# SystemC Primer

- SystemC Primer Information Module
- SystemC Stand-Alone Lab
- SystemC Lab using Riviera

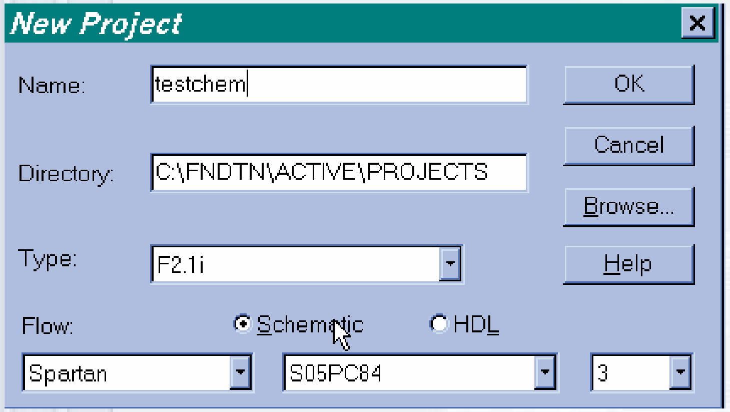
# SystemC

## Introduction

## Exploring Limitations

- One of the questions that Engineers must always ask when faced with a new design methodology is Why?
- Why should I learn this new method? The answer to this question can be quite complex, but one answer is **limitations**. *The techniques I know today may still do the job, but can they do them efficiently*.
- This is one of the main questions you must always ask yourself as you design, what are the limitations of my current design set, and can I learn a different way, and can that way allow me to do my job more  efficiently.

# Exploring Limitations


New Project

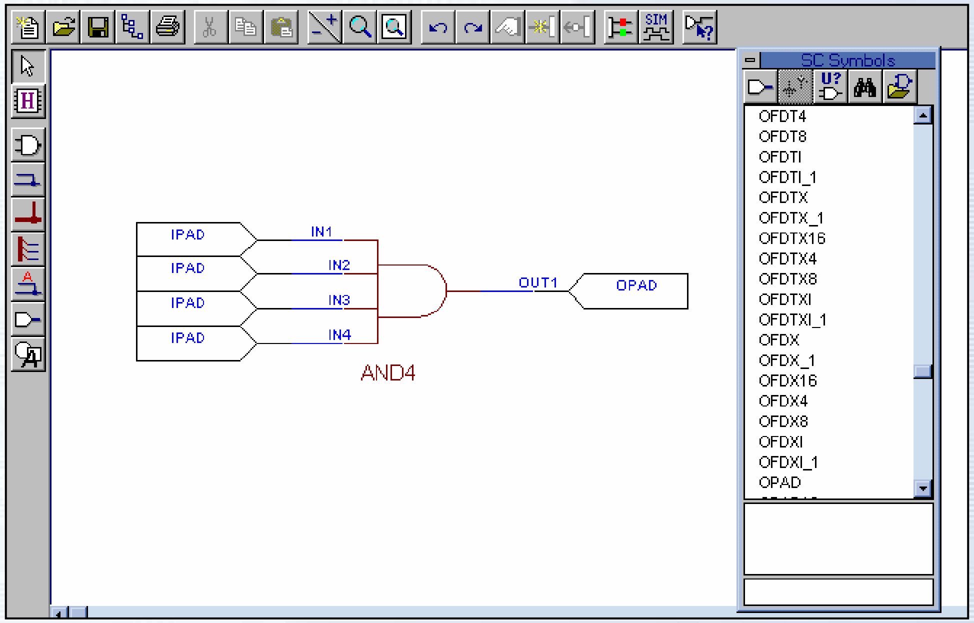
Name: testchem

Directory: C:\FNDTN\ACTIVE\PROJECTS

Type: F2.1i

Flow: [✓ Schematic] HDL

Spartan | S05PC84 | 3



When you are draw a schematic, you choose from a pre-defined set of library elements that have been defined for that specific device. If you wish to change devices, you often have to re-draw the diagram as the symbol library may not be the same. There is a limitation in simulating a schematic as well.

## Exploring Limitations

- When we write a VHDL design, we are not referring to any particular device library, we are describing the logical interactions of signals moving through a design. We are thinking at a higher level of abstraction.

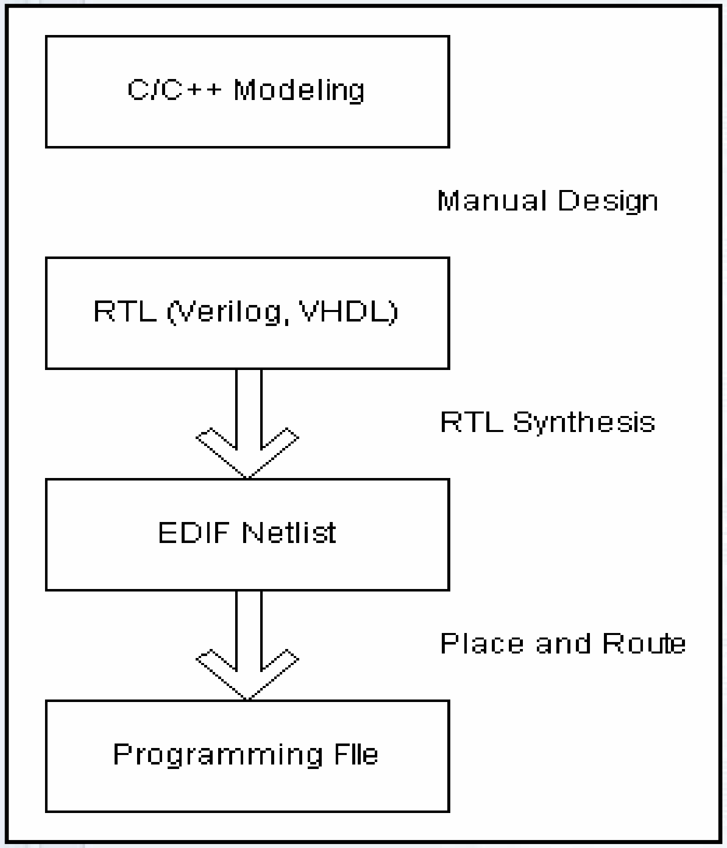
```
-- VHDL Source for a 4 input
-- AND GATE

library IEEE;
use IEEE.std_logic_1164.all;

entity logic1 is
 port(in1, in2: in std_logic;
      in3, in4: in std_logic;
      out1    : out std_logic);
end logic1;
```

The ability to describe behavior is a major step, and HDL languages have replaced schematic design for circuit descriptions.

*However, while HDL designs can be good for RTL coding they are limited in their ability to model behavior and test it at a higher level of abstraction*

```
architecture RTL of logic1  is
begin
  out1 <= in1 and in2 and in3
          and in4;
end RTL;
```

# Levels of Abstraction



C++ is a desirable modeling tool since it enables a sufficient level of abstraction to try out complex ideas.

Current design methodologies require that there be a manual step between the C++ modeling stage and the RTL Description. A desire is to have *bit accurate* models in C++, and then transition to RTL code.

## Why C++ ?

- Why C++?
  - Object Orientation is characterised by code re-use
  - Extensible features allow addition of HW constructs
  - Existing tool availability (IDE, debug, profiling)
  - Many designs are already started in C/C++
- Why not develop a new language?
  - New tool environment
  - New learning curve for both system and hardware

## C++ Design Mission

- Bring high-level design to the mainstream
  - Raise the level of abstraction available to the logic designer
  - Bridge the gap between the system designer and the hardware designer
  - Provide leading-edge synthesis technology

## Levels of Abstraction

Various parts of our design require different levels of abstraction when we describe them. Let's see how different languages deal with different levels of abstraction:

| Level of Abstraction | Verilog | VHDL | C/C++ |
|---|---|---|---|
| System Level | Not Suitable | Poor | Very Good |
| High Level (Behavioral) | Good | Very Good | Good |
| Medium Level (RTL) | Very Good | Very Good | Poor |
| Low Level (Gates) | Good | Poor | Not Suitable |

## Why Not Just Use C/C++?

- One of the fundamental issues in describing hardware constructs using a software language is the issue of *concurrency*. One of the main things that we learn when we use an HDL is that there is the concept of concurrency.

- This is an example of a C style rendition of a counter structure. The question is, how do we invoke this counter when the clock changes from 0 to 1 so that it acts like a *hardware* counter?

```
int counter (int clk)
{
    static int countval = 0;


    if (clk == 1) {
       if (countval == 255)
            countval = 0;
       else
            countval = countval + 1;
    }

    return countval;

}
```

## What is SystemC

- SystemC is a C++ class library and a methodology that you can use to effectively create a cycle-accurate model of software algorithms, hardware architecture, and interfaces of your SoC (System On a Chip) and system-level designs.

- You can use SystemC and standard C++ development tools to create a system-level model, quickly simulate to validate and optimize the design, explore various algorithms, and provide the hardware and software development team with an executable specification of the system.

- An executable specification is essentially a C++ program that exhibits the same behavior as the system when executed.

- With today's modern HDL frameworks, you can mix SystemC with an HDL design to make a more complex efficient model of a piece of hardware, or make a more robust testbench.

## SystemC Adds to C++

- The SystemC Class Library provides the necessary constructs to model system architectures, including hardware timing, concurrency, and reactive behaviors that are missing in standard C++.

- The C++ object-oriented programming language provides the ability to extend the language through classes, without adding new syntactic constructs. SystemC provides these necessary classes and allows designers to continue to use the familiar C++ language and development tools.

## SystemC Features

- SystemC supports hardware-software co-design and the description of the architecture of complex systems consisting of both hardware and software components. It supports the description of hardware, software, and interfaces in a C++ environment.
- The following features of SystemC allow it to be used as a co-design language:
- **Modules**: SystemC has a notion of a container class called a module. This is a hierarchical entity that can have other modules or processes contained in it.
- **Processes**: Processes are used to describe functionality. Processes are contained inside modules. SystemC provides three different process abstractions to be used by hardware and software designers.
- **Ports**: Modules have ports through which they connect to other modules. SystemC supports single-direction and bidirectional ports.

## SystemC Features

- **Signals**: SystemC supports resolved and unresolved signals. Resolved signals can have more than one driver (a bus) while unresolved signals can have only one driver.

- **Rich set of port and signal types**: To support modeling at different levels of abstraction, from the functional to the RTL, SystemC supports a rich set of port and signal types. This is different than languages like Verilog that only support bits and bit-vectors as port and signal types. SystemC supports both two-valued and four-valued signal types.

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

## SystemC Features

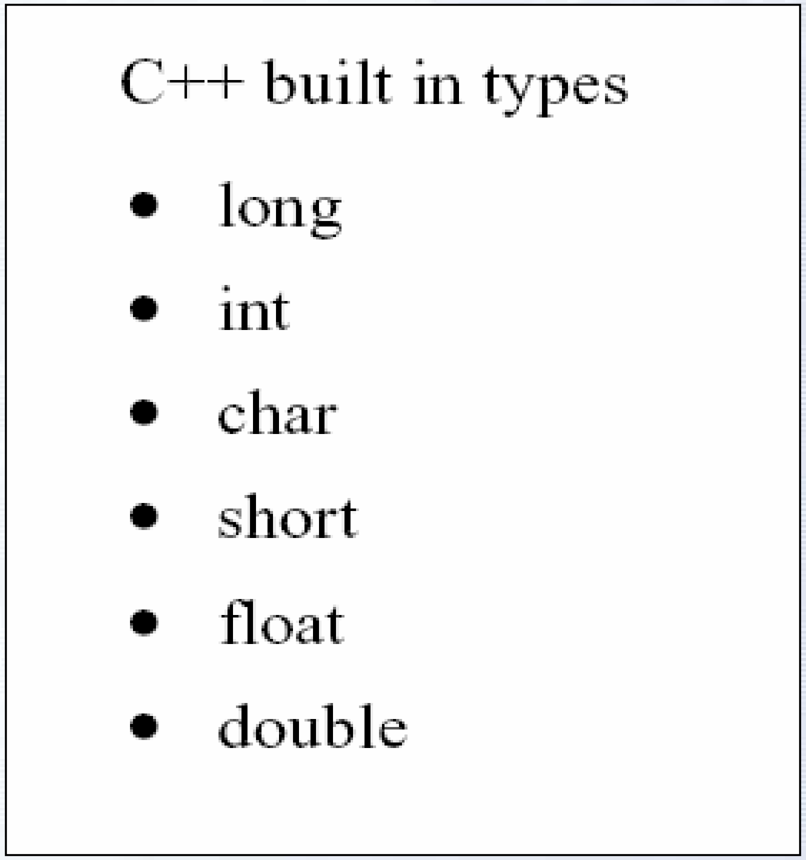
- **Multi-Level logic values**: SystemC supports both two-valued and four-valued data types. There are no size limitations for arbitrary precision SystemC types.
- **Clocks**: SystemC has the notion of clocks (as special signals). Clocks are the timekeepers of the system during simulation. Multiple clocks, with arbitrary phase relationship, are supported.
- **Cycle-based simulation**: SystemC includes an ultra light-weight cycle-based simulation kernel that allows high-speed simulation.

## SystemC

Data Types

## Built in C/C++ Types

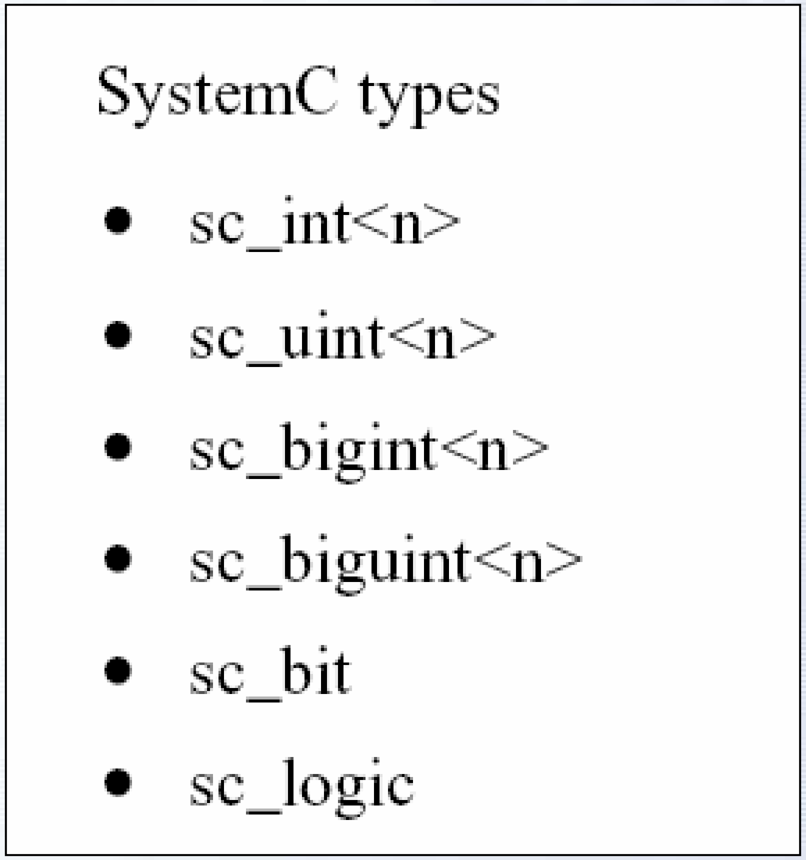
- System C supports all of the C/C++ data types.
- This means that existing algorithms can be brought directly into a SystemC design with little or no conversion.
- The limitation of the built in data types is in their range of values. For example an **int** cannot be ranged.


C++ built in types

- long
- int
- char
- short
- float
- double

## SystemC Data Types

- A number of new data types have been added as part of the SystemC library. These types reflect a more hardware style of data type, with some offering 4 value logic.



SystemC types

- sc_int<n>
- sc_uint<n>
- sc_bigint<n>
- sc_biguint<n>
- sc_bit
- sc_logic

## SC_BIT Data Type

- Type sc_bit is a two valued data type representing a single bit. A variable of type sc_bit can have the value ’0’(false) or ’1’(true) only.
- This type is useful for modeling parts of the design where Z (hi impedance) or X (unknown) values are not needed.
- Values are assigned using the character literals ’1’ and ’0’. When performing boolean operations type sc_bit objects can be mixed with the C/C++ bool type. Objects of type sc_bit are good for representing single bits of a design where logical operations will be performed.
- To declare an object of type sc_bit use the following syntax.

**sc_bit s;**

## SC_LOGIC

- A more general single bit type is sc_logic. This type has 4 values, ’0’(false), ’1’(true), ’X’ (unknown), and ’Z’ (hi impedance or floating). This type can be usedto model designs with multi driver busses, X propagation, startup values, and floating busses.

- Type sc_logic has the most common values used in VHDL and Verilog simulations at the RTL level.

- An example assignment is shown below:

```
sc_logic x; // object declaration
x = ’1’; // assign a 1 value
x = ’Z’; // assign a Z value
```

## SC_INT & SC_UINT

- Some systems need arithmetic operations on fixed size arithmetic operands. The Signed and Unsigned Fixed Precision Integer types provide this functionality in SystemC.

- The C++ int type is machine dependent, but usually 32 bits. If the designer were only going to use 32 bit arithmetic operations then this type would work. However the SystemC integer type provides integers from 1 to 64 bits in signed and unsigned forms.

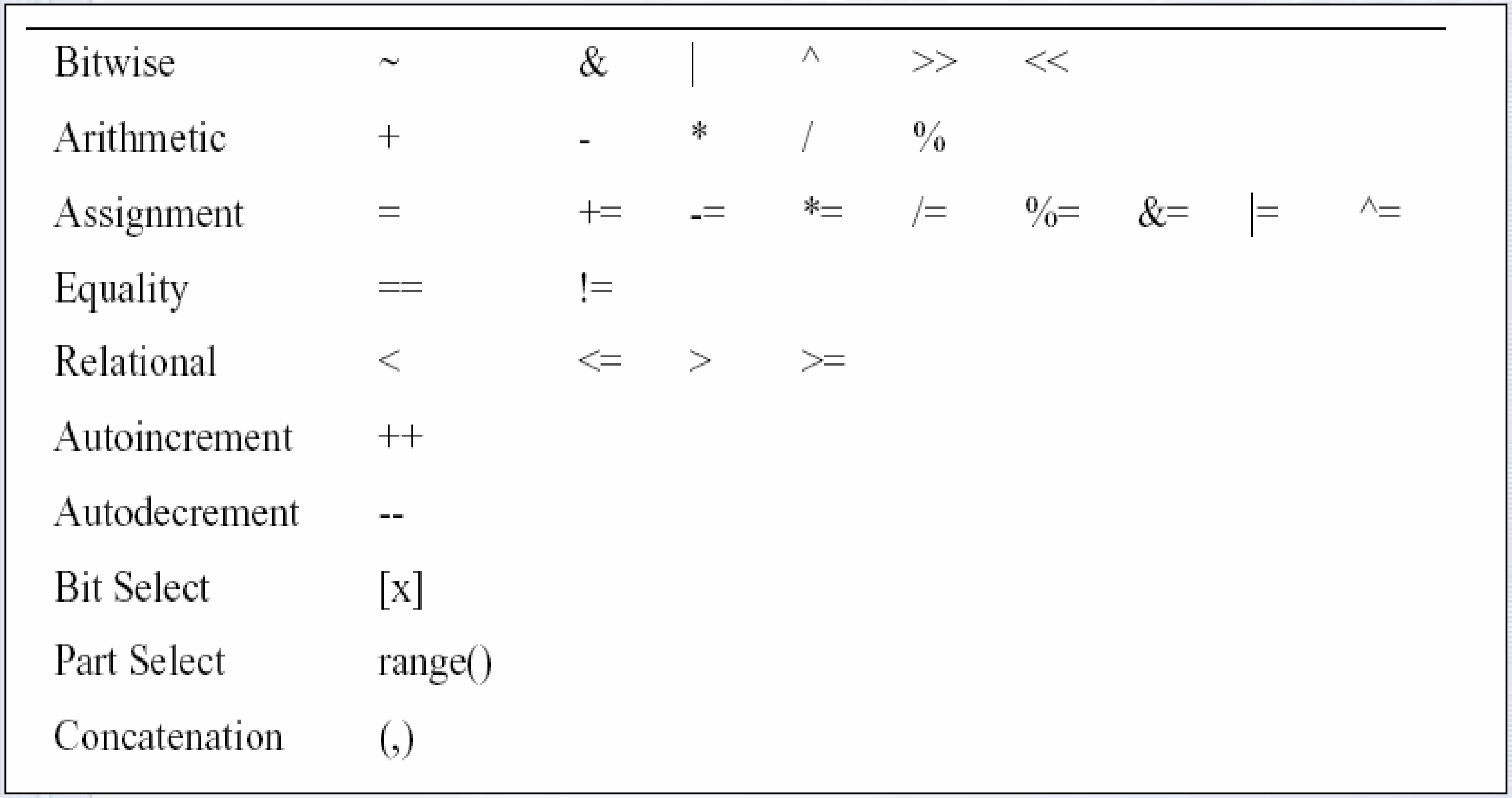
- The underlying implementation of the fixed precision type is a 64 bit integer. All operations are performed with a 64 bit integer and then converted to the appropriate result size through truncation. If the designer multiplies two 44 bit integers the maximum result size is 64 bits, so only 64 bits are retained. If the result is now assigned to a 44 bit result, 20 bits are removed.

## SC_INT and SC_UINT

- Type sc_int<n> is a Fixed Precision Signed Integer, while type sc_uint<n> is a Fixed Precision Unsigned Integer. The signed type is represented using a 2’s complement notation. The underlying operations use 64 bits, but the result size is determined at object declaration. For instance the following declaration declares a 64 bit unsigned integer and a 48 bit unsigned integer.

```
sc_int<64> x;
sc_uint<48> y;
```

## SC_INT & SC_UINT Operators



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Bitwise | ~ | & | \| | ^ | >> | << | | | |
| Arithmetic | + | - | * | / | % | | | | |
| Assignment | = | += | -= | *= | /= | %= | &= | \|= | ^= |
| Equality | == | != | | | | | | | |
| Relational | < | <= | > | >= | | | | | |
| Autoincrement | ++ | | | | | | | | |
| Autodecrement | -- | | | | | | | | |
| Bit Select | [x] | | | | | | | | |
| Part Select | range() | | | | | | | | |
| Concatenation | (,) | | | | | | | | |

## SC_BIGINT & SC_BIGUINT

- There are cases in HDL based design where operands need to be larger than 64 bits. For these types of designs sc_int and sc_uint will not work. For these cases use type sc_biguint (arbitrary size unsigned integer) or sc_bigint (arbitrary sized signed integer).

- These types allow the designer to work on integers of any size, limited only by underlying system limitations. Arithmetic and other operators also use arbitrary precision when performing operations. Of course this extra functionality comes at a price. These types execute more slowly than their fixed precision counterparts and therefore should only be used when necessary. While sc_bigint and sc_biguint will work with any operand sizes, they should only be used on operands larger than 64 bits or for operations where more than 64 bits of precision are required.

## SC_BIGINT & SC_BIGUINT

- Type sc_bigint is a 2’s complement signed integer of any size. Type sc_biguint is an unsigned integer of any size. When using arbitrary precision integers the precision used for the calculations depends on the sizes of the operands used. Look at the example below:

```
sc_biguint<128> b1;
sc_biguint<64> b2;
sc_biguint<150> b3;
b3 = b1*b2;
```

## SystemC

## Module Declaration

## Module Declaration

- A Functional Block can be declared using the SC_MODULE macro. This makes defining a C++ Class more HDL Like. Much like declaring an HDL module the ports, and member functions (Processes) are defined.
- The SC_CTOR constructor defines the sensitivity lists of the processes.

```
SC_MODULE(module_name) {
    // ports, data members, member functions
    // processes etc.
    SC_CTOR(module_name) {   // Constructor
    // body of constructor
    // process registration, sensitivity lists
    // module instantiations, port binding etc.
    }
};
```

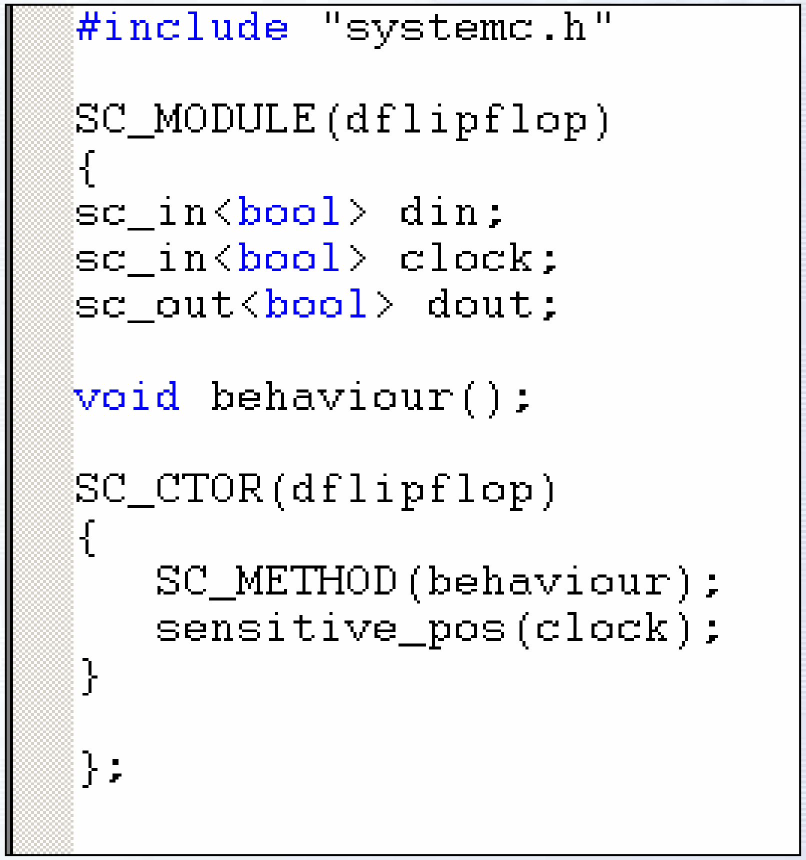
## Module Declaration

- A module declaration in SystemC is essentially the same as a Class definition in C++. It represents a functionality as an Entity/Architecture do in VHDL.

- The module declaration contains
  - Port Declarations
  - Local variable declarations if necessary
  - Method declarations (analogous to a process)
  - SC_CTOR which is the constructor which initializes variables, and specifies sensitivity lists.

- For example a module declaration for a D Flip Flop module would be as shown :

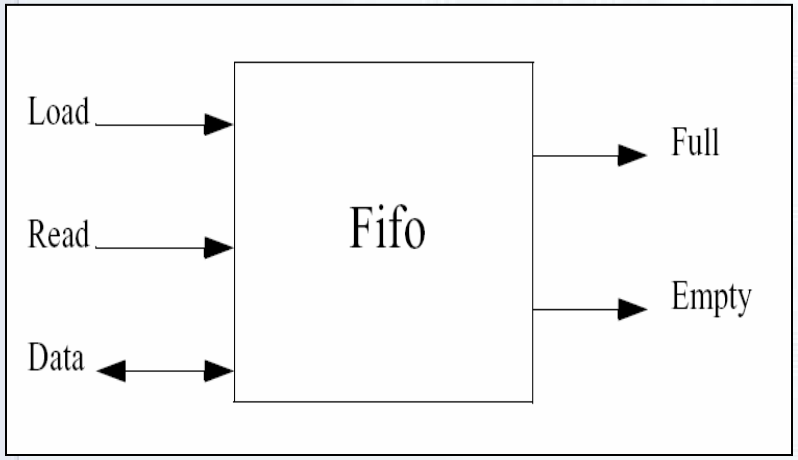
- The include file “systemc.h” is the interface to the SystemC library, and must be included in any file that contains references to SystemC functions.



```
#include "systemc.h"

SC_MODULE(dflipflop)
{
sc_in<bool> din;
sc_in<bool> clock;
sc_out<bool> dout;

void behaviour();

SC_CTOR(dflipflop)
{
   SC_METHOD(behaviour);
   sensitive_pos(clock);
}

};
```

## Module Ports

- Module Ports pass data to and from the processes of a module. You declare a port mode as **in**, **out**, or **inout**. You also declare the data type of the port as any C++ data type, SystemC data type, or user defined type.



```
SC_MODULE(fifo) {
    sc_in<bool> load;
    sc_in<bool> read;
    sc_inout<int> data;
    sc_out<bool> full;
    sc_out<bool> empty;
//rest of module not shown
}
```

## Member Functions

- A functionality is described using a member function, just as in C++. A member function is declared as part of the Module declaration.

- The actual functionality can be described inline with the module declaration or, as was shown earlier, just the prototype is defined in the modules declaration.

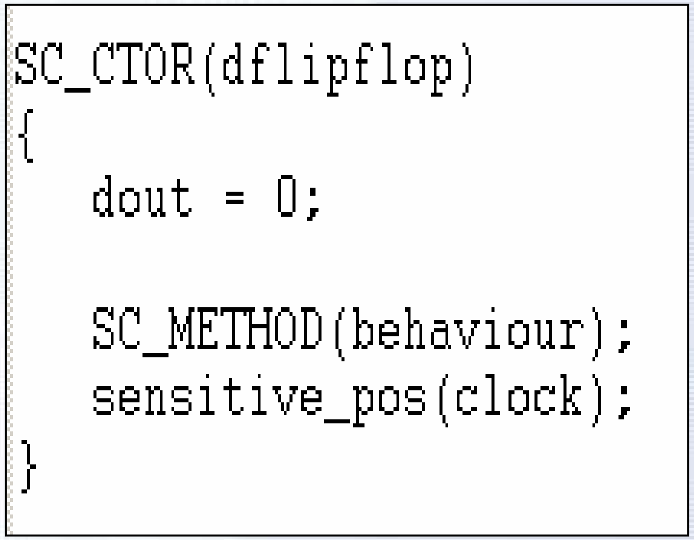
- The functionality is then defined separately.

```
#include "systemc.h"

SC_MODULE(dflipflop)
{
sc_in<bool> din;
sc_in<bool> clock;
sc_out<bool> dout;

void behaviour();

SC_CTOR(dflipflop)
{
   SC_METHOD(behaviour);
   sensitive_pos(clock);
}

};
```

```
#include "dflipflop.h"

void dflipflop::behaviour()
{
  dout = din;
};
```

## The Module Constructor

- The module constructor function SC_CTOR is where SystemC transforms C++ from a sequential language into a cycle based system.
- The SC_CTOR constructor serves double duty.
- It acts as a class constructor, giving initial values to the variables in the module (class)
- It also converts the member functions into *processes*.



```
SC_CTOR(dflipflop)
{
   dout = 0;

   SC_METHOD(behaviour);
   sensitive_pos(clock);
}
```

## Processes

- So far the interface of modules have been discussed, but not the part of the module that provides the functionality.

- The real work of the modules are performed in processes. Processes are functions that are identified to the SystemC kernel and called whenever signals these processes are “sensitive to” change value.

- A process contains a number of statements that implement the functionality of the process. These statements are executed sequentially until the end of the process occurs, or the process is suspended by one of the wait function calls.

## SC_METHOD Process

- This behavior is described by the following statements in the constructor for module dflipflop

```
SC_METHOD(behaviour);
sensitive_pos(clock);
```

- The first statement specifies that this process is an SC_METHOD process.  An SC_METHOD process is triggered by events and executes all of the statements in the method before returning control to the SystemC kernel.
- The second statement specifies that the process is sensitive to positive edge changes on input port clock.
- The process runs once when the first event (positive edge on clock) is received. It executes the assignment of din to dout and then returns control to the SystemC kernel.
- Another event causes the process to be invoked again, and the assignment statement is executed again. This is analogous to a process block in a VHDL design.

## SC_METHOD Process

- The allowable sensitivities for the SC_METHOD process are
  - **sensitive_pos** -- denotes positive edge
  - **sensitive_neg** -- denotes negative edge
  - **sensitive** -- level sensitivity for combinatorial behaviour.
- There are two notations that can be used in the sensitivity list. The first has been shown is
  - **sensitive_pos(clock)**
- The other way this can be specified is by using the **<<** (stream) operator, as in
  - **sensitive_pos << clock**
- Both are identical in functionality.
- For the sensitive level sensitivity, multiple arguments can be passed
  - **sensitive << a << b**

## Multiple Sensitivities

- If the process is sensitive to more than one event, then multiple sensitive() statements are required.

```
SC_MODULE(my_module) {
    sc_event c;
    sc_event d;
    void my_thread_proc();
    SC_CTOR(my_module) {
       SC_THREAD(my_thread_proc);
          // declare static sensitivity list
       sensitive(c); // sensitive to event c
       sensitive(d); //sensitive to event d
    }
    // rest of module not shown
};
```

## Multiple Methods in a Module

- A Module can consist of many methods, just like a VHDL Architecture can consist of many processes.
- When multiple processes are declared, the pattern is declaration followed by sensitivity list followed by declaration followed by sensitivity list and so on.

```
SC_MODULE(my_module) {
    sc_event c, d;
    void proc_1();
    void proc_2();
    void proc_3();
    SC_CTOR(my_module) {
       SC_THREAD(proc_1);
        sensitive << c << d; //proc_1 sensitive to c & d
       SC_THREAD(proc_2); // no static sensitivity
       SC_THREAD(proc_3);
        sensitive << d ; // proc_3 sensitive to d
    }
    // rest of module not shown
};
```

## Thread Processes

- Thread Processes can be suspended and reactivated. The 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().

- The input signals that cause the process to reactivate are specified by the sensitivity list. The sensitivity list is specified in the module constructor with the same syntax used in the Method Process.

## Thread Processes

```
#include "systemc.h"
SC_MODULE(traff) {
// input ports
     sc_in<bool> roadsensor;
     sc_in<bool> clock;
// output ports
     sc_out<bool> NSred;
     sc_out<bool> NSyellow;
     sc_out<bool> NSgreen;
     sc_out<bool> EWred;
     sc_out<bool> EWyellow;
     sc_out<bool> EWgreen;
void control_lights();

int i;
// Constructor
SC_CTOR(traff) {
  SC_THREAD(control_lights); // Thread Process
    sensitive << roadsensor;
    sensitive_pos << clock;
   }
 };  // end SC_MODULE
```

```
 void traff::control_lights() {
      NSred = false;
      NSyellow = false;
      NSgreen = true;
      EWred = true;
      EWyellow = false;
      EWgreen = false;
     while (true) {
       while (roadsensor == false)
          wait();
        NSgreen = false; // road sensor triggered
        NSyellow = true; // set NS to yellow
        NSred = false;
        For (i=0; i<5; i++)
            wait();
// rest of design…..
```

# SystemC

## Top Level (Testbench)

## Top Level

- The sc_main(), function is the entry point from the SystemC library to the user’s code. It is called by the function main() which is part of the SystemC library. Its prototype is:

**int sc_main( int argc, char* argv[] );**

- The arguments argc and argv[] are the standard command-line arguments. They are passed to sc_main() from main() in the library.

- The body of sc_main() typically consists of configuring simulation variables (default time unit, time resolution, etc.), Instantiation of the module hierarchy and channels, simulation, clean-up and returning a status code.

## Sc_signals

- After the sc_main statement, the local signals are declared to connect the module ports together.
- The systemC signal can have any data type, just like the port connections.

```
  sc_signal<packet_type> PACKET1, PACKET2, PACKET3,
PACKET4;
  sc_signal<long> DOUT;
  sc_signal<bool> TIMEOUT, START;
```

## Module Instantiation

- A module is instantiated in the sc_main() function in SystemC in a way very similar to the methodology seen in VHDL.

- **Note:** A module may also be instantiated from within another module. This is directly derived from a class using another class within its structure.  The syntax for doing this is beyond the scope of the present presentation.

```
transmit t1("transmit");
t1.tpackin(PACKET2);
t1.timeout(TIMEOUT);
t1.tpackout(PACKET1);
t1.start_timer(START);
t1.clock(CLOCK);

channel c1("channel");
c1.tpackin(PACKET1);
c1.rpackin(PACKET3);
c1.tpackout(PACKET2);
c1.rpackout(PACKET4);
```

## Clock Objects

- Clock objects are special objects in SystemC. They generate timing signals used to synchronize events in the simulation. Clocks order events in time so that parallel events in hardware are properly modeled by a simulator on a sequential computer.

- 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 following syntax:

```
sc_clock clock1("clock1", 20, 0.5, 2, true);
```

- This declaration will create a clock object named clock with a period of 20 time units, a duty cycle of 50%, the first edge will occur at 2 time units, and the first value will be true. 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.

## Clock Objects

- 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. This allows areas of the design or the entire design to be synchronized by the same clock. In the example below the sc_main routine of a design creates a clock and connects the clock to instantiated components within the main module.

```
int sc_main(int argc, char*argv[]) {
  sc_signal<int>  val;
  sc_signal<sc_logic> load;
  sc_signal<sc_logic> reset;
  sc_signal<int> result;

  sc_clock  ck1("ck1", 20, 0.5, 0, true);

  filter f1("filter");
  f1.clk(ck1.signal());
  f1.val(val);
  f1.load(load);
  f1.reset(reset);
  f1.out(result);

  // rest of sc_main not shown
```

## SC_START Function

- Once the instantiation of the lower level modules has been coded, and the clocks setup, the simulation is moved forward using the sc_start method. If an argument is given, then the simulation will move forward by that many time ticks. If an argument of -1 is given then the simulation will run forever.

```
#include "counter1.h"

int sc_main(int argc, char* argv[]) {

sc_signal<bool> Rst;
sc_signal<sc_int<8> > cval;

sc_clock CLOCK("clock", 20);

counter1 U1 ("count1");
U1.Clk(CLOCK);
U1.Reset(Rst);
U1.Ctr_Val(cval);


sc_start(500);


return (0);

}
```

## Another Way of Simulating

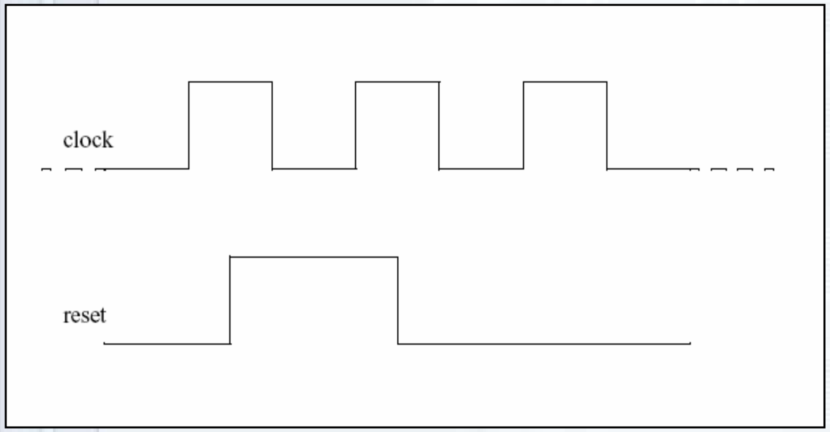
- Another method that can be used to simulate the design is to use the sc_cycle function to move the simulation along.
- In this example a 20 cycle period clock is defined using a loop.
- This methodology allows for more complex simulations, as the designer can control what happens when.

```
sc_signal<bool> clock;

sc_initialize();
for (int i = 0; i <= 200; i++)
   clock = 1;
   sc_cycle(10);
   clock = 0;
   sc_cycle(10);
}|
```

## Another Way of Simulating

- For example, using this method an asychronous signal can be generated easily.
- There are other ways of generating wave shapes and monitors, but those are beyond the scope of this presentation.

```
sc_initialize();
// Let the clock run for 10 cycles
for (int i = 0; i <= 200; i++)
   clock = 1;
   sc_cycle(10);
   clock = 0;
   sc_cycle(10);
}

// Inject asynchronous reset
clock = 1;
sc_cycle(5);
reset = 1;
sc_cycle(5);
clock = 0;
sc_cycle(10);
clock = 1;
sc_cycle(5);
reset = 0;
```



## SystemC

Graphical Output

## Signal Tracing

- Since SystemC is built upon C/C++ there is the whole language worth of functions to generate console and file I/O.
- However, one of the things that we like is graphical output. In order to accomplish this, SystemC has a series of trace functions which create a VCD file.
- VCD Files are text files in a known format that have been around since the time of plotters. There are many programs available that can read VCD files and Display them.
- The graphical system in the Aldec tools for example can display VCD files.

## Creating a Trace File

- The first step in tracing waveforms is creating the trace file. The trace file is usually created at the top level after all modules and signals have been instantiated. Fortracing waveforms using the VCD format, the trace file is created by calling the sc_create_vcd_trace_file() function with the name of the file as an argument. This function returns a pointer to a data structure that is used during tracing. For example,

```
sc_trace_file * my_trace_file;
my_trace_file = sc_create_vcd_trace_file(“my_trace”);
```

- creates the VCD file named my_trace.vcd (the .vcd extension is automatically added). A pointer to the trace file data structure is returned. You need to store this pointer so it can be used in calls to the tracing routines.

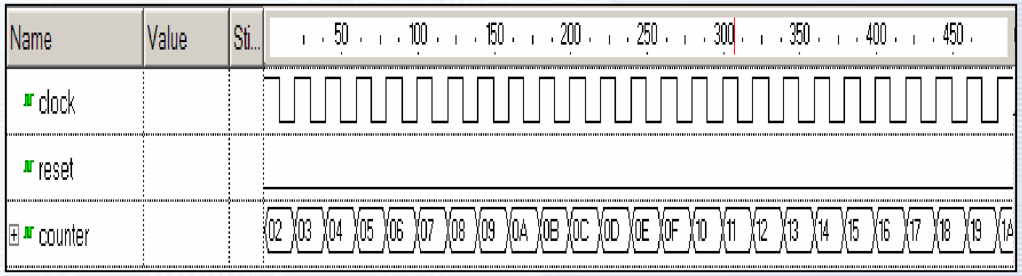
## Creating a Trace File

- SystemC provides tracing functions for scalar variables and signals. All tracing functions have the following in common:
  - The function is named sc_trace().
  - Their first argument is a pointer to the trace file data structure sc_trace_file.
  - Their second argument is a reference or a pointer to a variable being traced.
  - Their third argument is a reference to a string.
- For example, the following illustrates how a signal of type int and a variable of typefloat are traced.

```
sc_signal<int> a;
float b;

sc_trace(trace_file, a, “MyA”);
sc_trace(trace_file, b, “B”);
```

## Example

```
#include "counter1.h"

int sc_main(int argc, char* argv[]) {

sc_signal<bool> Rst;
sc_signal<sc_int<8> > cval;

sc_clock CLOCK("clock", 20);

counter1 U1 ("count1");
U1.Clk(CLOCK);
U1.Reset(Rst);
U1.Ctr_Val(cval);

// trace file

sc_trace_file *tf = sc_create_vcd_trace_file
("simplex");
// External Signals
sc_trace(tf, CLOCK.signal(), "clock");
sc_trace(tf, Rst, "reset");
sc_trace(tf, cval, "counter");

sc_start(500);

sc_close_vcd_trace_file(tf);

return (0);

}
```

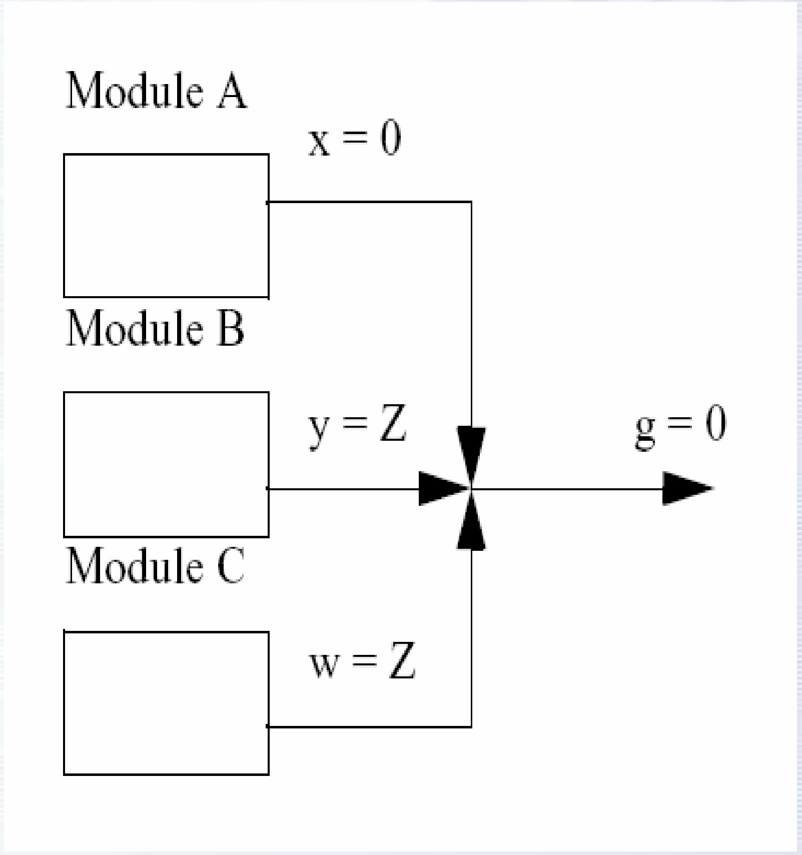

Name
Value
Sti...
50
100
150
200
250
300
350
400
450
clock
reset
counter
02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A

## SystemC

## Resolved Signals

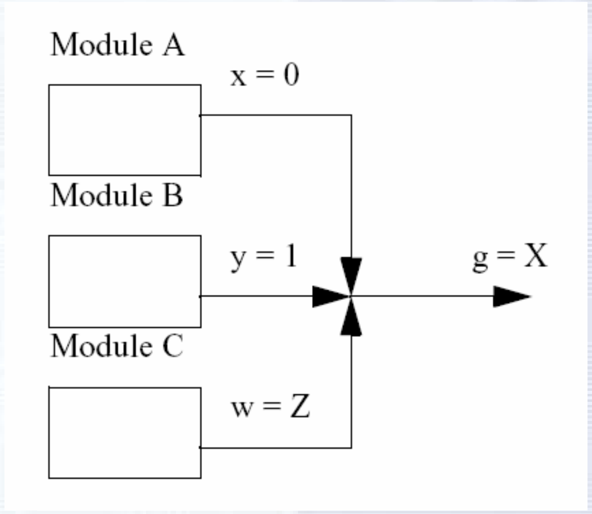
## Signal Resolution

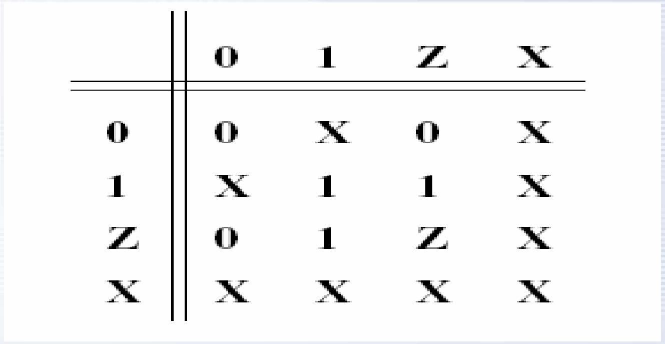
- Bus resolution becomes an issue when more than one driver is driving a signal.
- SystemC uses a Resolved Logic Vector signal type to handle this issue. Take a look at the example with three drivers x, y, w, driving signal g.
- Port x is driving a 0 value, and ports y and w are driving Z values. The resolution of these values will be assigned to signal g. In this example the resolved value will be 0. Ports y and w have their drivers disabled and are driving Z values. Therefore the 0 value from port x will “win”.



## Signal Resolution

- In this case ports x and y are driving a value while port w is not. However ports x and y are driving opposite values. Since values 0 and 1 are the same strength or priority the final value of signal g cannot be determined and the value assigned will be X.





| | 0 | 1 | Z | X |
|---|---|---|---|---|
| 0 | 0 | X | 0 | X |
| 1 | X | 1 | 1 | X |
| Z | 0 | 1 | Z | X |
| X | X | X | X | X |

## Resolved Vector Type

- The following shows how to create a resolved Module port.
- The only limitation on the size of n is underlying system limitations. Resolved Logic Vector ports should only be used where absolutely necessary as extra simulation overhead is added versus standard ports. Typically a standard port with a scalar or vector type should be used for better simulation efficiency.

```
sc_in_rv<n> x; //input resolved logic vector n bits wide

sc_out_rv<n> y;// output resolved logic vector n
               //bits wide

sc_inout_rv<n> z;  // inout resolved logic vector n
                   //bits wide
```

## Resolved Signal Type

- Signals are used to interconnect ports. Vector signals can be used to connect vector ports. The vector signal types are the same as the vector port types. The currently supported vector signal type is sc_signal_rv. This is a resolved vector of sc_logic signals.
- An example is shown below:

**sc_signal_rv<n> sig3; // resolved logic vector signal n bits wide**

- Signals of this type can be used to connect to resolved logic vector ports.

## SystemC

Interfacing to VHDL

## Interfacing to VHDL

- SystemC was developed with the idea of creating an executable program that is run from within a C/C++ environment.

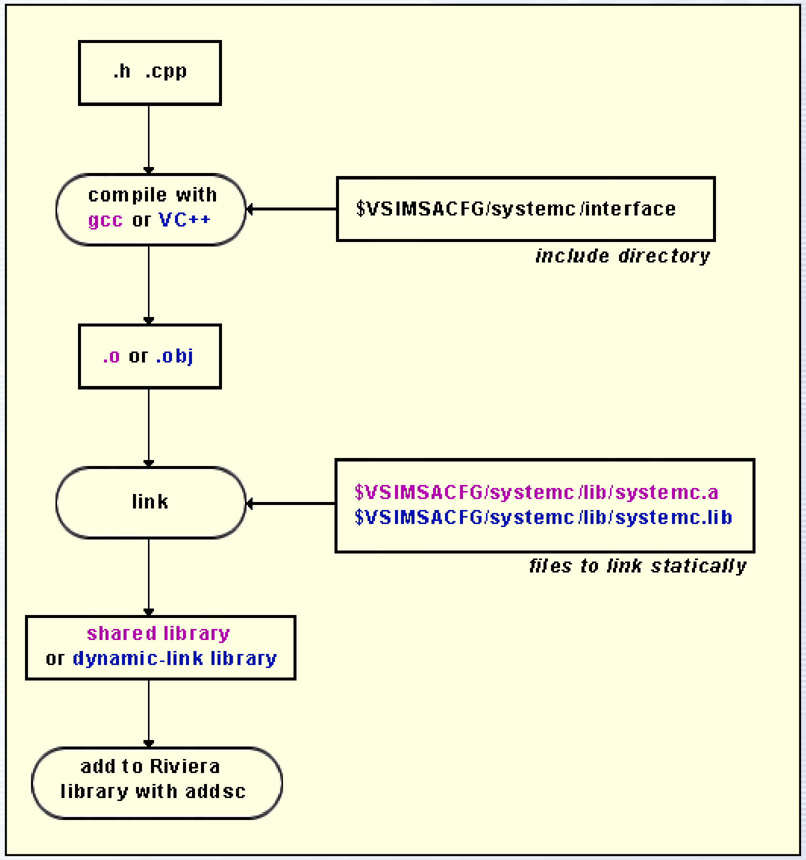
- However, users that develop models or testbenches using systemC may wish to interface these models with their other VHDL (synthesiziable) code. Aldec's Riviera simulator allows the user to instantiate a systemC module directly from VHDL or Verilog Code.

## Riviera Uses Library Instantiation

- Before you can simulate SystemC modules in Riviera, you should link your application. You need
- a shared object (**.so**) library on Linux
- a dynamic-link library (**.dll**) on Windows
- Riviera includes header files required for compilation and a SystemC library (**systemc.a** or **systemc.lib**) needed at the link stage. Compiling and linking is described both for gcc and Microsoft Visual C++.
- The source code of the SystemC library was modified by Aldec to allow seamless integration of SystemC models into an HDL flow. When compiling your SystemC application, you should use the header files from the **systemc/interface** directory in the Riviera installation directory. Likewise, you should use the **systemc.a** or the **systemc.lib** file from the **systemc/lib** directory for linking.

## SystemC Flow

- The diagram beside shows SystemC flow. SystemC modules should be compiled with a C++ compiler (gcc or VC++).
- You should use include files from the **systemc/interface** directory in the Riviera installation directory. (The installation directory is represented by the **$VSIMSACFG** variable on the diagram.)
- The header files delivered with Riviera are modified compared to the original SystemC distribution. A number of changes were required for a seamless integration with the HDL flow.
- Likewise, you should use the SystemC library delivered with Riviera for linking. Use **systemc.a** for gcc or **systemc.lib** for VC++.

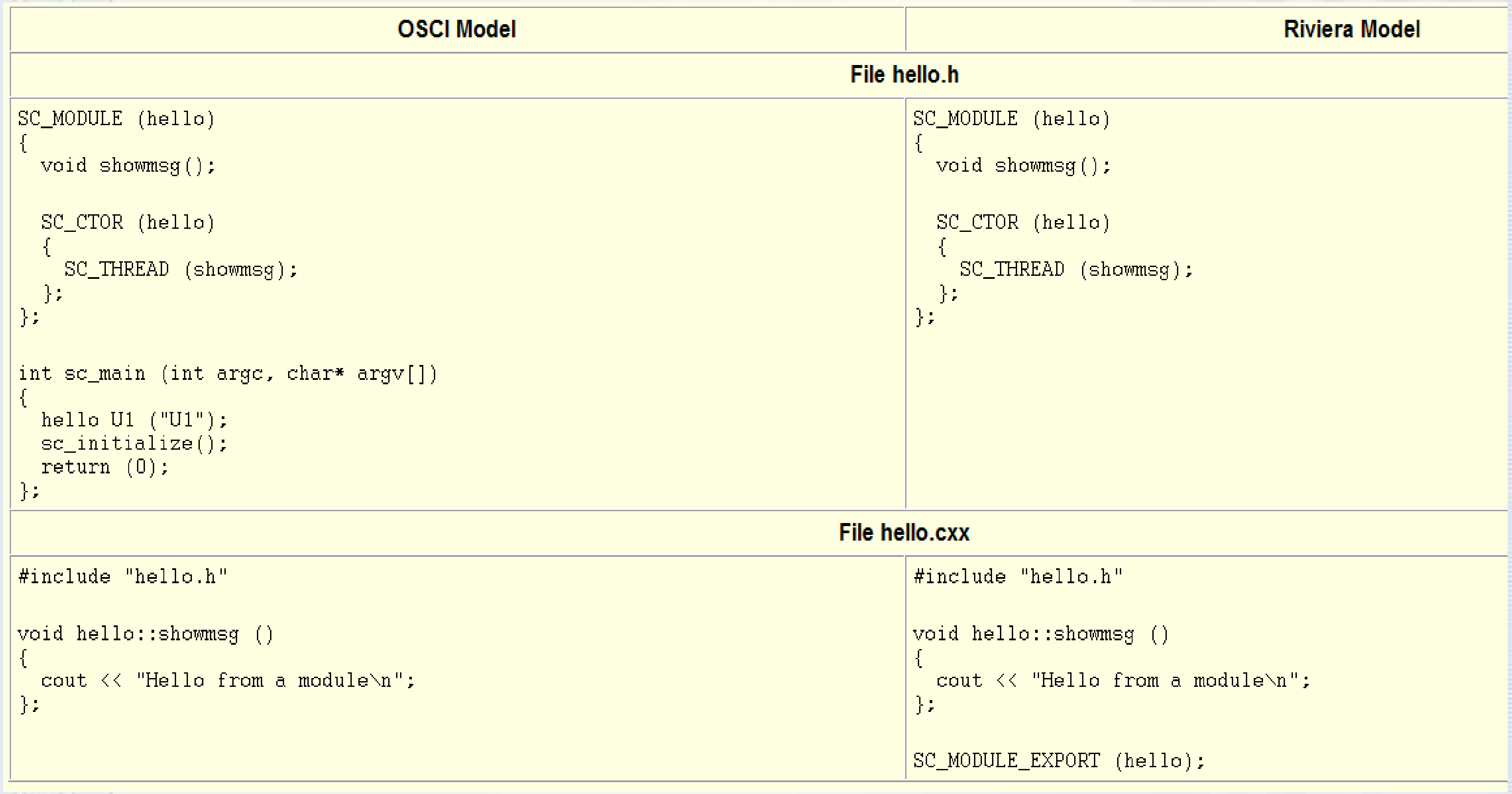


.h .cpp
compile with gcc or VC++
$VSIMSACFG/systemc/interface
include directory
.o or .obj
link
$VSIMSACFG/systemc/lib/systemc.a
$VSIMSACFG/systemc/lib/systemc.lib
files to link statically
shared library or dynamic-link library
add to Riviera library with addsc

## No SC_MAIN() used

- When SystemC and HDLs are mixed into one design and simulated in Riviera, a number of modifications to the SystemC code (and the build process) is necessary. The SystemC code should be compiled to a shared object (**.so**) or a dynamic-link (**.dll**) library. Riviera will never call the **sc_main()** function from your module. This has the following implications:
- 
- The **sc_main()** function must be replaced by a module.
- 
- Calls to **sc_set_time_resolution()** should be removed. Simulation resolution can be set in Riviera.
- 
- Calls to **sc_start()** function should be removed. Simulation can be controlled with the **run** command in Riviera.
- Testbench code mixed with multiple invocations of the **sc_start()** functions can be moved to **SC_THREAD()** process. Process execution can be controlled with calls to the **wait()** function.
- 
- SystemC modules that should become available in Riviera (after using the **addsc** command) should be exported with the **SC_MODULE_EXPORT (sc_module_name)** macro.

# Example Modification For Riviera



| OSCI Model | Riviera Model |
|---|---|

**File hello.h**

OSCI Model:

```
SC_MODULE (hello)
{
  void showmsg();

  SC_CTOR (hello)
  {
    SC_THREAD (showmsg);
  };
};

int sc_main (int argc, char* argv[])
{
  hello U1 ("U1");
  sc_initialize();
  return (0);
};
```

Riviera Model:

```
SC_MODULE (hello)
{
  void showmsg();

  SC_CTOR (hello)
  {
    SC_THREAD (showmsg);
  };
};
```

**File hello.cxx**

OSCI Model:

```
#include "hello.h"

void hello::showmsg ()
{
  cout << "Hello from a module\n";
};
```

Riviera Model:

```
#include "hello.h"

void hello::showmsg ()
{
  cout << "Hello from a module\n";
};

SC_MODULE_EXPORT (hello);
```

## Sample Execution

- The Riviera model is built to a shared-object (**.so**) or a dynamic-link (**.dll**) library. When the library is created, modules exported with the **SC_MODULE_EXPORT** macro must be added to Riviera library with the addsc command. The following commands can be used to run simulation:

- Note that the output of the **cout** stream will appear in Riviera console. (The output of **printf** function would be printed to the underlying terminal where Riviera was started.)

```
# Create library
alib work
set worklib work

# Import SystemC modules
addsc libhello.so

# List library contents
adir
# Library contents will be printed:
#   systemc module: hello

# Start simulation
asim hello

# Run simulation
run -all

# SystemC module will print the following message:
#   Hello World
```

# System C Lab #1 – Up/Down Counter

## Introduction

In this lab we are going to develop a SystemC model of an enabled 8 bit Up/Down Counter. We will then develop a main program (testbench) for the design. The design will then be compiled an executed in a **Cygwin** (unix emulation) shell in windows.

For the purposes of the design, the counter will be active on the positive edge of the clock, the reset will be active high and asynchronous, and the updown control will be active high (count up when 1).

## Starting the Lab

Using the windows explorer, browse to the **c:\labs\systemc\updown** folder. In this folder you will find 4 files which are the templates (starting point) for the project we are going to implement. The four files are as follows:

| | |
|---|---|
| **Updown.h** | This file contains the declaration of the updown counter method. |
| **Updown.cpp** | This file contains the implementation of the member functions (processes) for the updown counter. |
| **Main_updown.cpp** | This file is the top level, which implements the testbench for the updown counter. |
| **Makefile** | This is the makefile that compiles the project. |

You can edit these files with any editor available on the computer (*notepad* for example).

## Creating updown.h

The file updown.h is where the module (class) for the updown counter  is defined. Open up the file updown.h

```
#include "systemc.h"

SC_MODULE(updown)
{
      sc_in< bool >  P_clk;
      sc_in< bool >  P_reset;
      sc_in< bool >  P_updown;
      sc_out<sc_int<8> > P_ctr_val;

      int int_ctr_s;
      void behaviour();
      void print_res();

      SC_CTOR(updown)
      {
            int_ctr_s = 0;
// SC_METHOD code to be inserted here
      }
};
```

The file has already been blocked out. We are defining that a module called **updown** is being created. We see that there are 4 input ports (**P_clk**, **P_reset**, **P_updown**) and one output port (*8 bits wide*) (**P_ctr_val**).

An internal signal **int_ctr_s** has been declared to be the count variable. The existence of two functions has also been declared. The function **behaviour()** contains the functionality of the counter, and the function **print_res()** has been added for debugging, and it will print out the values of the counter variables.

What needs to be added to the declaration of the updown method is the sensitivity lists for **behaviour** and **print_res** to the updown **SC_TOR** (constructor).

**DESIGN STEP: Using the SC_METHOD, sensitive and sensitive_pos functions make the behaviour() function sensitive to the positive edge of P_clk and P_reset. Make the print_res() function sensitive to the P_clk, P_reset and P_updown signals.**

Save the file updown.h. We will compile this file in a future step.

## Creating updown.cpp

The next step is to create the functionality of the the two methods that have been declared in the updown.h header file.

Open up the file updown.cpp

```
#include "updown.h"

void updown::behaviour()
{

    // insert code for an updown counter here.

    P_ctr_val = int_ctr_s ;
}


void updown::print_res()
{
    cout << sc_simulation_time() << "  "<< P_reset << "  " << P_clk  <<
"  "
    << "  " << P_updown << "  " << int_ctr_s  << "\n";
}
```

The function **print_res** has already been defined in the file. It uses a simple C++ COUT directive in order to print all of the values of the variables in the updown counter. The function **sc_simulation_time()** returns the current simulation time. There is nothing that has to be done for this function.

The function **behaviour()** has been blocked out for you. The declaration has been taken care of, along with the final assignment of the internal count variable **int_ctr_s** to the module output **P_ctr_val**.

**DESIGN STEP: implement the code for the updown counter behaviour() function. The reset is active high, and the 8 bit counter (rolls over at 255) counts up when P_updown is 1 and down when P_updown is low.**

Save the file. It will be compiled in future step.

## Creating main_updown.cpp

The main_updown.cpp file is the main program for our design. Open up main_updown.cpp

```
#include "systemc.h"
#include "updown.h"

int sc_main(int argc, char* argv[]) {

  sc_signal<bool>              Rst;
  sc_signal< sc_int<8> >       cval;
  sc_signal<bool>              up_down;
  sc_signal<bool>              clk;

  int i;  // loop counter


  // initialize the input values


  Rst = 1;
  up_down = 1 ;
  clk = 0;

  // insert the instatiation here


 // install test values here

}
```

Two things have to be included in the **sc_main()** top level

1. Instantiation of the updown counter
2. Stimulus for the clk, rest and updown values

The four signals have been declared that are needed for the instantiation.

The test signals are going to be written in a very HDL style, that is, the time advancement and signals values are going to be controlled manually.

The following systemc values will be useful.

1. **sc_initialize()** to initialize the simulation
2. **sc_cycle(time_value)** which progresses the simulation by *time_value* time ticks.
3. **sc_stop()** ends the simulation.

A structure like;

```
for (i = 0; i<= 5; i++)
  {
        clk = 1;
        sc_cycle(10);
        clk = 0;
        sc_cycle(10);
  };
```

can be used to create clock cycles.  You must remember that the sc_main runs sequentially, so you must set the rst, up_down values, run the simulation for several clock cycles, change them, more clock cycles….

**DESIGN STEP:  Add the instantiation of the updown module and some test inputs to the main program.**

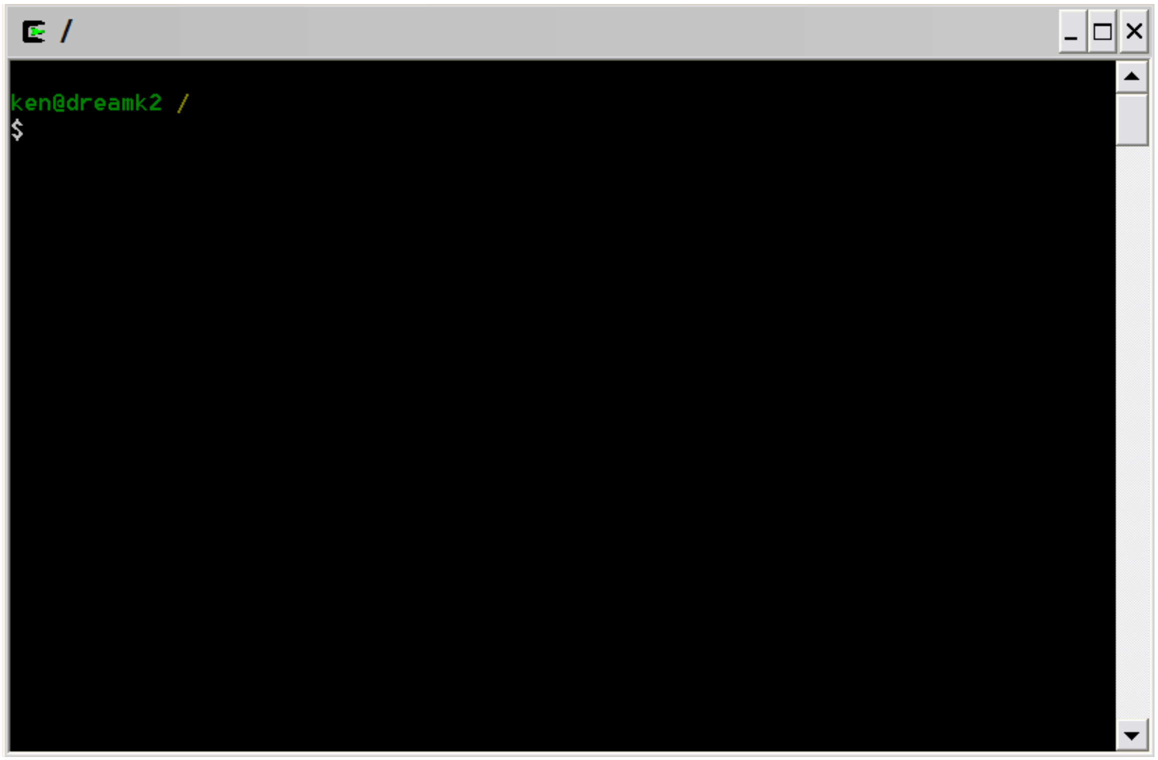
Save the file, we will compile it in the next step.

## Compiling and running the files

As mentioned in the introduction, we are going to compile and run the program using the CYGWIN shell. CYGWIN is a linux emulator for windows.

Find the Cygwin shell icon on the desktop.

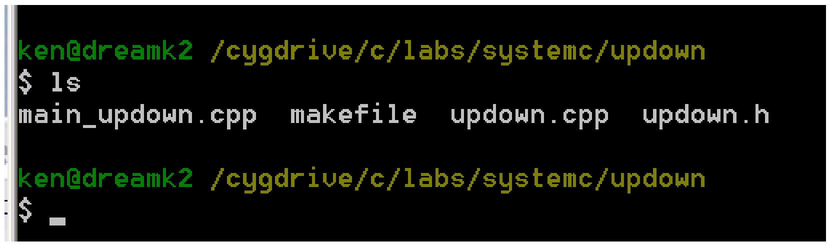
This will open up a command shell for the unix shell commands (next page).



```
ken@dreamk2 /
$
```

Type **cd /cygdrive/c/labs/systemc/updown** in order to go to the project directory.

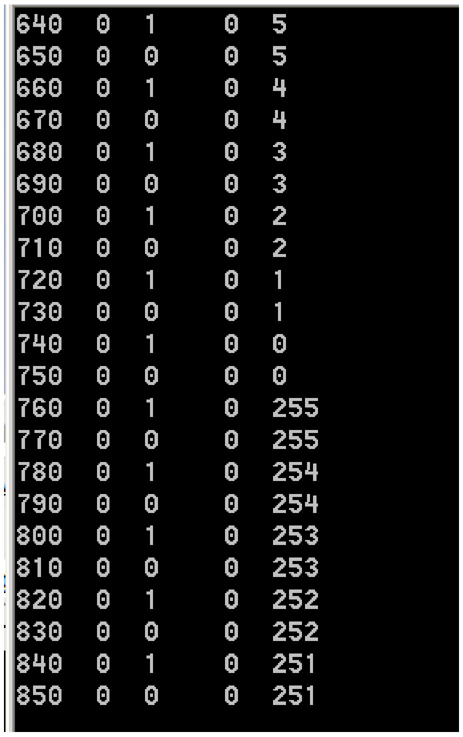
Type **ls** and make sure all of the files are there.



```
ken@dreamk2 /cygdrive/c/labs/systemc/updown
$ ls
main_updown.cpp  makefile  updown.cpp  updown.h

ken@dreamk2 /cygdrive/c/labs/systemc/updown
$ _
```

In order to make compiling easier, a makefile has been included. In order to compile the design, type **make** at the command prompt. This creates the executable **updown_run.exe.** You can review and fix any syntax errors. If you wish to delete the executable and the object files before compiling, type **make clean**. This deletes all of the intermediate files.

Once the syntax errors have been fixed and the design compiles (there may be a few warnings), type **updown_run.exe** at the command prompt in order to run the program.



```
640  0  1    0  5
650  0  0    0  5
660  0  1    0  4
670  0  0    0  4
680  0  1    0  3
690  0  0    0  3
700  0  1    0  2
710  0  0    0  2
720  0  1    0  1
730  0  0    0  1
740  0  1    0  0
750  0  0    0  0
760  0  1    0  255
770  0  0    0  255
780  0  1    0  254
790  0  0    0  254
800  0  1    0  253
810  0  0    0  253
820  0  1    0  252
830  0  0    0  252
840  0  1    0  251
850  0  0    0  251
```

You will see the printed results on the console.

## End of lab

You can now experiment either by changing the functionality of the counter, or by adding features such as the VCD function calls in order to create a VCD file to make a graphical output.

# Appendix

## Updown.h

```
#include "systemc.h"

SC_MODULE(updown)
{
      sc_in< bool >  P_clk;
      sc_in< bool >  P_reset;
      sc_in< bool >  P_updown;
      sc_out<sc_int<8> > P_ctr_val;

      int int_ctr_s;
      void behaviour();
      void print_res();

      SC_CTOR(updown)
      {
            int_ctr_s = 0;

            SC_METHOD(behaviour);
            sensitive_pos << P_clk;
            sensitive_pos << P_reset;

            SC_METHOD(print_res);
            sensitive << P_clk << P_reset << P_updown;
      }

};
```

## updown.cpp

```
#include "updown.h"

void updown::behaviour()
{
      if (P_reset)
      int_ctr_s = 0;
      else
      {
            if (P_updown)
            {
              if (int_ctr_s == 255)
                    int_ctr_s = 0;
              else
                    int_ctr_s++;
          }
        else
        {
                    if (int_ctr_s == 0)
                      int_ctr_s = 255;
                    else
                      int_ctr_s--;
          }


      }

    P_ctr_val = int_ctr_s ;
}


void updown::print_res()
{
      cout << sc_simulation_time() << "  "<< P_reset << "  " << P_clk
<< "  "
        << "  " << P_updown << "  " << int_ctr_s  << "\n";
}
```

## main_updown.cpp

```
#include "systemc.h"
#include "updown.h"

int sc_main(int argc, char* argv[]) {

  sc_signal<bool>   Rst;
  sc_signal< sc_int<8> >  cval;
  sc_signal<bool>     up_down;

 // sc_clock clk("clock", 20);
  sc_signal<bool> clk;

  int i;

  Rst = 1;  // power on reset
  up_down = 1 ;

  updown U1 ("updown");
  U1.P_clk(clk);
  U1.P_reset(Rst);
  U1.P_updown(up_down);
  U1.P_ctr_val(cval);

  // trace file

  sc_trace_file *tf = sc_create_vcd_trace_file
    ("updown_wave");
  // External Signals
  sc_trace(tf, clk, "clock");
  sc_trace(tf, Rst, "reset");
  sc_trace(tf, cval, "counter");
  sc_trace(tf, up_down, "updown");


  // sc_start(500);

  sc_initialize();

  for (i = 0; i<= 5; i++)
  {
        clk = 1;
        sc_cycle(10);
        clk = 0;
        sc_cycle(10);
  };

  Rst = 0;

  for (i = 0; i<= 15; i++)
    {
        clk = 1;
        sc_cycle(10);
        clk = 0;
```

```
            sc_cycle(10);
        };


      up_down =  0;

      for (i = 0; i<= 20; i++)
        {
            clk = 1;
            sc_cycle(10);
            clk = 0;
            sc_cycle(10);
        };



      sc_close_vcd_trace_file(tf);

      sc_stop();

      return (0);


    }
```

**makefile**

```
CC     = /usr/bin/gcc
CPP    = /usr/bin/g++
PATH   = c:/labs/systemc/updown
TARGET  = updown_run
OBJPATH      = $(PATH)
LD     = /usr/bin/ld
LD_PATH = /usr/bin/ld

OBJECTS = updown.o main_updown.o

TARGETS = updown main_updown

INCDIRS = -IC:/systemc/systemc/include

LIBS    = C:/systemc/systemc/lib-cygwin/libsystemc.a
/usr/lib/libstdc++.a

LIBDIRS      =

CC_OPT       = -Wno-deprecated -ggdb -shared -Wall  -D__int64=long\
long\ int
LD_OPT       =

target : $(TARGETS)
      $(CC) -W1,--noinhibit-exec, $(OBJECTS) $(LIBS) -o $(TARGET)

updown : $(PATH)/updown.cpp $(PATH)/updown.h
      ${CC} $(CC_OPT) $(INCDIRS) -c -o $(OBJPATH)/updown.o
$(PATH)/updown.cpp

main_updown   : $(PATH)/main_updown.cpp
      ${CC} $(CC_OPT) $(INCDIRS) -c -o $(OBJPATH)/main_updown.o
$(PATH)/main_updown.cpp

clean :
      /usr/bin/rm -f *.o
      /usr/bin/rm -f *.exe
      /usr/bin/rm -f *.vcd
```

## *Cygwin Package Install*

1. Launch Cygwin's `setup.exe`
2. Choose *Download from Internet* or *Install from Internet*

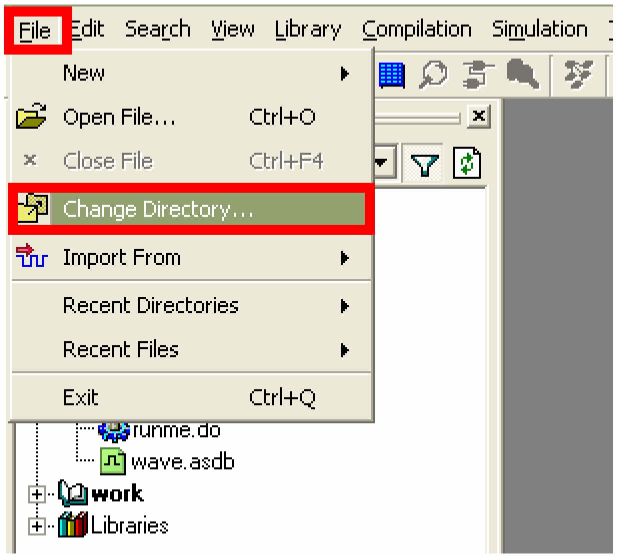
# *Riviera Lab – SystemC/HDL Instantiation*

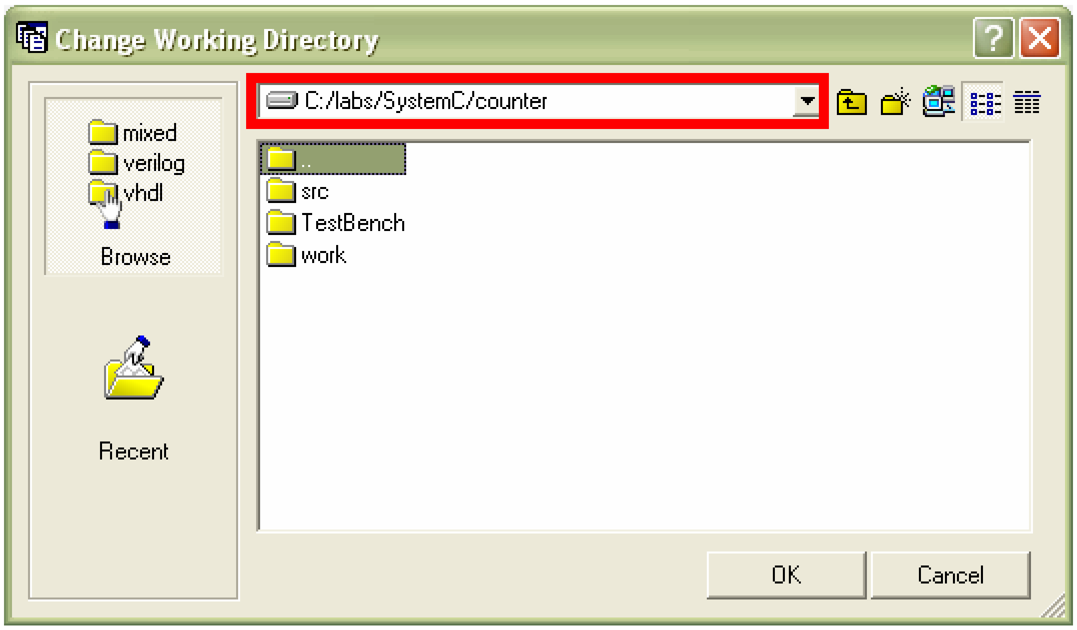
In this lab, we are going to use Aldec's Riviera HDL simulator in order to instantiate the up/down counter developed earlier. It has been modified, and its name changed

For your convenience the DLL has already been created using GCC. We are going to load it into the Riviera Interface and execute it.

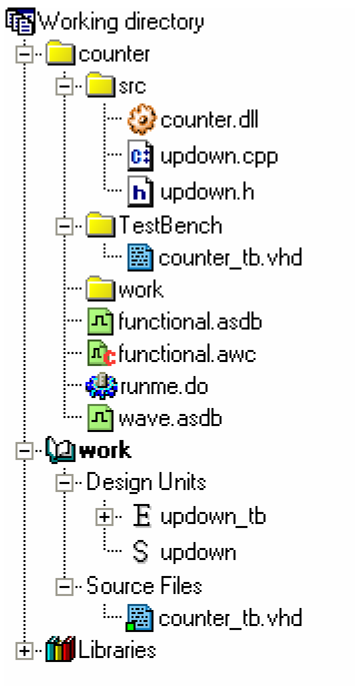
Start The Riviera software either from the desktop or the start menu.

Change the directory to **c:/labs/SystemC/counter**





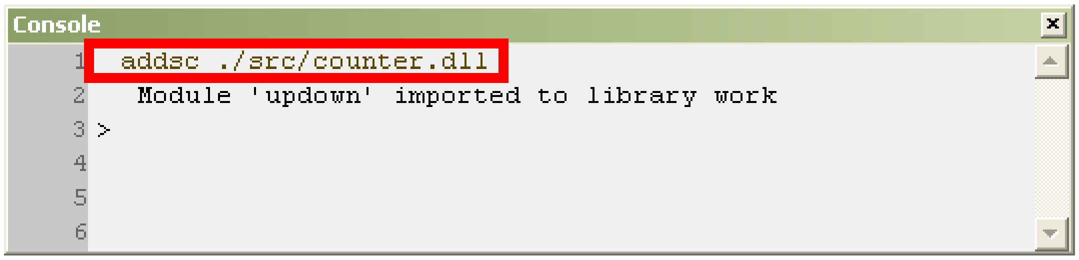
This directory contains the files necessary to execute the SystemC module. The src directory includes SystemC files. TestBench directory includes a VHDL testbench file.



You can open these files (double click on them) in order to view them.

In the console type in the following command

```
addsc ./src/counter.dll
```

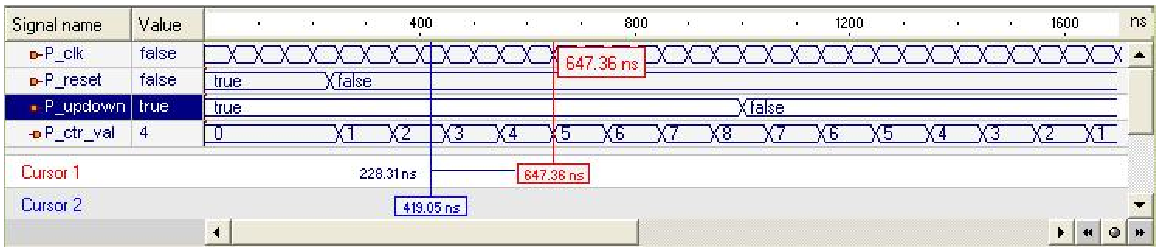

```
addsc ./src/counter.dll
 Module 'updown' imported to library work
>
```

This adds the SystemC module (updown) to the working library. From the standpoint of the HDL simulator, **updown** is now just a module in the library that can be instantiated and used in the design.

For your convenience, a script file called **runme.do** has been included in order to run the simulation.

At the console prompt type **do runme.do** and press **enter**.

The simulation should execute, and the waveform results display.


Signal name
Value
400
800
1200
1600
ns
P_clk
false
P_reset
false
true
false
P_updown
true
true
false
P_ctr_val
4
0 1 2 3 4 5 6 7 8 7 6 5 4 3 2 1
647.36 ns
Cursor 1
228.31 ns
647.36 ns
Cursor 2
419.05 ns