Introduction to SystemC

Overview

- Introduction
- SystemC
  - Transaction Level Modeling
- SystemC
  - Main language elements
- SystemC
  - Basic example
- More info…

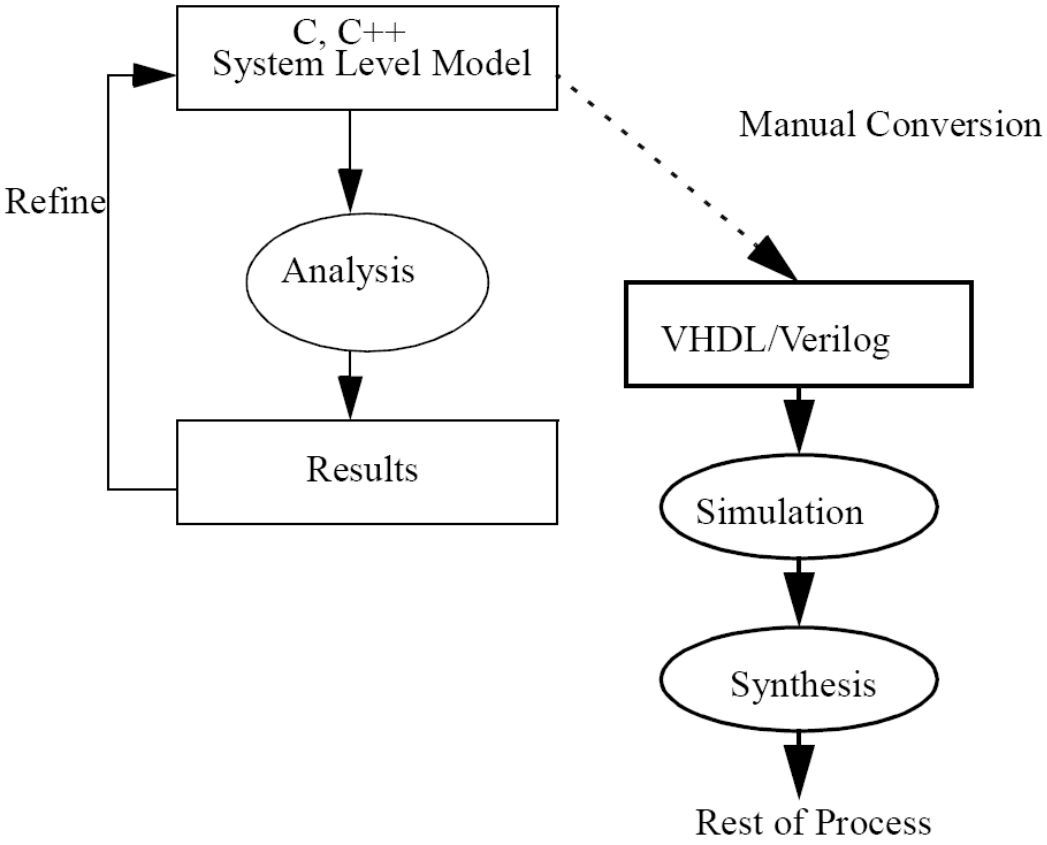
Introduction

Introduction

- The development of a digital electronic system starts from requirements collection, typically in natural language, and analysis
- The next step the creation of a system model by means of a modeling/specification language able to
  - Provide an higher level of abstraction to cope with very complex systems
  - Be technology independent but suitable for both HW and SW
  - Allow fast and reasonable accurate Design Space Exploration (DSE), early V&V (Simulation), and to reduce the design productivity gap

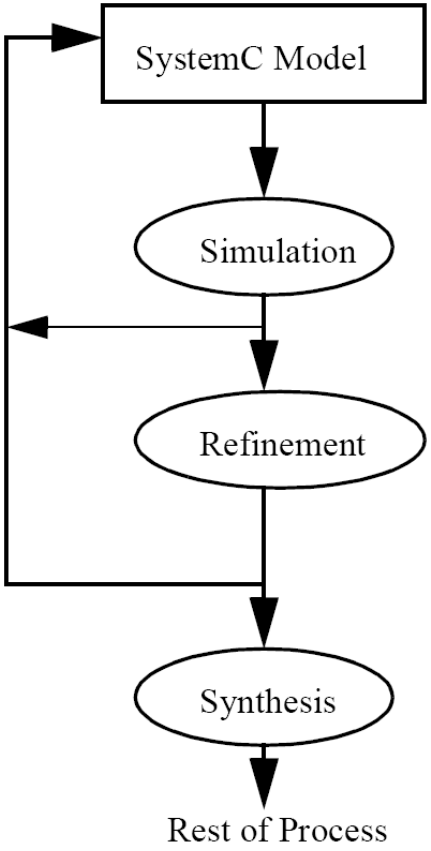
Introduction

- Classical system-level design flow



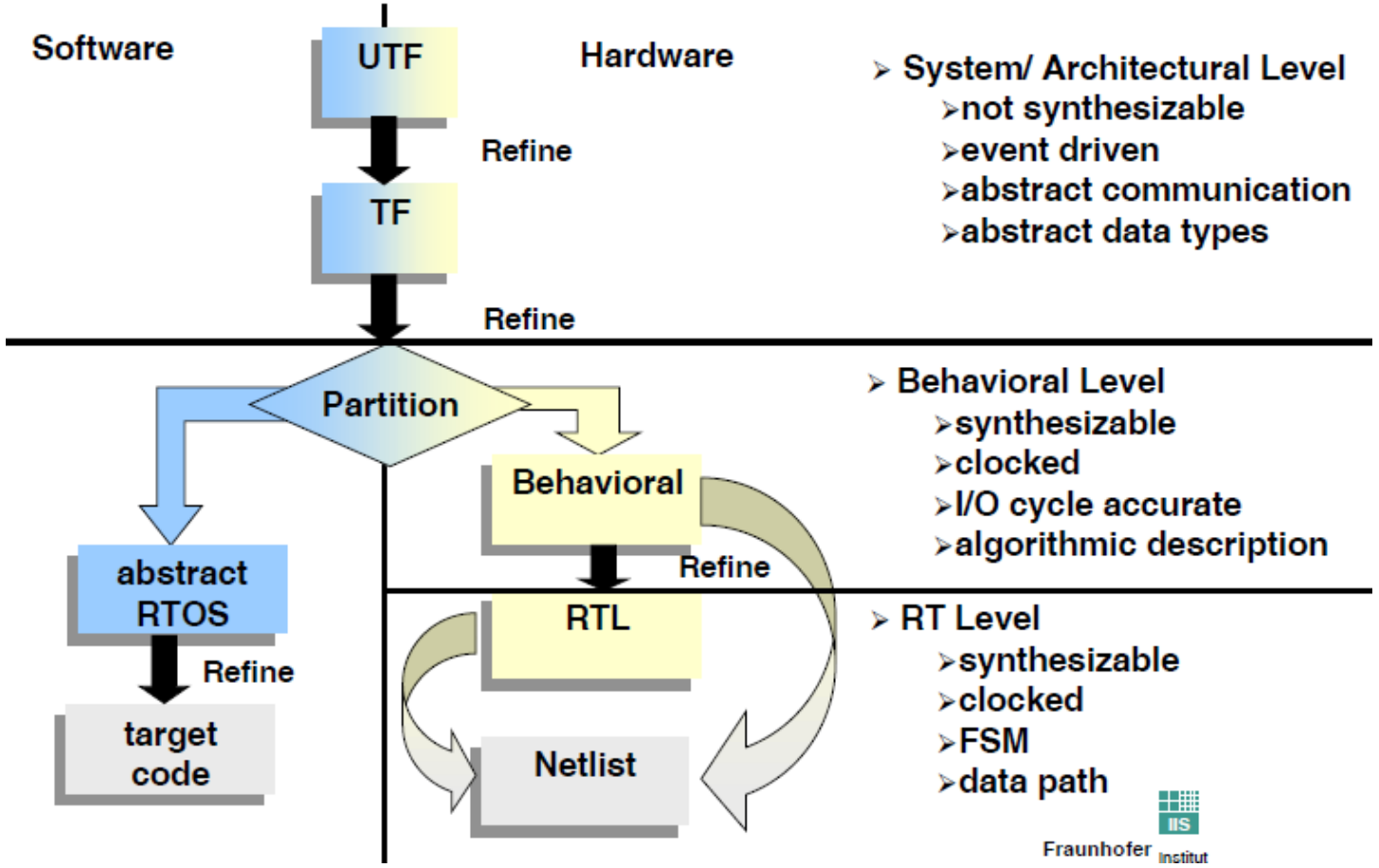
Introduction

- SystemC system-level design flow



Introduction

- SystemC system-level design flow



Introduction

- C++ is not suitable to describe complex HW/SW systems because it misses
  - Model of the time
  - Concurrency
  - HW data types (e.g. Z)
- But C++ is extensible thanks to the OOP power!

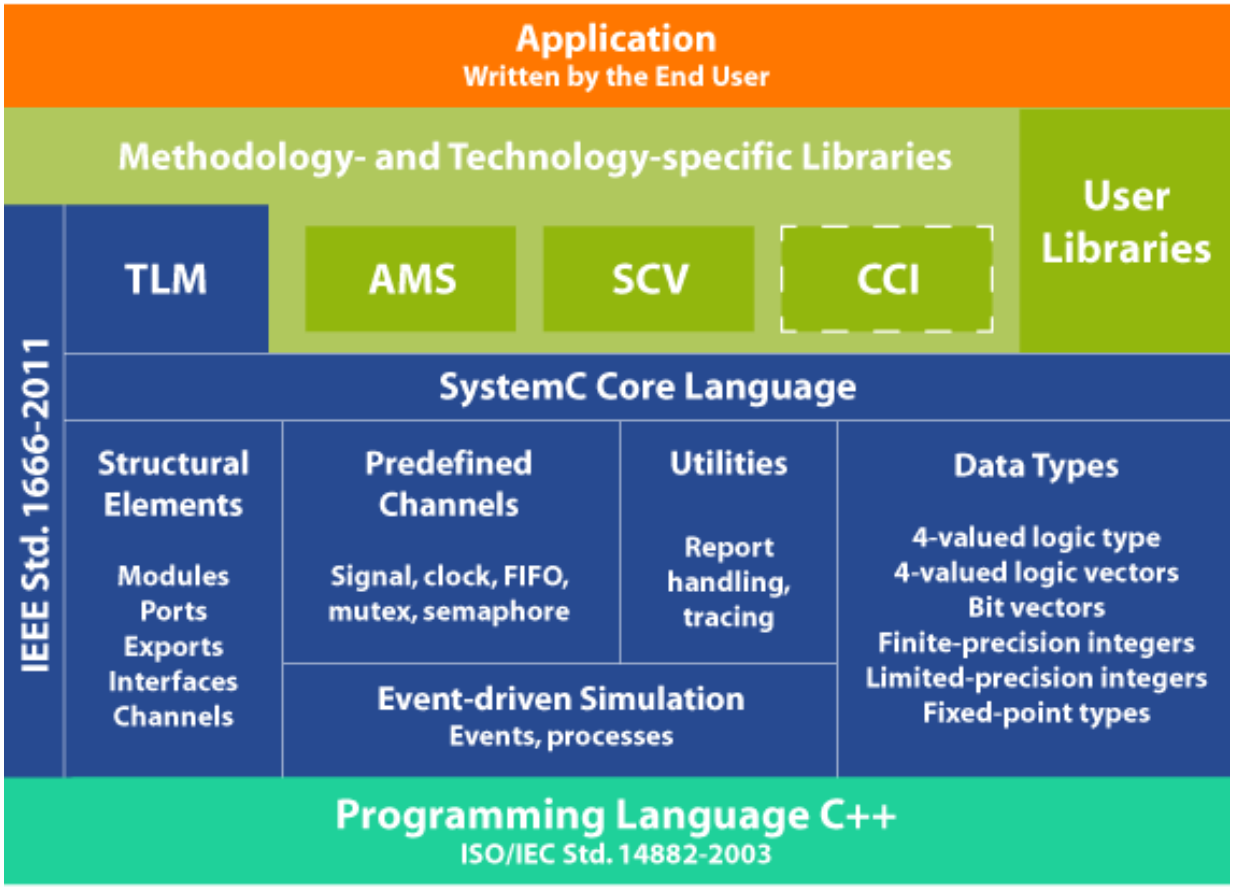
- In fact SystemC is a C++ library that allows to overcome the above limitations
  - Moreover, the model is directly executable (i.e. simulatable) since it is “just” a C++ program
    - The simulation Kernel is provided by the same library

Introduction

- SystemC library is available for free thanks to the OSCI (*Open SystemC Initiative*), an organization composed of the main players in the EDA domain
  - www.accellera.org (ex www.systemc.org)
- The library has evolved during the time
  - SystemC 1.x: RTL and *Behavioural* modeling (HW)
  - SystemC 2.x: System modeling (HW/SW)
    - Moreover, there are some specific extensions
      - TLM (*Transaction Level Modeling*)
      - AMS (Analog-Mixed Systems)
      - SCV (SystemC Verification Library)
      - CCI (Configuration, Control and Inspection)

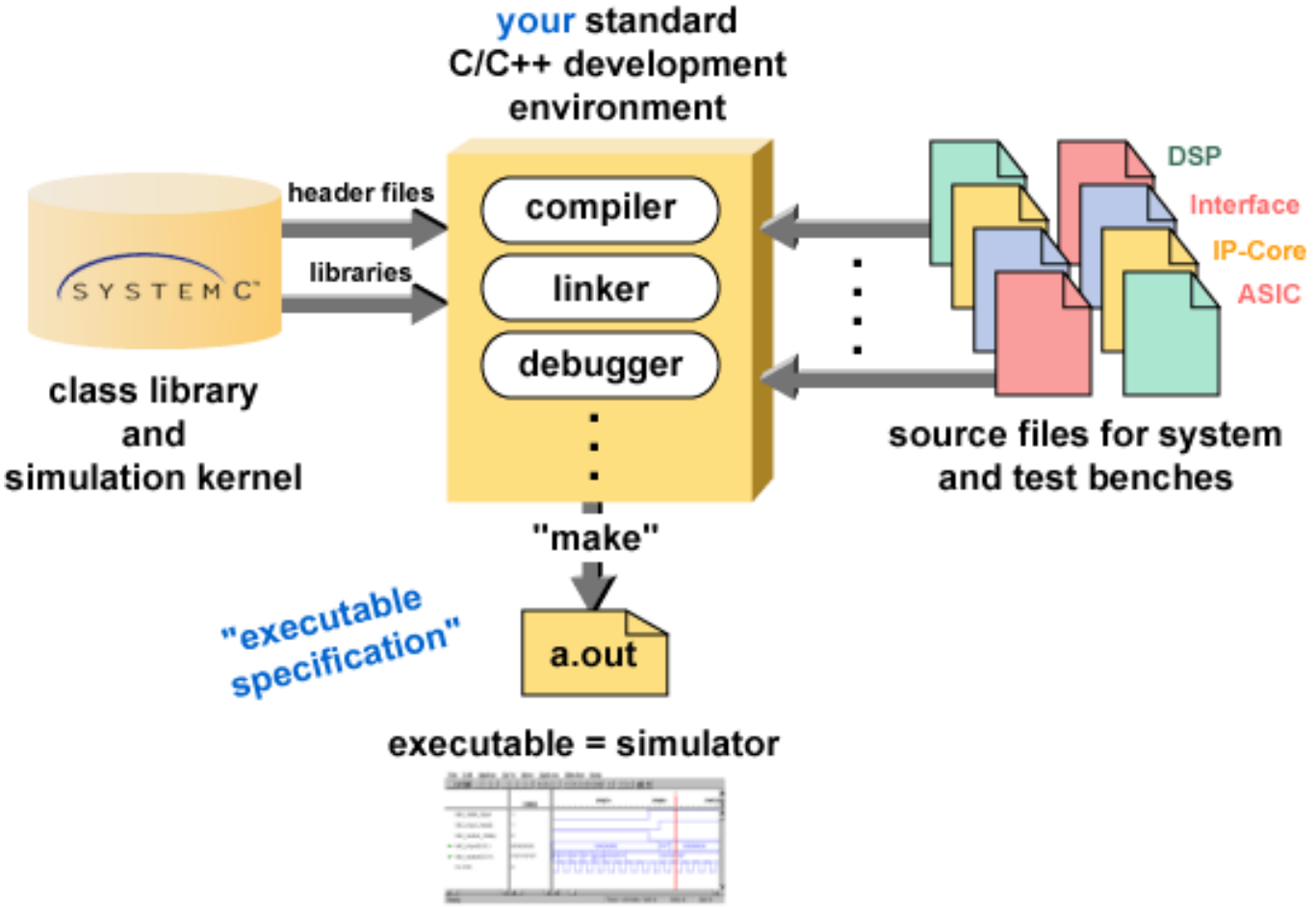
Introduction

- SystemC library architetcure (v2.3.1)



Introduction

- SystemC library usage

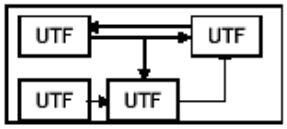


Introduction

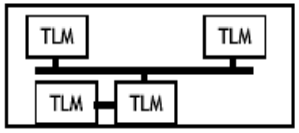
- The model is executable
  - Higher the abstraction level faster the simulation time
- Several abstraction levels can coexist in the same model
  - Gradual model refinement and continuos V&V
  - Unique simulation environment

Introduction

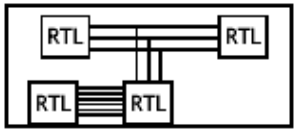
- *SystemC* 1.x
  - RTL and *Behavioural* modeling (HW)
- *SystemC* 2.x
  - System modeling (HW/SW)



- (U)TF: (UnTimed) Functional Level
  - No time accuracy for computation/communication



- TLM: Transaction Level Modeling
  - Abstract structural view for platform design and co-verification
    - » TLM API has been also standardized (TLM 2.0)



- RTL: Register Transfer Level
  - Time accuracy for computation/communication

SystemC

Transaction Level Model

SystemC

- Transaction Level Modeling
  - The goals are
    - To model a system by separating computation and communication modeling of its components
    - To provide a unique modeling language for description with different detail levels
    - To provide a unique modeling language for description of different model of computation
      - The underlying simulation Kernel is Discrete Event based

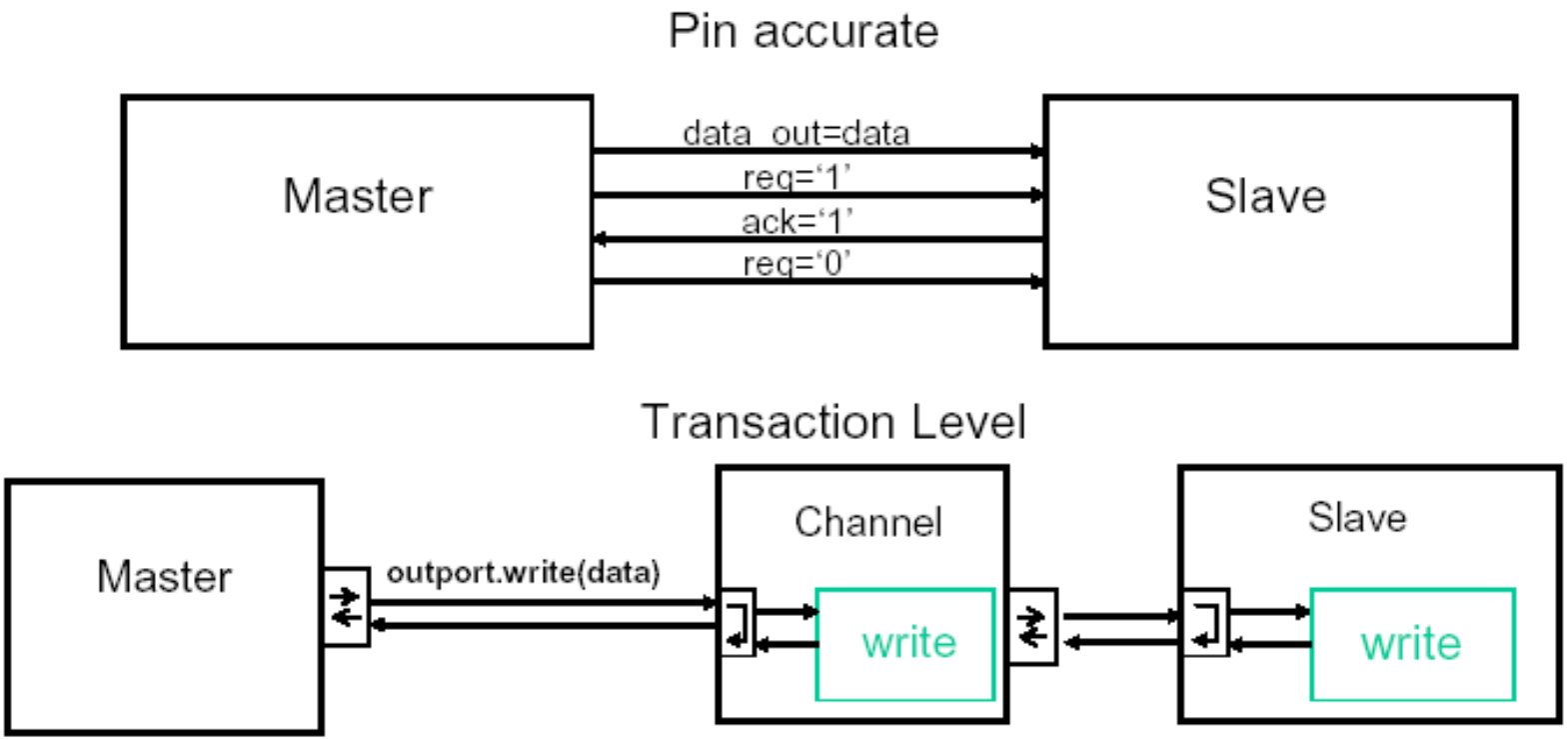
- Transaction Level Modeling
  - What is a *Transaction*?
    - It is a data exchange between two components of the model description
      - Examples
        - » The transfer of data between two stages of a pipeline
        - » The transfer of words between a processor and a memory
        - » The transfer of complex data structure between two phases of data processing
  - What vs. How
    - A transaction models only which data is transferred (type, value), not in which way the transfer is performed

SystemC

- Transaction Level Modeling
  - In the context of TLM, communications can be also modeled by introducing different time granularity
    - *Untimed functional*
      - Only the data exchange without concept of time
    - *Bus cycle accurate*
      - Data exchange with the concept of the time limited to the granularity needed to to perform an operation on the bus
    - *Cycle accurate*
      - Data exchange with the concept of the time equal to the clock cycle granularity

SystemC

- Transaction Level Modeling
  - Transaction Level vs. Pin Level (RTL)



SystemC

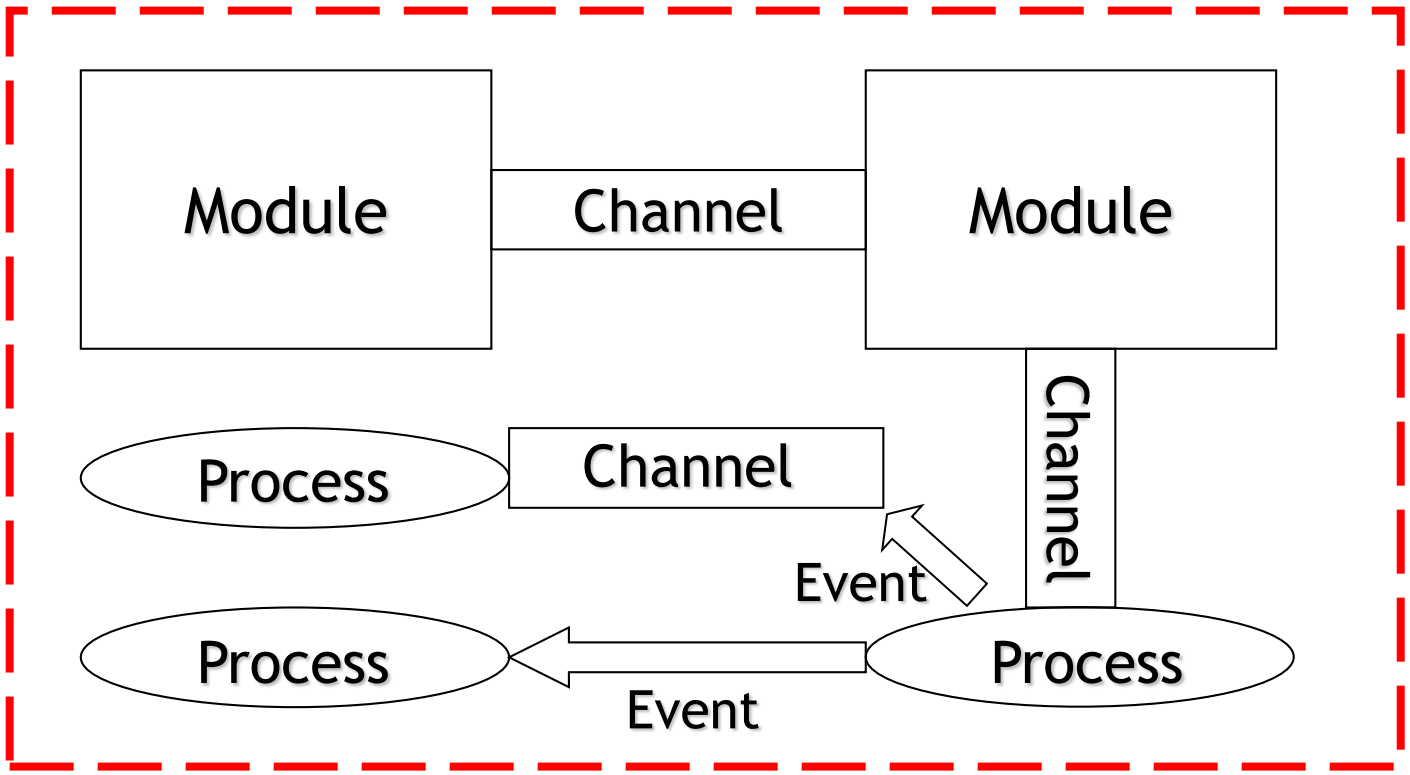
Main language elements

SystemC

- Main language elements
  - Overview
  - Data Types
  - Interfaces and Channels
  - Modules
  - Processes
  - Hierarchy
  - Examples

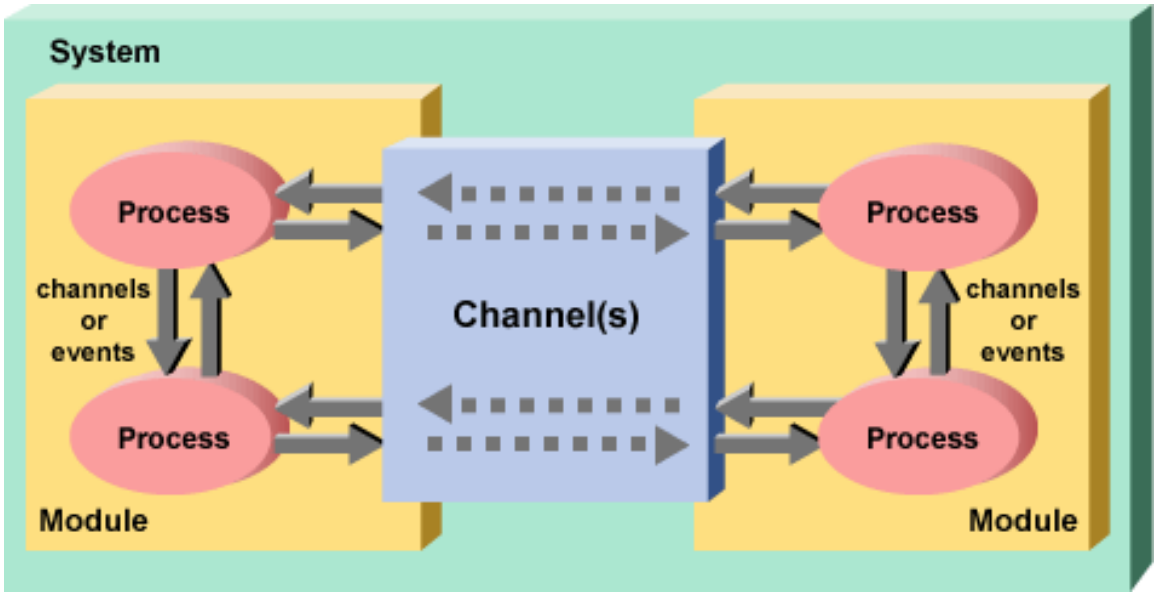
Main language elements

- Overview



Main language elements

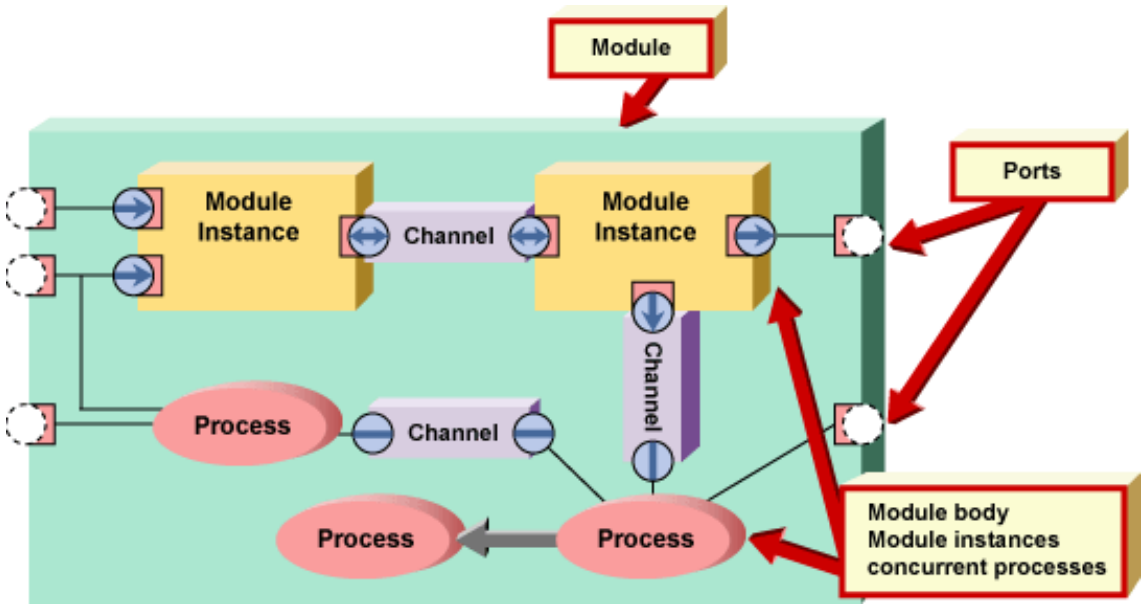
- Structure of a SystemC model



- The modeled system is composed of **modules**
  - Module behaviour is described by **processes** that communicates by **events** and/or **channels**
    - Inter-module communications is performed by channels

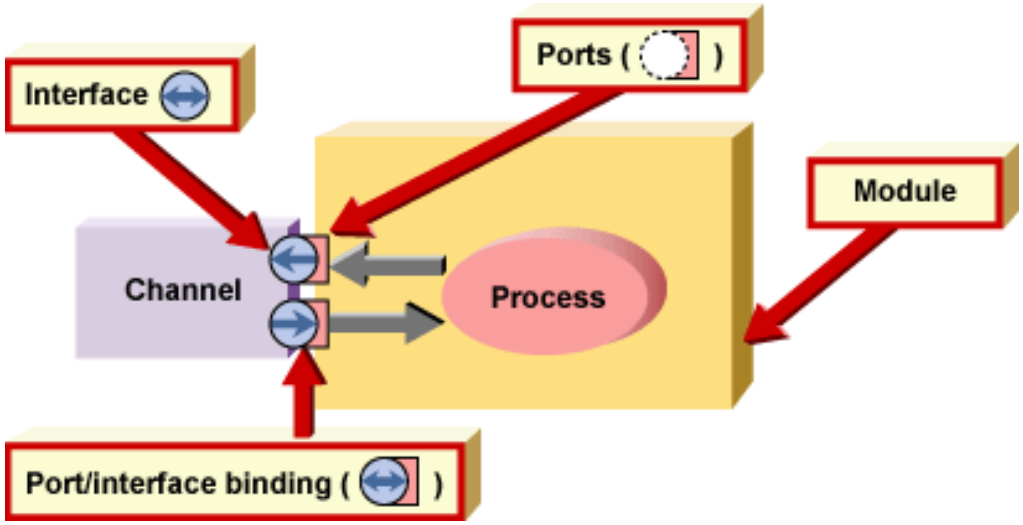
Main language elements

- Module instances at the same hierarchical level are connected by means of **ports** to channels
  - Processes are directly connected by channels or event
- Modules support hierarchy
  - Connections are performed by ports



Main language elements

- An **interface** describes the set of methods accessible by ports
  - Interfaces are implemented by channels
  - Interface are connected to ports
- Channels can be developed independently by modules (they have to guarantee the interface)

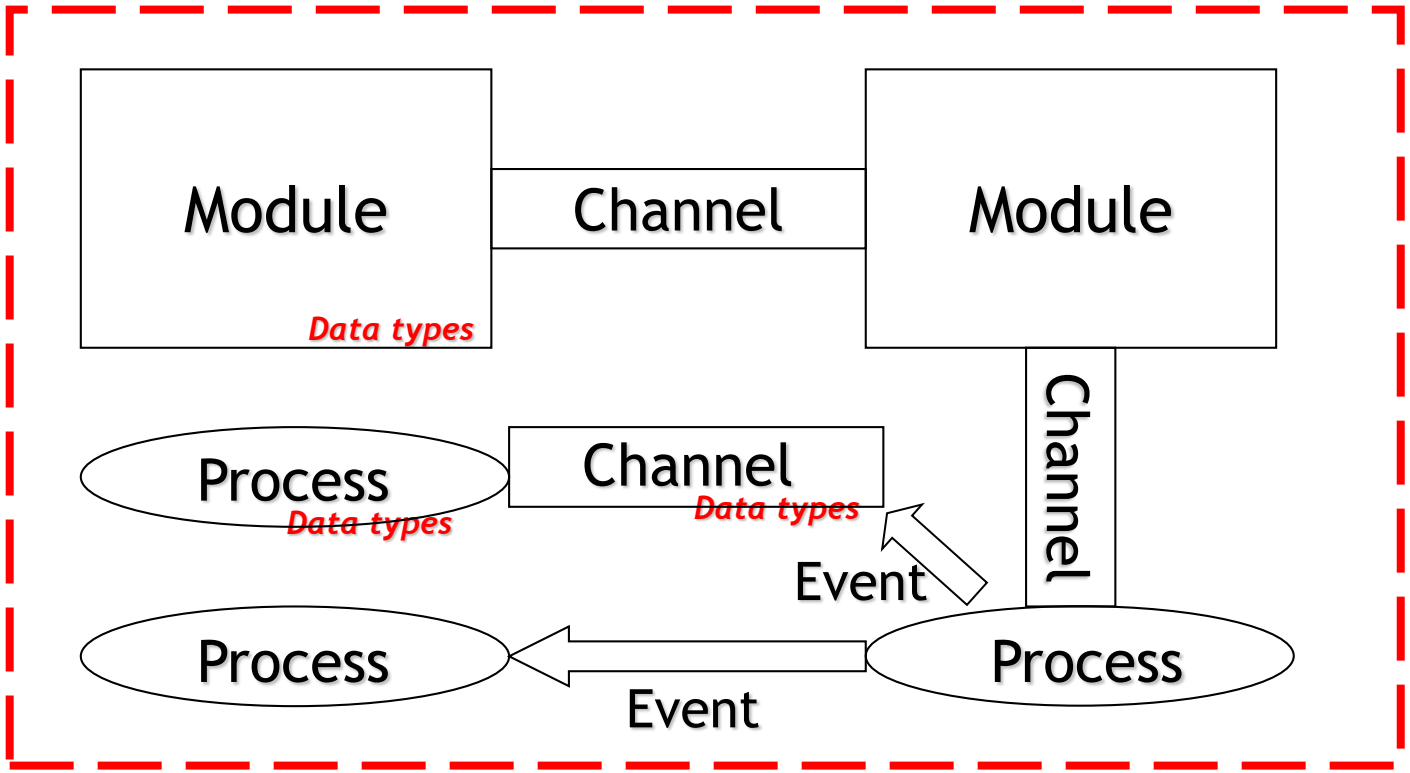


Main language elements

- **Events** (*sc_event*) are the basic synchronization element
  - Processes are executed and their output updated depending on events
    - SystemC library provides a scheduler that manages process execution
- A process is activated by the events of its **sensitivity list**
  - Sensitivity list can be
    - Static: defined before the simulation start
    - Dynamic: defined at runtime
  - A process can wait for an event
    - *wait()*

Main language elements

- Data Types



Main language elements

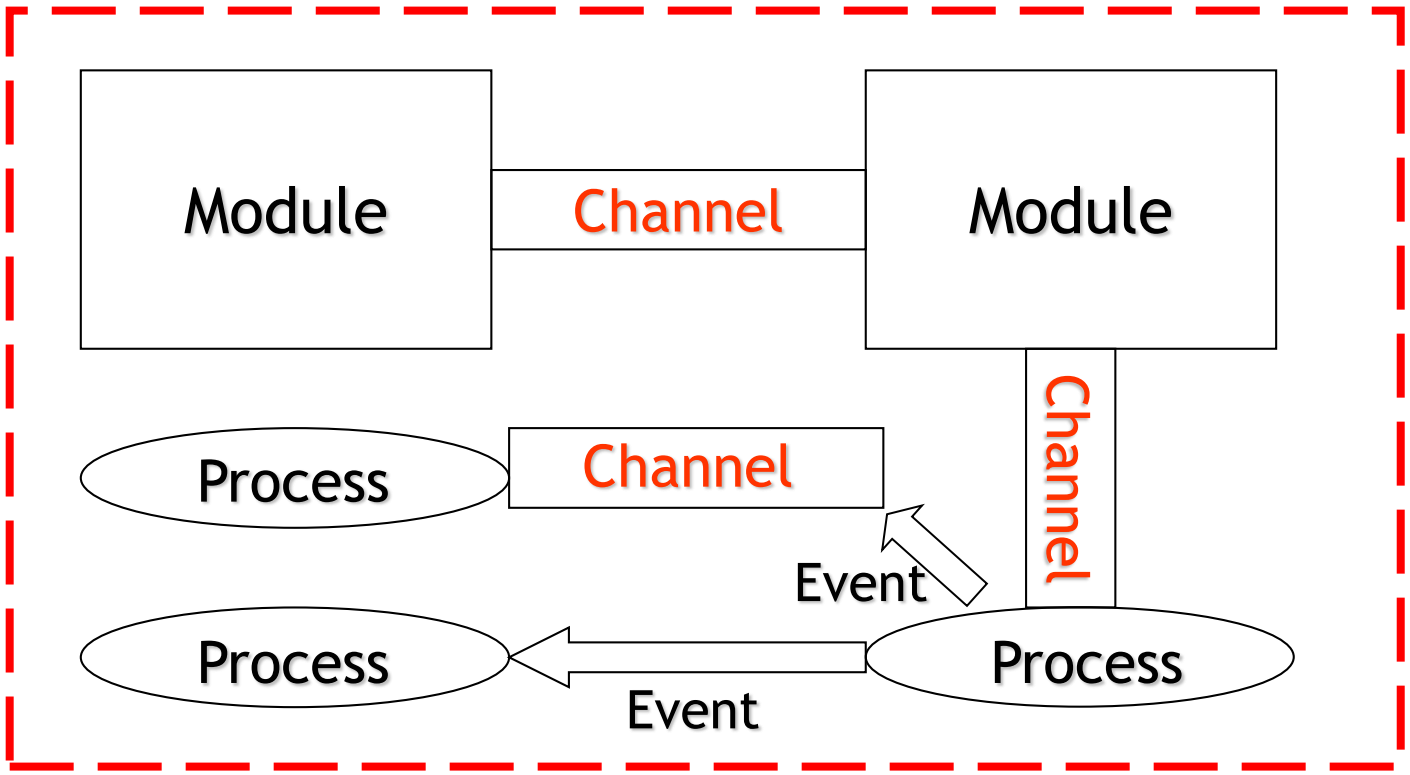
- Data Types
  - SystemC allow the use of both native C++ data types and library defined ones
    - SystemC data types allow a more detailed description but lead to less performant simulations
  - SystemC numeric data types
    - Fixed precision
    - Arbitrary precision
    - 4-values logic
    - Array of 4-values logic
    - Fixed point

Main language elements

- Data Types
  - SystemC numeric data types
    - *sc_uint*<>, *sc_int*<>: signed and unsigned integers
      - User-defined length (max 64 bit)
    - *sc_biguint*<>, *sc_bigint*<>
      - Arbitray length signed and unsigned integers
    - *sc_logic*
      - Used to represent a single bit with 4-values logic
        - (‘0’, ‘1’, ‘Z’, ‘X’)
    - *sc_lv*<>: array of *sc_logic* bit
    - *sc_fixed*<>, *sc_ufixed*<>: fixed point
      - Used to refine floating point data types

Main language elements

- Interfaces and Channels



Main language elements

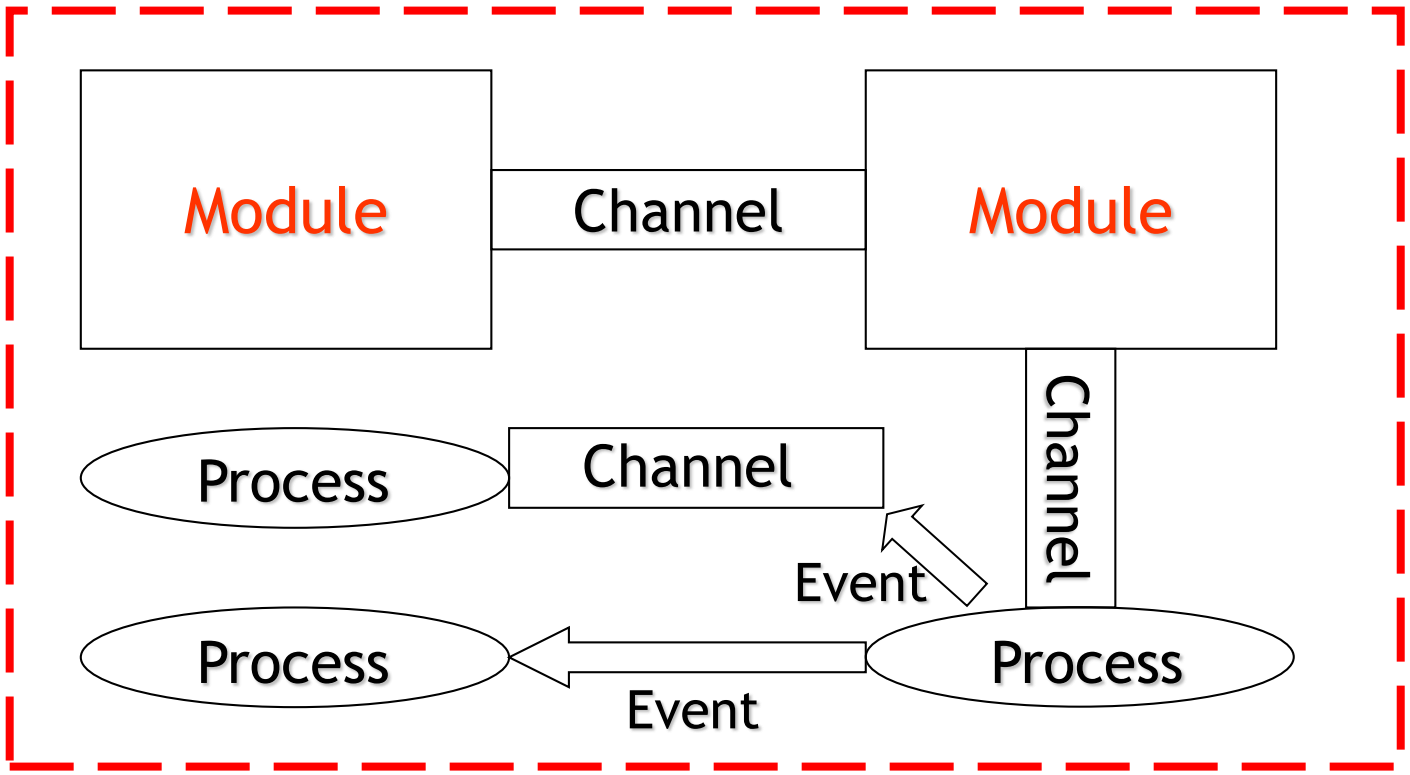
- Interfaces
  - Interfaces define a set of access methods
    - Their implementation is defined by channels
  - SystemC provides a set of predefined interfaces
    - sc_signal_in_if, sc_signal_out_if, sc_signal_inout_if, sc_fifo_in_if, sc_fifo_out_if, sc_mutex_if, sc_semaphore_if
      - Each ones defines some virtualmethods
        - » e.g. write() is defined only in out interfaces
  - If a channel implements an interface it has to implement all the related methods

Main language elements

- Channels
  - Primitive channels (predefined)
    - They don’t have a visible internal structure and cannot access to other primitive channels
      - sc_signal, sc_signal_rv
      - sc_fifo, sc_mutex
      - sc_semaphore, sc_buffer
  - Hierarchical channels
    - They are like modules and can contain processes and other channels
      - Hierarchical channels lead to less performant simulations

Main language elements

- Modules



Main language elements

- Modules
  - They are containers defined by an interface (.h) and a functionality (.cpp)
  - A module can be composed of
    - Ports
    - Internal channels
    - Internal variables
    - Processes
    - Standard C++ methods
    - Constructors
    - Instances of other modules

Main language elements

- Modules
  - A module is declared by means of the SC_MODULE macro

```
SC_MODULE ( module_name) {
   // body of module
};
```

It is only a way to hide inheritance from *sc_module class:*

class module_name::sc_module…

- Access to a module is perfomed by means of ports (*sc_port* class) that are linked to an interface
  - Three port types
    - » In, Out, Inout
  - Complete syntax: *sc_port<interface_type, N> port_name;*
    - » N: number of channels that can be connected to the port

Main language elements

- Modules
  - A module can declare internal channels used to
    - Connect modules at the same hierarchical level
    - Connect internal processes
    - Connect a process of a module to the port of an internal module
  - A module can declare internal variales
  - A module can declare internal methods and processes
    - Processes are particular methods that are “registered” in the consctructor
      - This allow the scheduler to properly manage them

Main language elements

- Modules
  - A module has a constructor used to create its instance with the internal data structures
    - Inits internal variables and signals
    - Defined the processes

```
SC_CTOR(my_module) {
  SC_METHOD(internal_process); //Method Process
  sensitive << in_p;
  internal_variable= 1;
}
```

Main language elements

- Modules
  - Channels *sc_signal*, *sc_signal_rv*, *sc_fifo* can exploit special ports to simplify declarations
    - *sc_signal*
      - *sc_in<T>*, *sc_out<T>*, *sc_inout<T>*
    - *sc_signal_rv*
      - *sc_in_rv<T>*, *sc_out_rv<T>*, *sc_inout_rv<T>*
    - *sc_fifo*
      - *sc_fifo_in<int>*, *sc_fifo_out<T>*, *sc_fifo_inout<T>*

Main language elements

- Modules
  - To read a value from a port (or internal channel) it is possible to use the *read()* method or the = operator
  - To write a value to a port (or internal channel) it is possible to use the *write()* method or the = operator
    - The methods are declared in the .cpp (funcionality) and acts on a port declared on the .h (interface)
      - Interface
        - sc_signal<int> data;
        - sc_signal<bool> condition;
        - int a;
      - Functionality
        - a=data.read();
        - condition.write(a);

Main language elements

- Modules
  - Example: interface

```
SC_MODULE(encode){
  sc_in< bool > clock; //Ports
  sc_in< bool > reset;
  sc_in< bool > input;
  sc_out< sc_bv<3> > output;
  sc_lv<8> trellis; //Variables
  sc_lv<3> tmp;
  sc_lv<8> input1;
  void codeGen(); //Function/method protoype
  SC_CTOR(encode) // Constructor
  {
   SC_CTHREAD(codeGen,clock.pos());
   watching(reset.delayed() == true);}
};
```

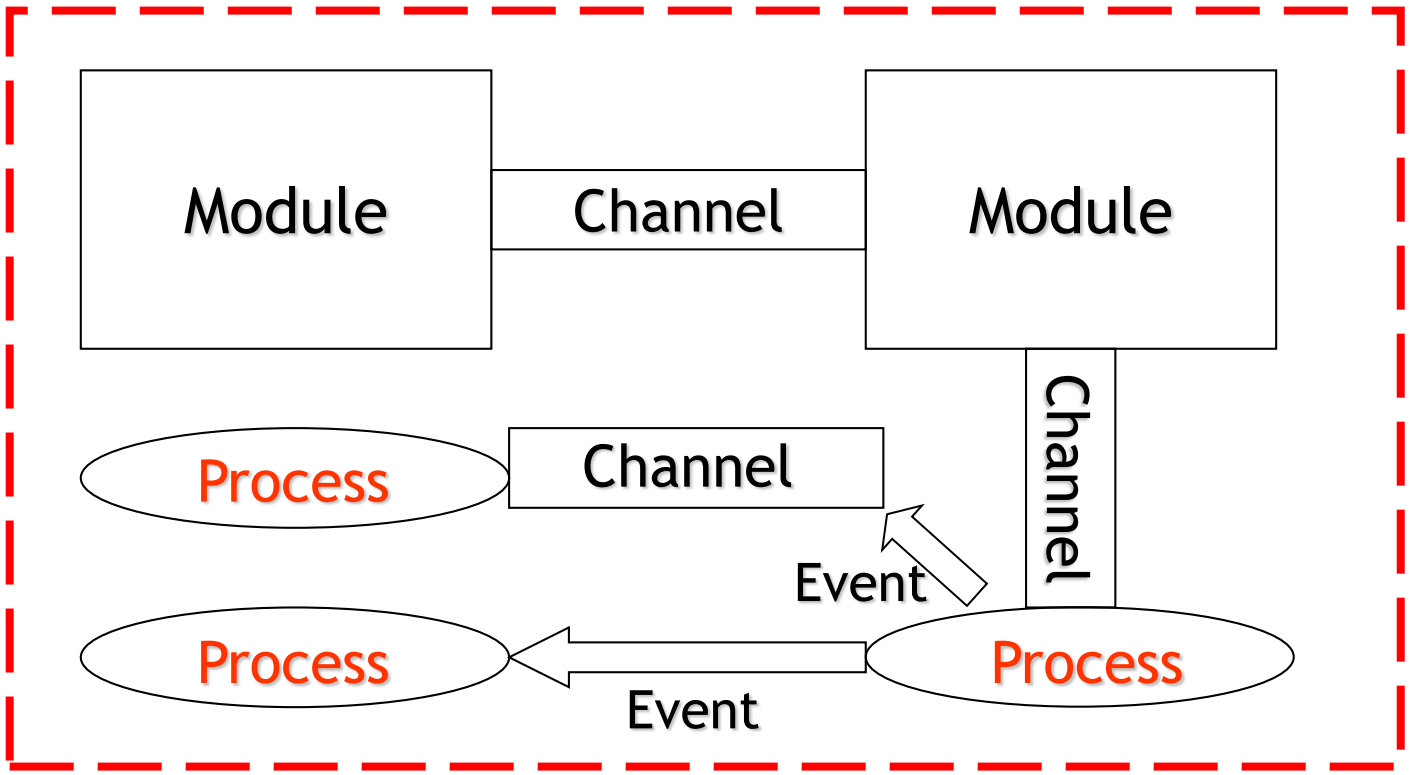
Main language elements

- Modules
  - Example: functionality

```
#include "encode.h"
void encode::codeGen(){
 trell=0x00; //Init
 wait();
 //Neverending loop…
 while(true){
  input1[0]=input.read();
  trell=((trell)<<1)|input1;
  tmp[2]=trell[7]^trell[4]^trell[2]^trell[0];
  output.write(tmp); //Scrittura uscita
  wait();}
}
```

Main language elements

- Processes



Main language elements

- Processes
  - Events are the basic elements for processes synchronization
    - *sc_event event_name;*
  - An event can be notified
    - Immediately: processes sensitive to the event became ready in the current *delta-cycle*
      - See later for more info about delta-cycle
    - Delayed: processes sensitive to the event became ready in the next *delta-cycle*
    - After a specified time

Main language elements

- Processes
  - System functionalitites (i.e. the system behaviour) are described in the processes
    - Processes exploit events or channels to comunicate
      - They have to be registered in the module constructors
  - There are three process types
    - SC_METHOD; SC_THREAD; SC_CTHREAD
      - SC_METHOD behaves as functions
        - » After a call they return to the caller
      - SC_THREAD and SC_CTHREAD behave like threads
        - » Thay are called only one time and then they have their execution flow

Main language elements

- Processes
  - SC_METHOD (asynchronous function)
    - Sensible to a set of signals/events
      - Sensitivity list
        - » e.g.
        - » *sensitive(signal1), sensitive<<s1<<s2<<s3*
        - » *sentitive_pos<<clk, sensitive_neg<<clk*
    - Each time it is invoked, the whole function statements are sequentially executed until the end with 0 simulation time
      - No *wait()* allowed inside the function
    - Mainly used for RTL modeling

Main language elements

- Processes
  - SC_THREAD (asynchronous thread)
    - Sensible to a set of signals/events
      - Sensitivity list
    - Each time it is invoked, the function statements before the first *wait()* are sequentially executed in 0 simulation time
      - At the following activation, the execution will restart from the previous *wait()* and so on…
        - » Variables keep their value
    - Used to model both synchronous and asynchronous behaviors

Main language elements

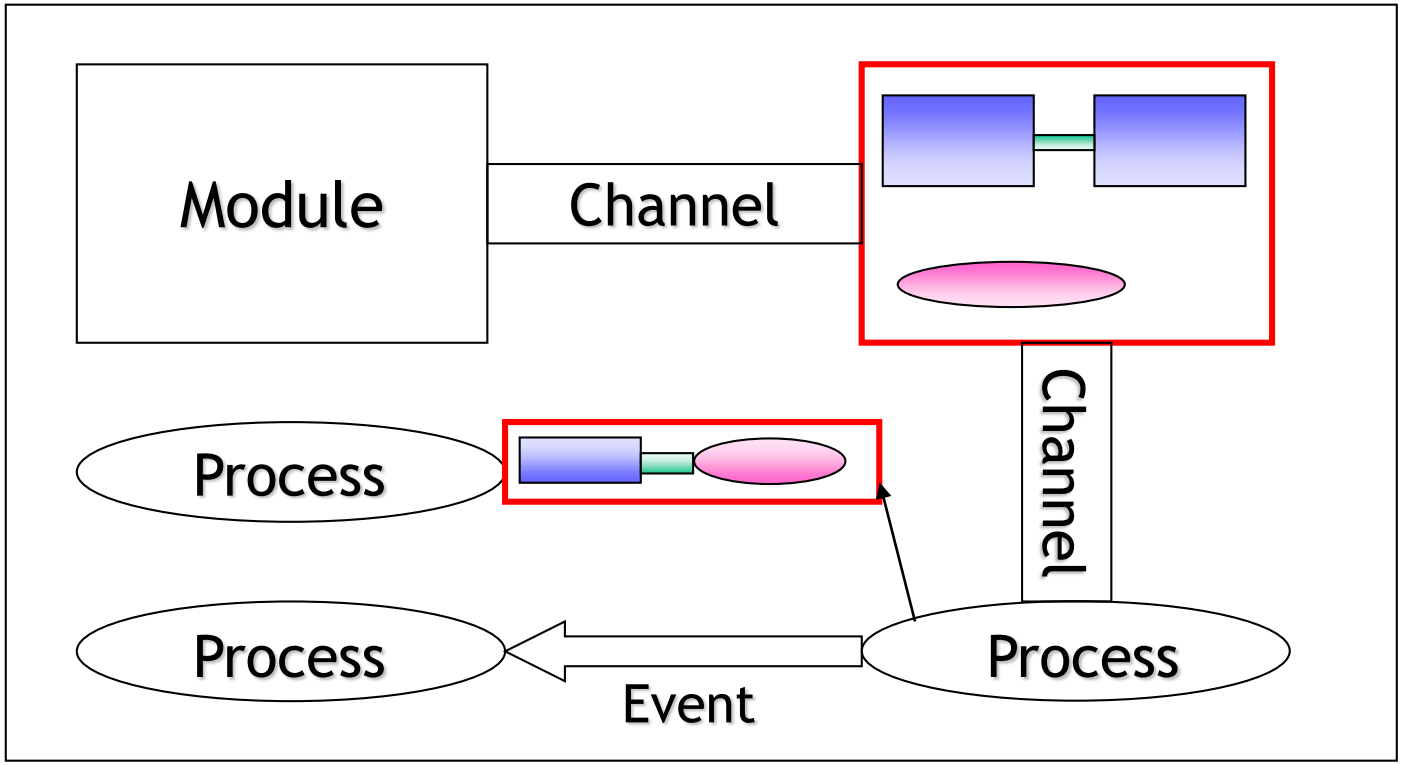
- Processes
  - SC_CTHREAD ("clocked" synchronous thread)
    - It is sensitive only to one front of one clock
    - Each time it is invoked, the function statements before the first *wait()* are sequentially executed in 0 simulation time
      - At the following activation, the execution will restart from the previous *wait()* and so on…
        - » Variables keep their value
    - Used to model synchronous behaviors
      - WARNING: Deprecated!

Main language elements

- Processes
  - WAIT() function
    - Used only in SC_THREAD and SC_CTHREAD
    - Suspend process execution until the next reactivation
      - SystemC 1.x
        - *wait()*
        - *wait(<var_int>)* – wait for a number of cycles
      - SystemC 2.x
        - *wait(event)*
        - *wait(e1 |e2 |e3)* – wait for one of the events
        - *wait(e1 & e2 & e3)* – wait for all the events
        - *wait(100, SC_NS, e1 | e2)* – wait with time-out

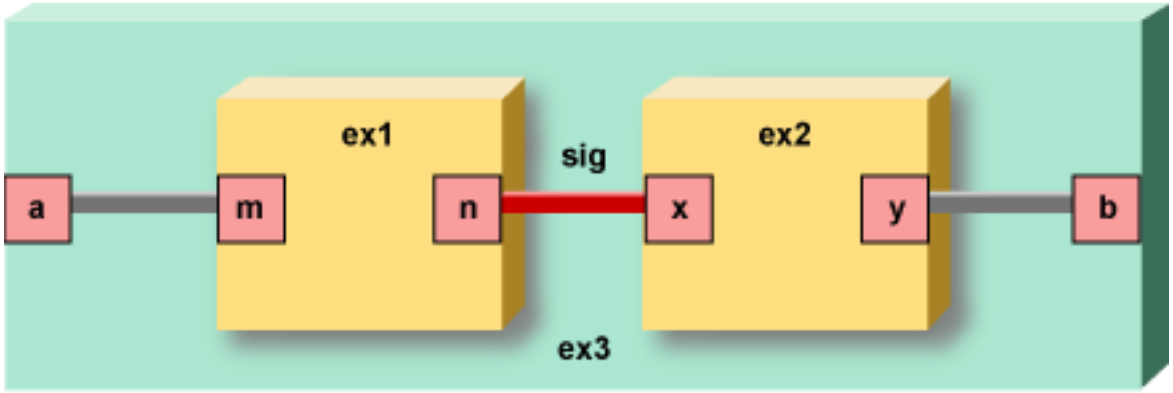
Main language elements

- Hierarchy



Main language elements

- Hierarchy
  - By means of modules it is possible to build a hierarchical structure
    - As in VHDL it is possible to connect father-child ports
      - There is the need for channels to connect modules at the same hierarchical level



Main language elements
Hierarchy

```
SC_MODULE(ex3){
   sc_port<sc_fifo_in_if<int> >a;
   sc_port<sc_fifo_out_if<int> > b;
   sc_fifo<int> sig;
   // Instances of ex1 and ex2
   ex1 ex1_instance;
   ex2 ex2_instance;
   // Module Constructor
   SC_CTOR(ex3):
   ex1_instance("ex1_instance"),
   ex2_instance("ex2_instance")
   {
        // Named connection for ex1
        ex1_instance.m(a);
        ex1_instance.n(sig);
        // Positional connection for ex2
        ex2_instance(sig, b);
   }
};
```

To create a hierarchy:

- Create the instance of the module
- Init the instance of the module
- Make the port binding

Binding can be:

- Named
- Positional

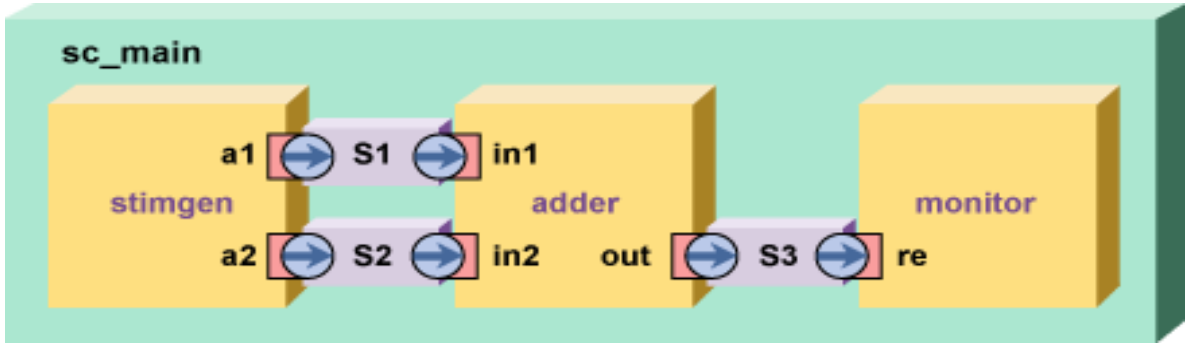
Main language elements

- Hierarchy
  - The *top level* is represented by the special function *sc_main()* that is called by the simulation kernel at the beginning of the simulation
    - Modules can be instantiated in the same way
  - At a given point of main, *sc_start()* is used to start the simulation
    - *sc_start()*
      - Run until there are events
    - sc_start(arg)
      - Run for a specified simulation time

Main language elements

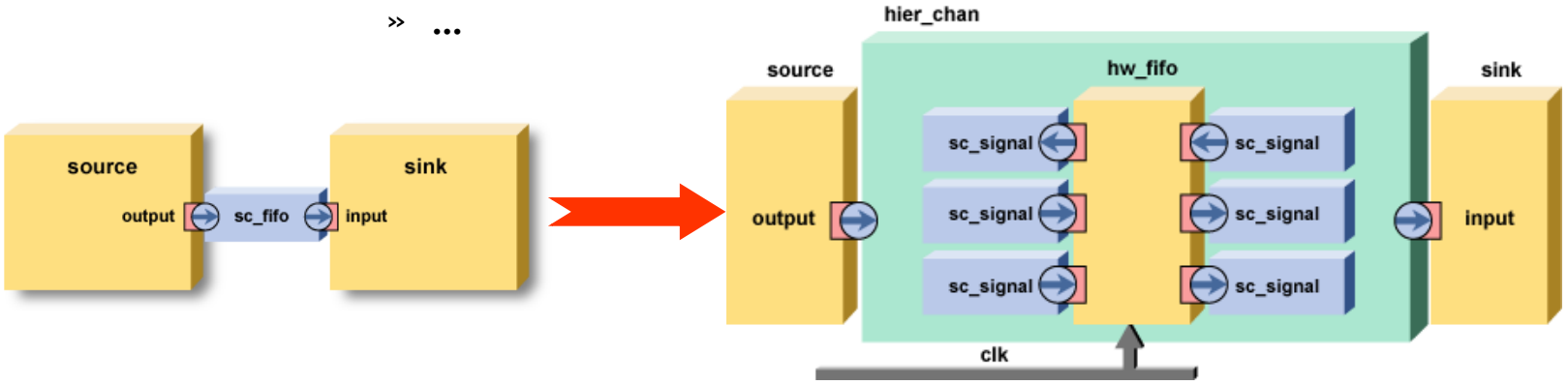
- Hierarchy

```
#include "systemc.h"
#include "adder.h"
#include "stimgen.h"
#include "monitor.h"
int sc_main(int argc, char *argv[ ])
{
  // Create fifos with a depth of 10
  sc_fifo<int> s1(10), s2(10), s3(10);

  // Module instantiations
  stimgen stim("stim");
  stim(s1, s2);
  // Adder
  adder add("add");
  add(s1, s2, s3);
  // Response Monitor
  monitor mon ("mon");
  mon.re(s3);
  …
  sc_start();
}
```



Main language elements

- Hierarchy: *Hierarchical Channels*
  - Hierarchical Channels are like modules so they can contain processes, other modules, etc.
    - They are very powerful but also complex so they can affect simulation performance
      - e.g.
        - » to replace a FIFO with a hierarchical channel that acts like a wrapper for a RTL FIFO description
        - » to describe the behaviour of a shared bus with arbiter
        - » …



Main language elements

- VHDL vs SystemC
  - Very similar for RTL modeling
    - VHDL not suitable for system-level
    - SystemC could be less efficient for synthesis

	VHDL	SystemC
Hierarchy	Entity	Module
Communication	Signal	Signal, Channel
Functionality	Process	Process
TestBench		Object orientation
System Level		Channel, interface, event, abstract data types,…
I/O	Simple file I/O	C++ I/O capabilities

Main language elements

- VHDL vs SystemC: RTL D FF

```
library ieee;
use ieee.std_logic_1164.all;
entity dffa is
  port( clock : in std_logic;
        reset : in std_logic;
        din   : in std_logic;
        dout  : out std_logic);
end dffa;

architecture rtl of dffa is
begin
  process(reset, clock)
  begin
    if reset = '1' then
      dout <= '0';
    elsif clock'event and clock = '1' then
      dout <= din;
    end if;
  end process;
end rtl;
```

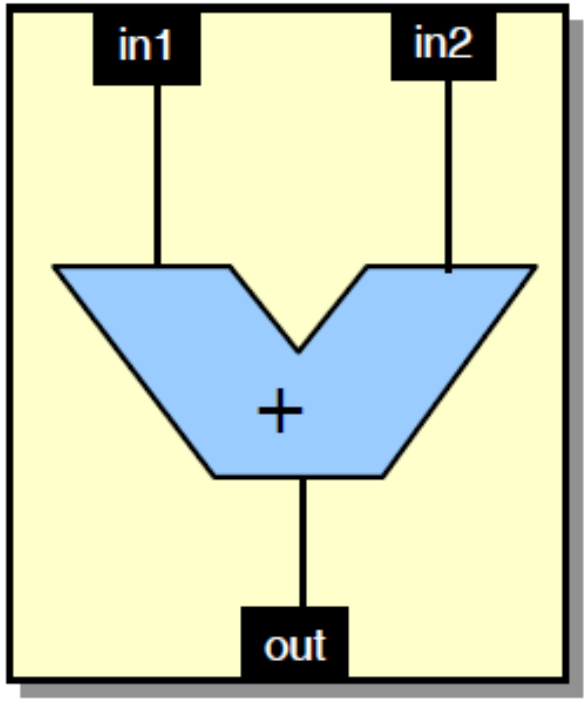
```
// dffa.h

#include "systemc.h"

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

  void do_ffa()
  {
    if (reset) {
      dout = false;
    } else if (clock.event()) {
      dout = din;
    }
  };

  SC_CTOR(dffa)
  {
    SC_METHOD(do_ffa);
    sensitive(reset);
    sensitive_pos(clock);
  }
};
```

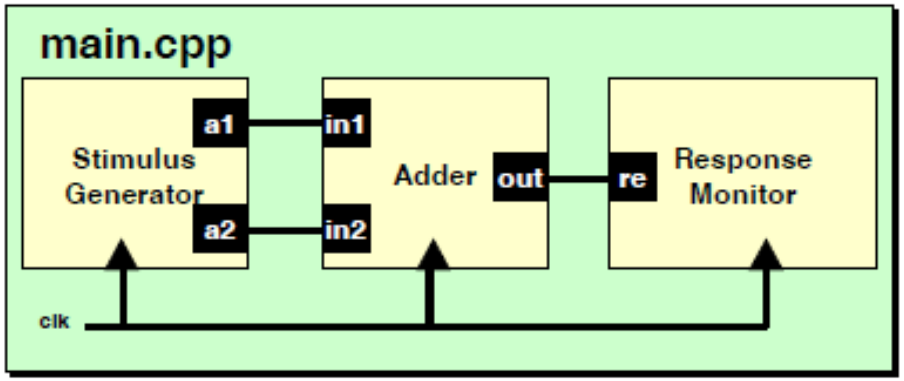
SystemC

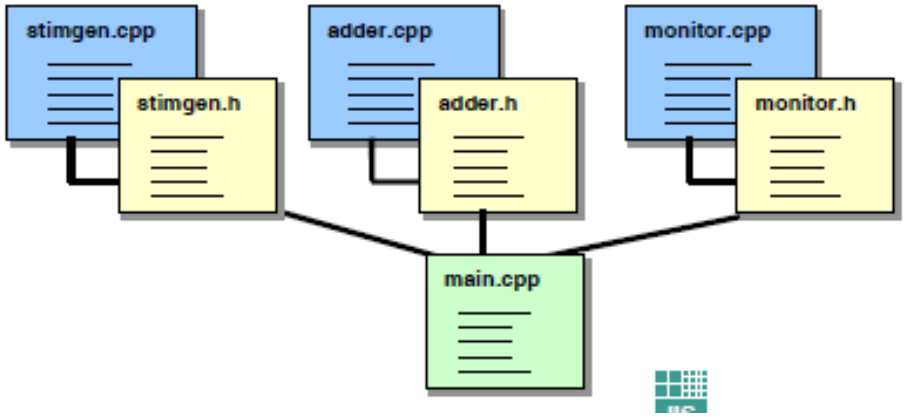
Basic Example

Basic example

- Overview



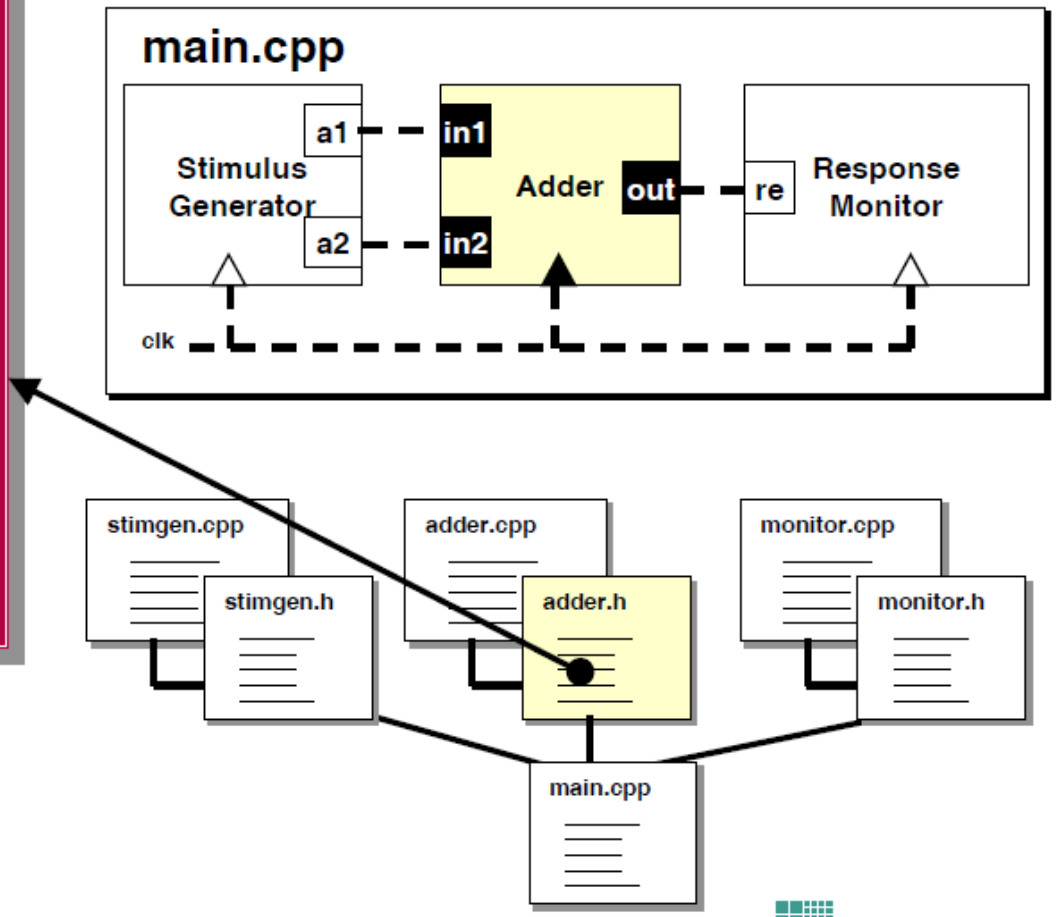




Basic example

- Simulation model

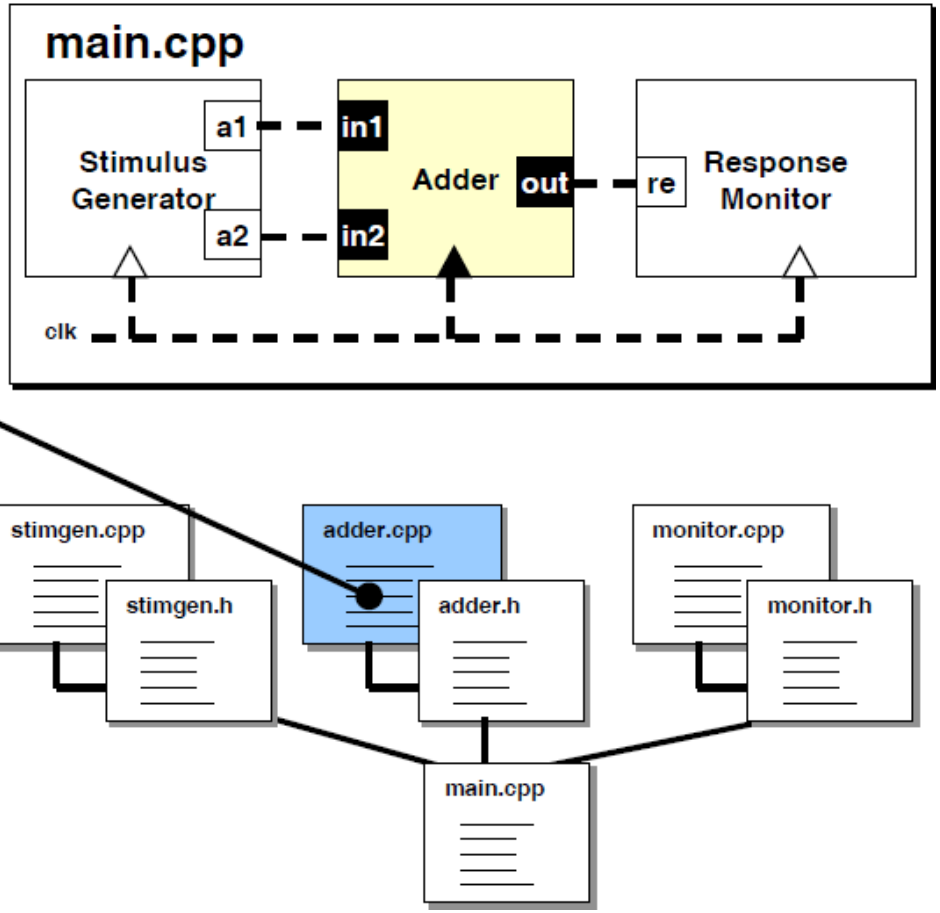
```
// header file adder.h
typedef int T_ADD;

SC_MODULE(adder) {
   // Input ports
   sc_port<sc_fifo_in_if<T_ADD> >  in1;
   sc_port<sc_fifo_in_if<T_ADD> >  in2;

   // Output ports
   sc_port<sc_fifo_out_if<T_ADD> > in2;

   // Constructor
   SC_CTOR(adder){
     SC_THREAD(main);
   }

   // Functionality of the process
   void main();
};
```



Basic example

- Simulation model

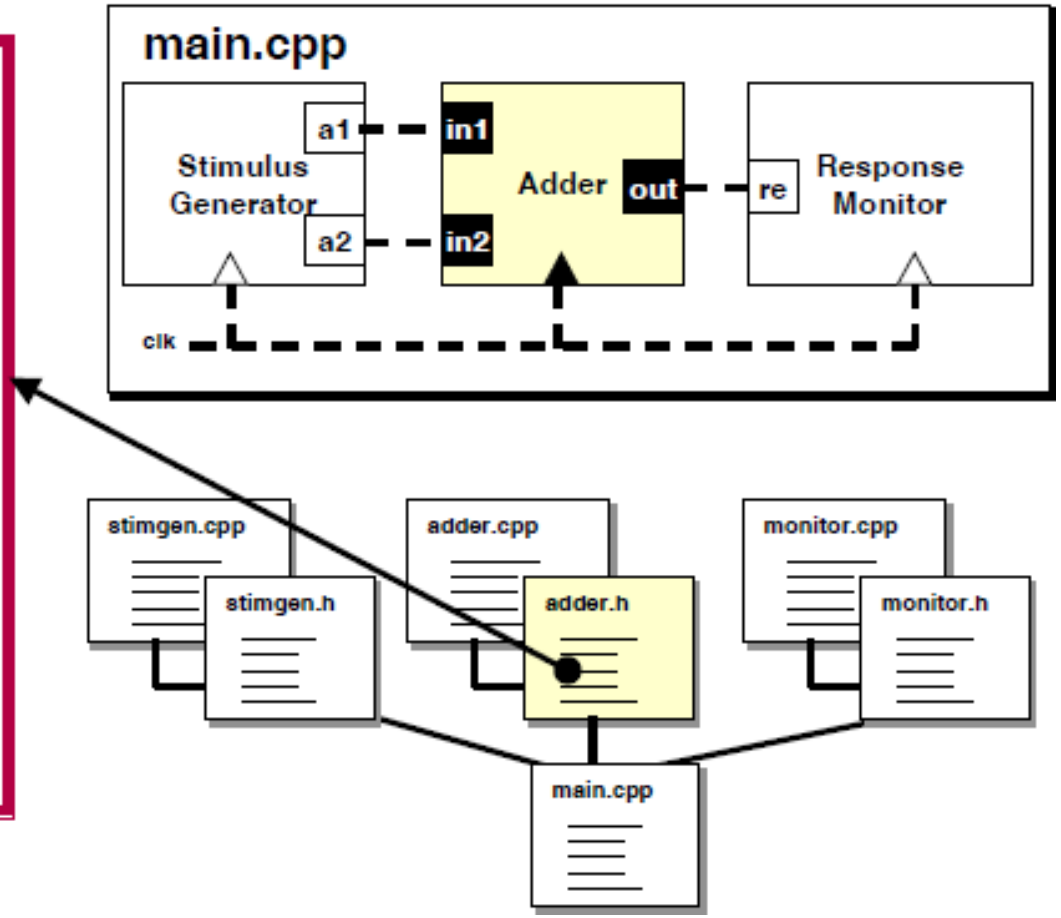
```
// Implementation file adder.cpp
#include "systemc.h"
#include "adder.h"

void adder::main()
{
   while (true) {
      out->write(in1->read() + in2->read());
      // assign a run-time to process
      wait(10, SC_NS);
   }
}
```



Basic example

- Behavioural model

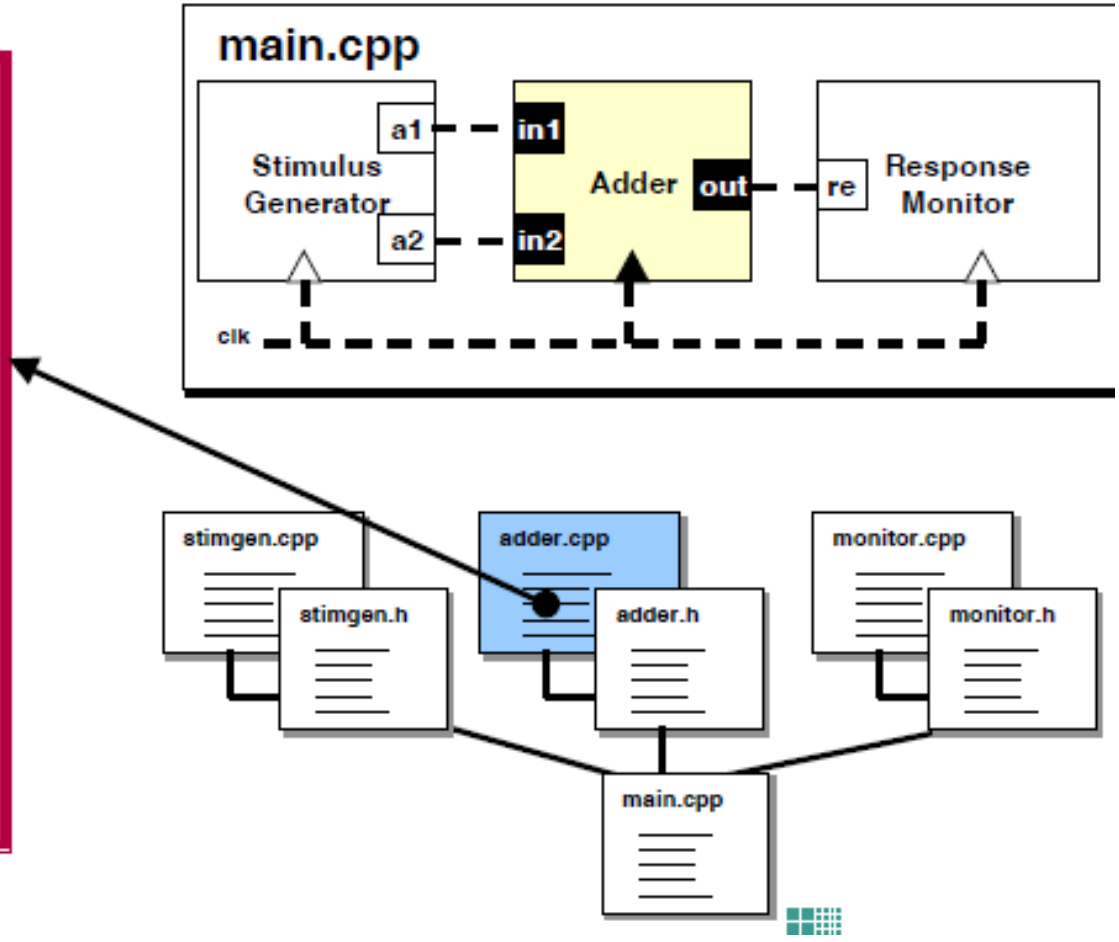
```
// header file adder.h

typedef sc_int<8> T_ADD;

SC_MODULE(adder) {
   // Clock introduced
   sc_in_clk        clk;
   // Input ports
   sc_in<T_ADD>     in1;
   sc_in<T_ADD>     in2;
   // Output port
   sc_out<T_ADD>    out;

   // Constructor
   SC_CTOR(adder){
      SC_CTHREAD(main, clk.pos());
   }

   // Functionality of the process
   void main();
};
```



Basic example

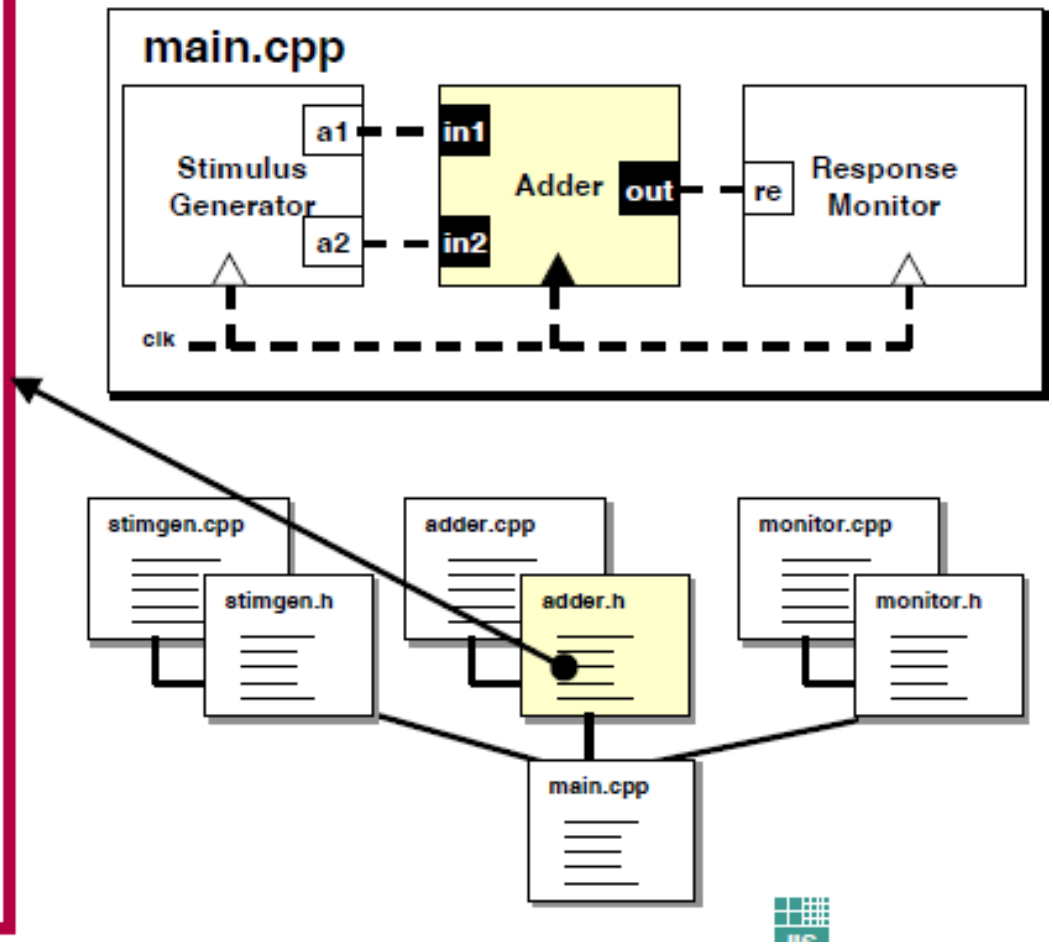
- Behavioural model

```
// Implementation file adder.cc
#include "systemc.h"
#include "adder.h"

void adder::main()
{
      // initialization
      T_ADD __in1 = 0;
      T_ADD __in2 = 0;
      T_ADD __out = 0;

      out.write(__out);
      wait();

      // infinite loop
      while(1) {
            __in1 = in1.read();
            __in2 = in2.read();
            __out = __in1 + __in2;
            out.write(__out);
            wait();
      }
}
```



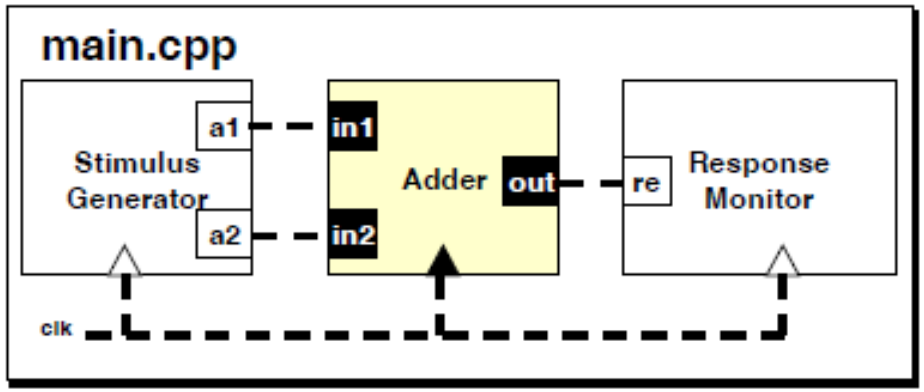
Basic example
RTL model

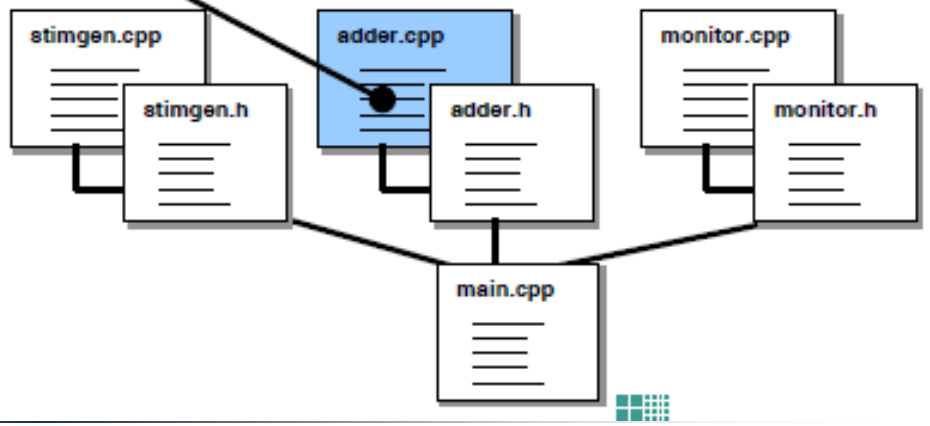
```
// header file adder.h

typedef sc_int<8> T_ADD;

SC_MODULE(adder) {
   // Clock introduced
   sc_in_clk        clk;
   // Input ports
   sc_in<T_ADD>     in1;
   sc_in<T_ADD>     in2;
   // Output port
   sc_out<T_ADD>    out;

   // internal signal
   sc_signal<T_ADD> sum;

   // Constructor
   SC_CTOR(adder){
      SC_METHOD(add);
      sensitive << in1 << in2;
      SC_METHOD(reg);
      sensitive_pos << clk;
   }

   // Functionality of the process
   void add();
   void reg();
};
```



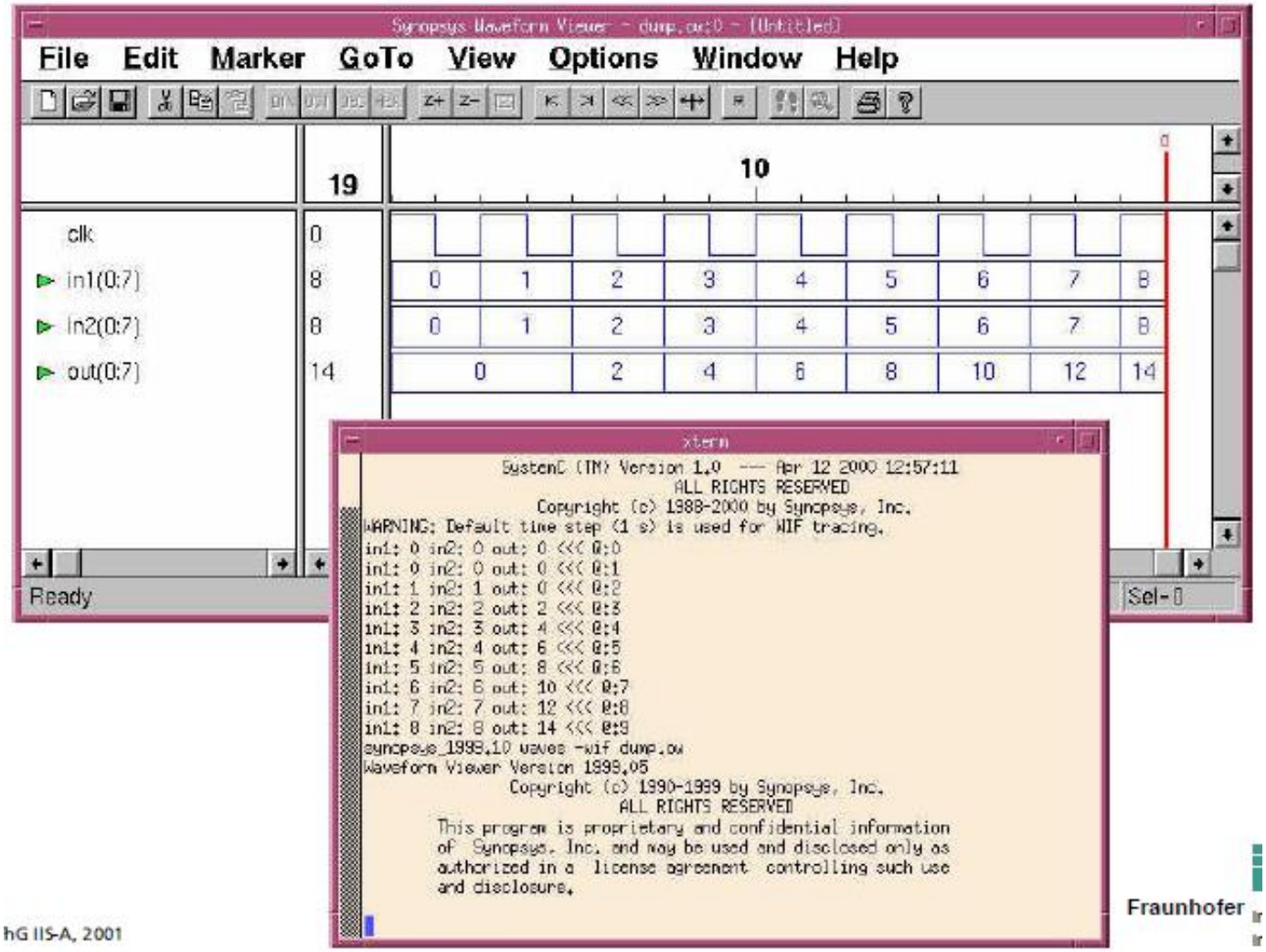
Basic example
RTL model

```
// Implementation file adder.cc
#include "systemc.h"
#include "adder.h"

void adder::add()
{
     T_ADD __in1 = in1.read();
     T_ADD __in2 = in2.read();

     sum.write( __in1 + __in2 );
}

void adder::reg()
{
     out.write( sum );
}
```





Screenshot


Synopsys Waveform Viewer - dump.ow:0 - [Untitled]
File Edit Marker GoTo View Options Window Help
19
10
clk
in1(0:7)
in2(0:7)
out(0:7)
Ready
Sel-0
xterm
SystemC (TM) Version 1.0 --- Apr 12 2000 12:57:11
ALL RIGHTS RESERVED
Copyright (c) 1988-2000 by Synopsys, Inc.
WARNING: Default time step (1 s) is used for WIF tracing.
in1: 0 in2: 0 out: 0 <<< @:0
in1: 0 in2: 0 out: 0 <<< @:1
in1: 1 in2: 1 out: 0 <<< @:2
in1: 2 in2: 2 out: 2 <<< @:3
in1: 3 in2: 3 out: 4 <<< @:4
in1: 4 in2: 4 out: 6 <<< @:5
in1: 5 in2: 5 out: 8 <<< @:6
in1: 6 in2: 6 out: 10 <<< @:7
in1: 7 in2: 7 out: 12 <<< @:8
in1: 8 in2: 8 out: 14 <<< @:9
synopsys_1999.10 waves -wif dump.ow
Waveform Viewer Version 1999.05
Copyright (c) 1990-1999 by Synopsys, Inc.
ALL RIGHTS RESERVED
This program is proprietary and confidential information
of Synopsys, Inc. and may be used and disclosed only as
authorized in a license agreement controlling such use
and disclosure.
hG IIS-A, 2001
Fraunhofer

More info…

- Official site
  - www.accellera.org (ex www.systemc.org)
    - SystemC 2.3.1
    - http://www.accellera.org/downloads/standards/systemc
  - One of the available online free tutorial
    - https://www.doulos.com/knowhow/systemc/
  - A good book
    - *SystemC: From the Ground Up*
      - Second Edition Springer - 2009
  - A world to be discovered…
    - www.soclib.fr
      - Tutorial: https://www.soclib.fr/appliance/