80
81  SC_CTOR(signal_bind) {
82    block = new some_block("SOME_BLOCK");
83    block->clock       (clock)  ;
84    block->data        (data)   ;
85    block->reset       (reset)  ;
86    block->inv         (inv)    ;
87    block->out         (out)    ;
88    done = 0;
89
90    SC_CTHREAD(do_test,clock.pos());
91    SC_METHOD(monitor);
92      sensitive << data << reset << inv << out;
93  }
94 };
95

96 int sc_main (int argc, char* argv[]) {
97   sc_signal<bool> clock;
98   int i;
99
```

```
100   signal_bind  object("SIGNAL_BIND");
101     object.clock (clock);
102   sc_trace_file *wf = sc_create_vcd_trace_file("signal_bind");
103    sc_trace(wf, object.clock, "clock");
104    sc_trace(wf, object.reset, "reset");
105    sc_trace(wf, object.data, "data");
106    sc_trace(wf, object.inv, "inv");
107    sc_trace(wf, object.out, "out");
108
109   sc_start(0);
110   for(i=0;i<100;i++) {
111     if (object.done == 0) {
112       clock = 0;
113       sc_start(1);
114       clock = 1;
115       sc_start(1);
116     }
117   }
118   sc_close_vcd_trace_file(wf);
119
120   cout<<"Terminating Simulation"<<endl;
121   return 0;// Terminate simulation
122 }
```

```
SystemC
@0 s data :0 reset :0 inv :0 out :0
WARNING: Default time step is used for VCD tracing.
@3 ns Starting test
@7 ns Driving 0 all inputs
@7 ns Asserting reset
@7 ns data :0 reset :1 inv :0 out :0
@11 ns De-Asserting reset
@11 ns data :0 reset :0 inv :0 out :0
@15 ns Asserting data
@15 ns data :1 reset :0 inv :0 out :0
@17 ns data :1 reset :0 inv :0 out :1
@19 ns Asserting inv
@19 ns data :1 reset :0 inv :1 out :1
@21 ns data :1 reset :0 inv :1 out :0
@23 ns De-Asserting inv
@23 ns data :1 reset :0 inv :0 out :1
Terminating Simulation
```

Clocks



- Clocks used for synchronizing SystemC models (as HDLs).
- A clock object has a number of data members to store clock settings and methods to perform clock actions.
- To create a clock object use the syntax: sc_clock clock("my_clock", 10, 0.5, 1, false);
- This declaration will create a clock object named clock with a period of 10 time units, a duty cycle of 50%, the first edge will occur at 1 time units, and the first value will be false. All of these arguments have default values except for the clock name. The period defaults to 1, the duty cycle to 0.5, the first edge to 0, and the first value to true.
- Typically clocks are created at the top level of the design in the testbench and passed down through the module hierarchy to the rest of the design.
- When binding the clock to a port the designer must use the clock signal generated by the clock object to map to a port. This dis one by using the signal method of the clock object.
- For SC_CTHREAD processes the clock object is directly mapped to the clock input of the process and the signal() method is not required.

Example of Clocks

```
1 #include <systemc.h>
 2
 3 SC_MODULE (some_block) {
 4   sc_in<bool>   clock;
 5   sc_in<sc_bit> data;
 6   sc_in<sc_bit> reset;
 7   sc_in<sc_bit> inv;
 8   sc_out<sc_bit> out;
 9
10   void body () {
11     if (reset.read() == 1) {
12       out = sc_bit(0);
13     } else if (inv.read() == 1) {
14       out = ~out.read();
15     } else {
16       out.write(data.read());
17     }
18   }
19
20   SC_CTOR(some_block) {
21     SC_METHOD(body);
22       sensitive << clock.pos();
23   }
24 };
25
26
27 SC_MODULE (signal_bind) {
28   sc_in<bool> clock;
29
30   sc_signal<sc_bit> data;
31   sc_signal<sc_bit> reset;
32   sc_signal<sc_bit> inv;
33   sc_signal<sc_bit> out;
34   some_block *block;
35   int  done;
36
```

```
37   void do_test() {
38     while (true) {
39       wait();
40       if (done == 0) {
41         cout << "@" << sc_time_stamp() <<" Starting test"<<endl;
42         wait();
43         wait();
44         inv = sc_bit('0');
45         data = sc_bit('0');
46         cout << "@" << sc_time_stamp() <<" Driving 0 all inputs"<<endl;
47         // Assert reset
48         reset = sc_bit('1');
49         cout << "@" << sc_time_stamp() <<" Asserting reset"<<endl;
50         // Deassert reset
51         wait();
52         wait();
53         reset = sc_bit('0');
54         cout << "@" << sc_time_stamp() <<" De-Asserting reset"<<endl;
55         // Assert data
56         wait();
57         wait();
58         cout << "@" << sc_time_stamp() <<" Asserting data"<<endl;
59         data = sc_bit('1');
60         wait();
61         wait();
62         cout << "@" << sc_time_stamp() <<" Asserting inv"<<endl;
63         inv = sc_bit('1');
64         wait();
65         wait();
66         cout << "@" << sc_time_stamp() <<" De-Asserting inv"<<endl;
67         inv = sc_bit('0');
68         wait();
69         wait();
70         done = 1;
71       }
72     }
73   }
74
```

Example of Clocks

```
75   void monitor() {
76     cout << "@" << sc_time_stamp() << " data :" << data
77       << " reset :" << reset << " inv :"
78       << inv << " out :" << out <<endl;
79   }
80
81   SC_CTOR(signal_bind) {
82     block = new some_block("SOME_BLOCK");
83     block->clock       (clock)  ;
84     block->data        (data)   ;
85     block->reset       (reset)  ;
86     block->inv         (inv)    ;
87     block->out         (out)    ;
88     done = 0;
89
90     SC_CTHREAD(do_test,clock.pos());
91     SC_METHOD(monitor);
92       sensitive << data << reset << inv << out;
93   }
94
95 };
96
97
98 int sc_main (int argc, char* argv[]) {
99   sc_clock clock ("my_clock",1,0.5);
100
101   signal_bind  object("SIGNAL_BIND");
102     object.clock (clock.signal());
103   sc_trace_file *wf = sc_create_vcd_trace_file("clock");
104     sc_trace(wf, object.clock, "clock");
105     sc_trace(wf, object.reset, "reset");
106     sc_trace(wf, object.data, "data");
107     sc_trace(wf, object.inv, "inv");
108     sc_trace(wf, object.out, "out");
109
110   sc_start(100);
111   sc_close_vcd_trace_file(wf);
112
113   cout<<"Terminating Simulation"<<endl;
114   return 0;// Terminate simulation
115 }
```

```
SystemC
@0 s data :0 reset :0 inv :0 out :0
WARNING: Default time step is used for VCD tracing.
@1 ns Starting test
@3 ns Driving 0 all inputs
@3 ns Asserting reset
@3 ns data :0 reset :1 inv :0 out :0
@5 ns De-Asserting reset
@5 ns data :0 reset :0 inv :0 out :0
@7 ns Asserting data
@7 ns data :1 reset :0 inv :0 out :0
@8 ns data :1 reset :0 inv :0 out :1
@9 ns Asserting inv
@9 ns data :1 reset :0 inv :1 out :1
@10 ns data :1 reset :0 inv :1 out :0
@11 ns De-Asserting inv
@11 ns data :1 reset :0 inv :0 out :1
Terminating Simulation
```

Time Functions

- Notion of time and concurrency.
- Time data type
  - sc_time
  - sc_start()
  - sc_time_stamp()
  - wait(sc_time)
  - sc_simulation_time()
  - sc_event

sc_time

- The sc_time data type represents
  - simulation time
  - time intervals, incl. delays & timeouts.

- sc_time constructed from a **double** and an **sc_time_unit**
- Now time represented internally as an **unsigned integer of at least 64 bits**, but some member functions still use double!
- Enum types denote various time units
  - SC_FS = 0
  - SC_PS = 1
  - SC_NS = 2
  - SC_US = 3
  - SC_MS = 4
  - SC_SEC= 5
- Like any other SystemC data type, sc_time allows arithmetic operations

Time Resolution

- Time represented as an integer multiple of time resolution.
  - default time resolution is 1 ns
  - time resolution can only be changed by calling sc_set_time_resolution
- This function
  - only called during elaboration,
  - not called more than once, and
  - not called after constructing an object of type sc_time with a non-zero time value
- sc_get_time_resolution returns the time resolution

sc_zero_time

- SC_ZERO_TIME represents 0 time delay.
- It is good practice to use this constant whenever writing a time value of zero, for example, when creating a delta notification or a delta time-out.

Example of sc_time

```
1 #include <systemc.h>
2
3 int sc_main (int argc, char* argv[]) {
4   // sc_set_time_resol... should be called before
      sc_time
5   // variable declration
6   sc_set_time_resolution(1,SC_PS);
7   // Declare the sc_time variables
8   sc_time             t1(10,SC_NS);
9   sc_time             t2(5,SC_PS);
10   sc_time             t3,t4(1,SC_PS),t5(1,SC_PS);
11   // Print all the variables
12   cout <<"Value of t1 "<< t1.to_string() << endl;
13   cout <<"Value of t2 "<< t2.to_string() << endl;
14   cout <<"Value of t3 "<< t3.to_string() << endl;
15   cout <<"Value of t4 "<< t4.to_string() << endl;
16   cout <<"Value of t5 "<< t5.to_string() << endl;
17   // Start the sim
18   sc_start(0);
19   sc_start(1);
20   // Get the current time
21   t3 = sc_time_stamp();
22   cout <<"Value of t3 "<< t3.to_string() << endl;
23   // Run for some more time
24   sc_start(20);
25   // Get the current time
26   t4 = sc_time_stamp();
27   // Print the variables for new time
28   cout <<"Value of t4 "<< t4.to_string() << endl;
29   // This is how you do arth operation
30   t5 = t4 - t3;
31   cout <<"Value of t5 "<< t5.to_string() << endl;
32   // This is how we do compare operation
33   if (t5 > t2) {
34     cout <<" t5 is greated then t2" << endl;
35   } else {
36     cout <<" t2 is greated then t5" << endl;
37   }
38   return 1;
39 }
```

Output

Value of t1 10 ns
Value of t2 5 ps
Value of t3 0 s
Value of t4 1 ps
Value of t5 1 ps
Value of t3 1 ns
Value of t4 21 ns
Value of t5 20 ns
t5 is greated then t2

sc_start

- sc_start() starts the simulation phase, which consists of initialization and execution.

The sc_start() methods performs the following operations

- **Called First Time :** sc_start starts the scheduler, which will run up to the simulation time passed as an argument (if an argument was passed).
- **When called on the second and subsequent occasions :** sc_start will resume the scheduler from the time it had reached at the end of the previous call to sc_start. The scheduler will run for the time passed as an argument (if an argument was passed), relative to the current simulation time..
- **When a time is passed as an argument :** the scheduler will execute up to and including the timed notification phase that advances simulation time to the end time (calculated by adding the time given as an argument to the simulation time when function sc_start is called).
- **Called without any arguments :** the scheduler will run until it completes.
- **Called with a zero-valued time argument :** the scheduler shall run for one delta cycle.
- Once started, the scheduler runs until it completes, the application calls **sc_stop**, or exception occurs.
- Once the function sc_stop has been called, function sc_start shall not be called again. Function sc_start may be called from function **sc_main**, and only from function sc_main.

sc_time

```
1 #include <systemc.h>
 2
 3 SC_MODULE (some_block) {
 4   sc_in<bool>   clock;
 5   sc_in<sc_bit> data;
 6   sc_in<sc_bit> reset;
 7   sc_in<sc_bit> inv;
 8   sc_out<sc_bit> out;
 9
10   void body () {
11     if (reset.read() == 1) {
12       out = sc_bit(0);
13     } else if (inv.read() == 1) {
14       out = ~out.read();
15     } else {
16       out.write(data.read());
17     }
18   }
19
20   SC_CTOR(some_block) {
21     SC_METHOD(body);
22       sensitive << clock.pos();
23   }
24 };
25
```

```
26
27 SC_MODULE (signal_bind) {
28   sc_in<bool> clock;
29
30   sc_signal<sc_bit> data;
31   sc_signal<sc_bit> reset;
32   sc_signal<sc_bit> inv;
33   sc_signal<sc_bit> out;
34   some_block *block;
35
36   void do_test() {
37     while (true) {
38       wait();
39       cout << "@" << sc_time_stamp() <<" Starting test"<<endl;
40       wait();
41       wait();
42       inv = sc_bit('0');
43       data = sc_bit('0');
44       cout << "@" << sc_time_stamp() <<" Driving 0 all inputs"<<endl;
45       // Assert reset
46       reset = sc_bit('1');
47       cout << "@" << sc_time_stamp() <<" Asserting reset"<<endl;
48       // Deassert reset
49       wait();
50       wait();
51       reset = sc_bit('0');
52       cout << "@" << sc_time_stamp() <<" De-Asserting reset"<<endl;
```

sc_time

```
    // Assert data
     wait();
     wait();
     cout << "@" << sc_time_stamp() <<" Asserting data"<<endl;
     data = sc_bit('1');
     wait();
     wait();
     cout << "@" << sc_time_stamp() <<" Asserting inv"<<endl;
     inv = sc_bit('1');
     wait();
     wait();
     cout << "@" << sc_time_stamp() <<" De-Asserting inv"<<endl;
     inv = sc_bit('0');
     wait();
     wait();
     cout<<"Terminating Simulation"<<endl;
     sc_stop(); // sc_stop triggers end of simulation
    }
  }

  void monitor() {
    cout << "@" << sc_time_stamp() << " data :" << data
      << " reset :" << reset << " inv :"
      << inv << " out :" << out <<endl;
  }

  SC_CTOR(signal_bind) {
     block = new some_block("SOME_BLOCK");
     block->clock       (clock)  ;
     block->data        (data)   ;
     block->reset       (reset)  ;
     block->inv         (inv)    ;
     block->out         (out)    ;

     SC_CTHREAD(do_test,clock.pos());
     SC_METHOD(monitor);
       sensitive << data << reset << inv << out;
    }
  };

  int sc_main (int argc, char* argv[]) {
    sc_clock clock ("my_clock",1,0.5);

    signal_bind  object("SIGNAL_BIND");
     object.clock (clock.signal());
    sc_trace_file *wf = sc_create_vcd_trace_file("sc_start");
     sc_trace(wf, object.clock, "clock");
     sc_trace(wf, object.reset, "reset");
     sc_trace(wf, object.data, "data");
     sc_trace(wf, object.inv, "inv");
     sc_trace(wf, object.out, "out");
    sc_start(0); // First time called will init schedular
    sc_start(1); // Increment simulation by 1 time unit
    sc_start();  // Run the simulation till sc_stop is encountered
    // sc_start() is terminated and return to next startment after
    // encountering sc_stop()
    cout<<"Closing VCD File"<<endl;
    sc_close_vcd_trace_file(wf);
    return 0;// Terminate simulation
  }
```

Output

```
SystemC
 @0 s data :0 reset :0 inv :0 out :0
 WARNING: Default time step is used for VCD tracing.
 @1 ns Starting test
 @3 ns Driving 0 all inputs
 @3 ns Asserting reset
 @3 ns data :0 reset :1 inv :0 out :0
 @5 ns De-Asserting reset
 @5 ns data :0 reset :0 inv :0 out :0
 @7 ns Asserting data
 @7 ns data :1 reset :0 inv :0 out :0
 @8 ns data :1 reset :0 inv :0 out :1
 @9 ns Asserting inv
 @9 ns data :1 reset :0 inv :1 out :1
 @10 ns data :1 reset :0 inv :1 out :0
 @11 ns De-Asserting inv
 @11 ns data :1 reset :0 inv :0 out :1
 Terminating Simulation
 SystemC: simulation stopped by user.
 Closing VCD File
```

sc_time_stamp

- The function sc_time_stamp returns the current simulation time
  - During elaboration and initialization the function will return a value of zero

```
 1 #include <systemc.h>
 2
 3 int sc_main (int argc, char* argv[]) {
 4   cout<<"Current time is "<< sc_time_stamp() << endl;
 5   sc_start(1);
 6   cout<<"Current time is "<< sc_time_stamp() << endl;
 7   sc_start(100);
 8   cout<<"Current time is "<< sc_time_stamp() << endl;
 9   sc_stop();
10   cout<<"Current time is "<< sc_time_stamp() << endl;
11   return 0;// Terminate simulation
12 }
```

```
SystemC …
Current time is 0 s
Current time is 1 ns
Current time is 101 ns
SystemC: simulation stopped by user.
Current time is 101 ns
```

sc_event

- sc_event used for process synchronization
  - same as event in Verilog.
- A process instance may be triggered or resumed on the occurrence of an event, that is when the event is notified
- Thus, event_name.notify() creates an immediate notification
  - any and all process instances sensitive to the event shall be made runnable before control is returned from function notify.
  - same as -> event_name in Verilog.
- notify() can be called with parameters
  - **event_name.notify()**: immediate notification
  - **event_name.notify(SC_ZERO_TIME):** Delta notification
  - **event_name.notify(1, SC_NS):**Timed notification
  - **event_name.cancel()** deletes any pending notification for this event.

Example of sc_event

```
1 #include <systemc.h>
2
3 SC_MODULE (events) {
4   sc_in<bool> clock;
5
6   sc_event  e1;
7   sc_event  e2;
8
9   void do_test1() {
10    while (true) {
11      // Wait for posedge of clock
12      wait();
13      cout << "@" << sc_time_stamp() <<" Starting test"<<endl;
14      // Wait for posedge of clock
15      wait();
16      cout << "@" << sc_time_stamp() <<" Triggering e1"<<endl;
17      // Trigger event e1
18      e1.notify(5,SC_NS);
19      // Wait for posedge of clock
20      wait();
21      // Wait for event e2
22      wait(e2);
23      cout << "@" << sc_time_stamp() <<" Got Trigger e2"<<endl;
24      // Wait for posedge of clock
25      wait();
26      cout<<"Terminating Simulation"<<endl;
27      sc_stop(); // sc_stop triggers end of simulation
28    }
29  }
30
31  void do_test2() {
32    while (true) {
33      // Wait for event e2
34      wait(e1);
35      cout << "@" << sc_time_stamp() <<" Got Trigger e1"<<endl;
36      // Wait for 3 posedge of clock
37      wait(3);
38      cout << "@" << sc_time_stamp() <<" Triggering e2"<<endl;
39      // Trigger event e2
40      e2.notify();
41    }
42  }
43
44  SC_CTOR(events) {
45    SC_CTHREAD(do_test1,clock.pos());
46    SC_CTHREAD(do_test2,clock.pos());
47  }
48 };
49
50 int sc_main (int argc, char* argv[]) {
51   sc_clock clock ("my_clock",1,0.5);
52
53   events  object("events");
54     object.clock (clock.signal());
55
56   sc_start(0); // First time called will init schedular
57   sc_start();  // Run the simulation till sc_stop is encountered
58   return 0;// Terminate simulation
59 }
```

```
SystemC
@1 ns Starting test
@2 ns Triggering e1
@7 ns Got Trigger e1
@10 ns Triggering e2
@11 ns Got Trigger e2
Terminating Simulation
SystemC: simulation stopped by user.
```

wait()

- wait() suspends the thread or clocked thread from which it is called until that process instance is resumed
- The dynamic sensitivity is determined by the arguments passed to function wait (see example of clock)
- A call to the function wait with an empty argument list or with a single integer shall use the static sensitivity of the process instance.
  - This is the only form of wait permitted within a clocked thread process
  - A call to the function wait with one or more non-integer arguments shall override the static sensitivity of the process instance
  - When calling function wait with a passed-by-reference parameter, the application shall be obliged to ensure that the lifetimes of any actual arguments passed-by-reference extend from the time the function is called to the time the function call completes, and moreover in the case of a parameter of type sc_time, the application shall not modify the value of the actual argument during that period

wait()

The function wait can take a parameter:

- wait(): waits for the sensitive list event to occur
- wait(int): waits for n events to occur, events are the ones in sensitivity list
- wait(event): waits for event mentioned as parameter to occur
- wait(double,sc_time_unit): waits for specified time
- wait(double,sc_time_unit, event): waits for specified time or event to occur

Example of wait

```
1 #include <systemc.h>
2
3 SC_MODULE (wait_example) {
4   sc_in<bool> clock;
5
6   sc_event  e1;
7   sc_event  e2;
8
9   void do_test1() {
10    while (true) {
11      // Wait for posedge of clock
12      wait();
13      cout << "@" << sc_time_stamp() <<" Starting test"<<endl;
14      // Wait for 5 posedge of clock
15      wait(5);
16      cout << "@" << sc_time_stamp() <<" Triggering e1"<<endl;
17      // Trigger event e1
18      e1.notify(5,SC_NS);
19      // Wait for 3 NS
20      cout << "@" << sc_time_stamp() <<" Time before wait 3 ns"<<endl;
21      wait(3, SC_NS);
22      cout << "@" << sc_time_stamp() <<" Time after wait 3 ns"<<endl;
23      // Wait for event e2
24      wait(e2);
25      cout << "@" << sc_time_stamp() <<" Got Trigger e2"<<endl;
26      // Wait for posedge of clock
27      wait(30);
28      cout<<"Terminating Simulation"<<endl;
29      sc_stop(); // sc_stop triggers end of simulation
30    }
31  }
32
33  void do_test2() {
34    while (true) {
35      // Wait for event e2
36      wait(e1);
37      cout << "@" << sc_time_stamp() <<" Got Trigger e1"<<endl;
38      // Wait for 3 posedge of clock
39      wait(20);
40      cout << "@" << sc_time_stamp() <<" Triggering e2"<<endl;
41      // Trigger event e2
42      e2.notify();
43      // Wait for either 20ns or event e1
44      wait(20,SC_NS,e1);
45      cout << "@" << sc_time_stamp() <<" Done waiting for 20ns or event e1"<<endl;
46    }
47  }
48
49  SC_CTOR(wait_example) {
50    SC_CTHREAD(do_test1,clock.pos());
51    SC_CTHREAD(do_test2,clock.pos());
52  }
53 };
54
55 int sc_main (int argc, char* argv[]) {
56   sc_clock clock ("my_clock",1,0.5);
57
58   wait_example  object("wait");
59     object.clock (clock.signal());
60
61   sc_start(0); // First time called will init schedular
62   sc_start();  // Run the simulation till sc_stop is encountered
63   return 0;// Terminate simulation
64 }
```

```
SystemC
@1 ns Starting test
@6 ns Triggering e1
@6 ns Time before wait 3 ns
@9 ns Time after wait 3 ns
@11 ns Got Trigger e1
@31 ns Triggering e2
@32 ns Got Trigger e2
@51 ns Done waiting for 20ns or event e1
Terminating Simulation
SystemC: simulation stopped by user.
```

next_trigger

- next_trigger() is used with methods, not threads.
- next_trigger does not suspend the method, but always executes to completion before returning control to the kernel.
  - This is different from wait() method used with threads.

- next_trigger(): In the absence of static sensitivity, the process shall not be triggered again during the current simulation
- next_trigger(event): The process shall be triggered when the event passed as an argument is notified
- next_trigger(double,sc_time_unit): The process shall be triggered when specified time has elapsed
- next_trigger(double,sc_time_unit, event): The process shall be triggered when the given event is notified or after given time, whichever occurs first

Example of next_trigger

```
1 #include <systemc.h>
 2
 3 SC_MODULE (next_trigger_example) {
 4   sc_in<bool> clock;
 5
 6   sc_event  e1,e2;
 7   int cnt;
 8
 9   void do_test1() {
10     switch (cnt) {
11       case 0 : cout << "@" << sc_time_stamp() <<" Default trigger clk triggered"<<endl;
12                next_trigger(e1);
13                break;
14       case 1 : cout << "@" << sc_time_stamp() <<" Event e1 triggered"<<endl;
15                next_trigger(10, SC_NS);
16                break;
17       case 2 : cout << "@" << sc_time_stamp() <<" Event e1 occured or time 10ns passed"<<endl;
18                next_trigger(e1 | e2);
19                break;
20       case 3 : cout << "@" << sc_time_stamp() <<" Event e1 or e2 triggered"<<endl;
21                break;
22       default :cout << "@" << sc_time_stamp() <<" Default trigger clk triggered"<<endl;
23                break;
24     }
25     cnt ++;
26   }
27
28   void do_test2() {
29     while (true) {
30       wait(2);
31       // Trigger event e1
32       cout << "@" << sc_time_stamp() <<" Triggering e1"<<endl;
33       e1.notify();
34       wait(20);
35       cout << "@" << sc_time_stamp() <<" Triggering e2"<<endl;
36       e2.notify();
37       // Wait for  2 posedge of clocks
38       wait(2);
39       cout << "@" << sc_time_stamp() <<" Terminating simulation"<<endl;
40       sc_stop(); // sc_stop triggers end of simulation
41     }
42   }
43
44   SC_CTOR(next_trigger_example) {
45     cnt = 0;
46     SC_METHOD(do_test1);
47       sensitive << clock;
48     SC_CTHREAD(do_test2,clock.pos());
49   }
50 };
51
52 int sc_main (int argc, char* argv[]) {
53   sc_clock clock ("my_clock",1,0.5);
54
55   next_trigger_example  object("wait");
56     object.clock (clock.signal());
57
58   sc_start(0); // First time called will init schedular
59   sc_start();  // Run the simulation till sc_stop is encountered
60   return 0;// Terminate simulation
61 }
```

```
SystemC
@0 s Default trigger clk triggered
@2 ns Triggering e1
@2 ns Event e1 triggered
@12 ns Event e1 occured or time 10ns passed
@22 ns Triggering e2
@22 ns Event e1 or e2 triggered
@22500 ps Default trigger clk triggered
@23 ns Default trigger clk triggered
@23500 ps Default trigger clk triggered
@24 ns Default trigger clk triggered
@24 ns Terminating simulation
SystemC: simulation stopped by user.
```

Primitive Channels

- SystemC provides channels for communication between two modules.
- Channels make complex communication easy.
- SystemC has two types of channels.
  - Primitive
  - Hierarchical

- The sc_prim_channel is the base class for all primitive channels
  - provides unique access to the update phase of the scheduler

- Predefined primitive channels to model communication/sync mechanisms
  - sc_mutex
  - sc_semaphore
  - sc_fifo

sc_mutex

- The sc_mutex is a predefined primitive channel modeling the behavior of a mutual exclusion lock controlling access to a resource shared by concurrent processes
  - A mutex is in one of two exclusive states: unlocked or locked
  - only one process can lock a given mutex at one time
  - A mutex can only be unlocked by the particular process that locked the mutex, but may be locked subsequently by a different process

Mutex constructor initializes mutex to unlocked

sc_mutex comes with the following pre-defined methods

- int lock(); Lock the mutex if it is free, else wait till mutex gets free.
- int unlock(); Unlock the mutex
- int trylock(); Check if mutex is free, if free then lock it else return -1.
- char* kind(); Return string "sc_mutex"

Example of sc_mutex

```
#include <systemc.h>

SC_MODULE (sc_mutex_example) {
 sc_in<bool> clock;

 sc_mutex   bus;
 int     cnt;


 void do_bus(int who) {
   cout << "@" << sc_time_stamp() <<" Bus access by instance " << who << endl;
 }

 void do_test1() {
   while (true) {
     wait();
     cout << "@" << sc_time_stamp() <<" Checking mutex intance 0"<<endl;
     // Check if mutex is available
     if (bus.trylock()  ! = -1) {
       cout << "@" << sc_time_stamp() <<" Got mutex for intance 0"<<endl;
       cnt ++;
       do_bus(0);
       wait(2);
       // Unlock the mutex
       bus.unlock();
     }
     if (cnt >= 3) {
       sc_stop(); // sc_stop triggers end of simulation
     }
   }
 }

 void do_test2() {
   while (true) {
     wait();
     cout << "@" << sc_time_stamp() <<" Checking mutex intance 1"<<endl;
     // Wait till mutex is available
     bus.lock();
     cout << "@" << sc_time_stamp() <<" Got mutex for intance 1"<<endl;
     do_bus(1);
     wait(3);
     // Unlock the mutex
     bus.unlock();
   }
 }


 SC_CTOR(sc_mutex_example) {
   cnt = 0;
   SC_CTHREAD(do_test1,clock.pos());
   SC_CTHREAD(do_test2,clock.pos());
 }
};

int sc_main (int argc, char* argv[]) {
 sc_clock clock ("my_clock",1,0.5);

 sc_mutex_example  object("wait");
   object.clock (clock.signal());

 sc_start(0); // First time called will init schedular
 sc_start();  // Run the simulation till sc_stop is encountered
 return 0;// Terminate simulation
}
```

```
SystemC
@1 ns Checking mutex intance 1
@1 ns Got mutex for intance 1
@1 ns Bus access by instance 1
@1 ns Checking mutex intance 0
@2 ns Checking mutex intance 0
@3 ns Checking mutex intance 0
@4 ns Checking mutex intance 0
@4 ns Got mutex for intance 0
@4 ns Bus access by instance 0
@5 ns Checking mutex intance 1
@7 ns Got mutex for intance 1
@7 ns Bus access by instance 1
@7 ns Checking mutex intance 0
@8 ns Checking mutex intance 0
@9 ns Checking mutex intance 0
@10 ns Checking mutex intance 0
@10 ns Got mutex for intance 0
@10 ns Bus access by instance 0
@11 ns Checking mutex intance 1
@13 ns Got mutex for intance 1
@13 ns Bus access by instance 1
@13 ns Checking mutex intance 0
@14 ns Checking mutex intance 0
@15 ns Checking mutex intance 0
@16 ns Checking mutex intance 0
@16 ns Got mutex for intance 0
@16 ns Bus access by instance 0
@17 ns Checking mutex intance 1
SystemC: simulation stopped by user.
```

sc_semaphore

The sc_semaphore is a predefined primitive channel modeling the behavior of a software semaphore

- provides limited concurrent access to a shared resource

Semaphore constructor sets a semaphore value (int) to the permitted no concurrent accesses

sc_semaphore comes with

- int wait(); If the semaphore value is 0, member execution suspends until the semaphore value is incremented (by another process), at which point it shall resume and decrement the semaphore value.
- int trywait(); If the semaphore value is 0, trywait immediately returns the value -1 without modifying the semaphore value.
- int post(); Member function post increments the semaphore value. If processes exist that are suspended and are waiting for the semaphore value to be incremented, exactly one of these processes shall be permitted to decrement the semaphore value (the choice of process being non-deterministic) while the remaining processes shall suspend again.
- int get_value(); Returns the semaphore value.
- char* kind(); Returns string "sc_semaphore"

Example of sc_semaphore

```
#include <systemc.h>

SC_MODULE (sc_semaphore_example) {
 sc_in<bool> clock;

 sc_semaphore   bus;
 int     cnt;

 void bus_semaphore() {
   while (true) {
     wait();
     cout << "@" << sc_time_stamp() <<" Check if semaphore is 0 " << endl;
     if (bus.get_value() == 0) {
       cout << "@" << sc_time_stamp() <<" Posting 2 to semaphore " << endl;
       bus.post();
       bus.post();
       if (cnt >= 3) {
         sc_stop(); // sc_stop triggers end of simulation
       }
       cnt ++;
     }
   }
 }

 void do_read() {
   while (true) {
     wait();
     cout << "@" << sc_time_stamp() <<" Checking semaphore for intance 0"<<endl;
     // Check if semaphore is available
     if (bus.trywait()  ! = -1) {
       cout << "@" << sc_time_stamp() <<" Got semaphore for intance 0"<<endl;
       wait(2);
     }
   }
 }

 void do_write() {
   while (true) {
     wait();
     cout << "@" << sc_time_stamp() <<" Checking semaphore for intance 1"<<endl;
     // Wait till semaphore is available
     bus.wait();
     cout << "@" << sc_time_stamp() <<" Got semaphore for intance 1"<<endl;
     wait(3);
   }
 }

  SC_CTOR(sc_semaphore_example) : bus(0){
    cnt = 0;
    SC_CTHREAD(do_read,clock.pos());
    SC_CTHREAD(do_write,clock.pos());
    SC_CTHREAD(bus_semaphore,clock.pos());
  }
};

int sc_main (int argc, char* argv[]) {
 sc_clock clock ("my_clock",1,0.5);

 sc_semaphore_example  object("semaphore");
   object.clock (clock.signal());

 sc_start(0); // First time called will init schedular
 sc_start();  // Run the simulation till sc_stop is encountered
 return 0;// Terminate simulation
}
```

```
SystemC
@1 ns Check if semaphore is 0
@1 ns Posting 2 to semaphore
@1 ns Checking semaphore for intance 1
@1 ns Got semaphore for intance 1
@1 ns Checking semaphore for intance 0
@1 ns Got semaphore for intance 0
@2 ns Check if semaphore is 0
@2 ns Posting 2 to semaphore
@3 ns Check if semaphore is 0
@4 ns Check if semaphore is 0
@4 ns Checking semaphore for intance 0
@4 ns Got semaphore for intance 0
@5 ns Check if semaphore is 0
@5 ns Checking semaphore for intance 1
@5 ns Got semaphore for intance 1
@6 ns Check if semaphore is 0
@6 ns Posting 2 to semaphore
@7 ns Check if semaphore is 0
@7 ns Checking semaphore for intance 0
@7 ns Got semaphore for intance 0
@8 ns Check if semaphore is 0
@9 ns Check if semaphore is 0
@9 ns Checking semaphore for intance 1
@9 ns Got semaphore for intance 1
@10 ns Check if semaphore is 0
@10 ns Posting 2 to semaphore
@10 ns Checking semaphore for intance 0
@10 ns Got semaphore for intance 0
SystemC: simulation stopped by user.
```

sc_fifo

- sc_fifo is a predefined primitive channel intended to model a fifo.
  - A fifo is an object of class sc_fifo.
  - Each fifo has a number of slots for storing values. The number of slots is fixed when the object is constructed. Default slots size is 16.

The sc_fifo has following predefined methods.

- write(): This method write the values passed as an argument into the fifo. If fifo if full then write() function waits till fifo slot is available
- nb_write(): This method is same as write(), only difference is, when fifo is full nb_write() does not wait till fifo slot is avaible. Rather it returns false.
- read(): This method returns the least recent written data in fifo. If fifo is empty, then read() function waits till data is available in fifo.
- nb_read():This method is same as read(), only difference is, when fifo is empty, nb_read() does not wait till fifo has some data. Rather it returns false.
- num_available(): This method returns the numbers of data values available in fifo in current delta time.
- num_free(): This method returns the number of free slots available in fifo in current delta time.

Example of sc_fifo

```
#include <systemc.h>

SC_MODULE(example_fifo) {
  void example_fifo::m_drain_packets(void);
  void example_fifo::t_source1(void);
  void example_fifo::t_source2(void);
  // Constructor
  SC_CTOR(example_fifo) {
    SC_METHOD(m_drain_packets);
    SC_THREAD(t_source1);
    SC_THREAD(t_source2);
    // Size the packet_fifo to 5 ints.
    sc_fifo<int> packet_fifo (5);
  }

  // Declare the FIFO
  sc_fifo<int> packet_fifo;
};

void example_fifo::m_drain_packets(void) {
  int val;
  if (packet_fifo.nb_read(val)) {
    std::cout << sc_time_stamp() << ": m_drain_packets(): Received " << val <<
      std::endl;
  } else {
    std::cout << sc_time_stamp() << ": m_drain_packets(): FIFO empty." << std::endl;
  }
  // Check back in 2ns
  next_trigger(2, SC_NS);
}

void example_fifo::t_source1(void) {
  int val = 1000;
  for (;;) {
    wait(3, SC_NS);
    val++;
    packet_fifo.write(val);
    std::cout << sc_time_stamp() << ": t_thread1(): Wrote " << val << std::endl;
  }
}

void example_fifo::t_source2(void) {
  int val = 2000;
  for (;;) {
    wait(5, SC_NS);
    val++;
    packet_fifo.write(val);
    std::cout << sc_time_stamp() << ": t_thread2(): Wrote " << val << std::endl;
  }
}

int sc_main(int argc, char* argv[]) {
  example_fifo ex_fifo ("ex_fifo0");
  sc_start(30, SC_NS);
  return 0;
}
SystemC
0 s: m_drain_packets(): FIFO empty.
2 ns: m_drain_packets(): FIFO empty.
3 ns: t_thread1(): Wrote 1001
4 ns: m_drain_packets(): Received 1001
5 ns: t_thread2(): Wrote 2001
6 ns: m_drain_packets(): Received 2001
6 ns: t_thread1(): Wrote 1002
8 ns: m_drain_packets(): Received 1002
9 ns: t_thread1(): Wrote 1003
10 ns: m_drain_packets(): Received 1003
10 ns: t_thread2(): Wrote 2002
12 ns: m_drain_packets(): Received 2002
12 ns: t_thread1(): Wrote 1004
14 ns: m_drain_packets(): Received 1004
15 ns: t_thread1(): Wrote 1005
15 ns: t_thread2(): Wrote 2003
16 ns: m_drain_packets(): Received 1005
18 ns: m_drain_packets(): Received 2003
18 ns: t_thread1(): Wrote 1006
20 ns: m_drain_packets(): Received 1006
20 ns: t_thread2(): Wrote 2004
21 ns: t_thread1(): Wrote 1007
22 ns: m_drain_packets(): Received 2004
24 ns: m_drain_packets(): Received 1007
24 ns: t_thread1(): Wrote 1008
25 ns: t_thread2(): Wrote 2005
26 ns: m_drain_packets(): Received 1008
27 ns: t_thread1(): Wrote 1009
28 ns: m_drain_packets(): Received 2005
```