

SystemC: an overview

ET 4351

Alexander de Graaf, EEMCS/ME/CAS

12-5-2010

Acknowledgement

- These slides are based upon:

F. Angiolini, "SystemC Tutorial", DEIS Univ. di Bologna, Italy
R.A. Shafik, B.H. Al-Hashimi, "Electronic Systems Design with SystemC", Univ. of Southampton, United Kingdom
D.C. Black, J. Donovan, "SystemC From The Ground Up",Eklectic Ally Inc., USA

Outline.

1. *Introduction*
2. *Modules and hierarchy*
3. *A notion of time*
4. *Concurrency*
5. *Interfaces, channels and ports*

1.

Introduction

Motivations

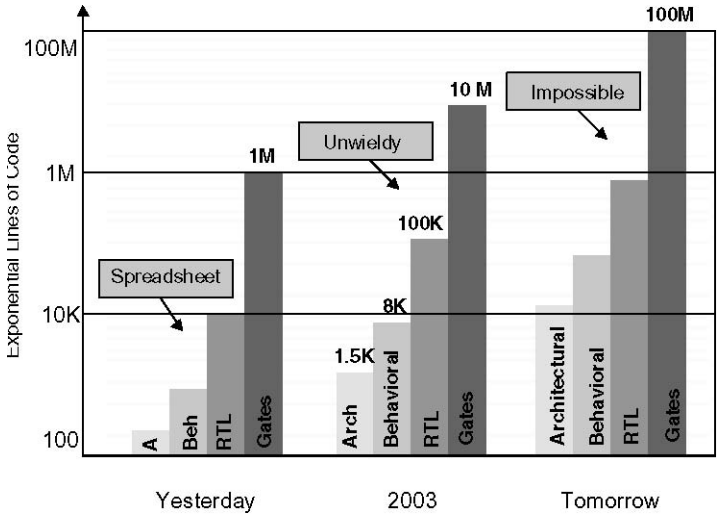


Figure 1-2. Design Complexity from Different Design Generation

- The complexity of SoCs keeps increasing
- Using gate-level or even RTL simulation to characterize and explore a complete SoC for typical use-case scenarios is not feasible:
  - Building models at these levels is quite time consuming
  - Complete models are usually ready at a late phase of the system design
  - **Simulation is too slow!**
- Solution: model at higher abstraction level → SystemC
  - Fewer architectural details → earlier in the design flow
  - Higher simulation speed
  - Possibility to simulate more complex systems

What is SystemC..?

- SystemC is a HDL, design capabilities at
  - Register Transfer Level
  - Behavioural Level Modelling
  - Transaction Level Modeling
- SystemC provides
  - Notion of timing
  - Process Concurrency
  - Event Sequencing,

What is SystemC?

- A C++ class library that allows the functional modeling of systems
  - hierarchical decomposition of a system into *modules*
  - structural connectivity between those modules using ports and exports
  - scheduling and synchronization of concurrent processes using events and sensitivity
  - passing of simulated time
  - separation of computation (processes) from communication (channels)
  - independent refinement of computation and communication using interfaces
  - Hardware-oriented data types for modeling digital logic and fixed-point arithmetic

Class Basics

```
#ifndef _IMAGE_H_
#define _IMAGE_H_

#include <assert.h>
#include "vectors.h"

class Image {

public:
   ...

private:
   ...

};

#endif
```

Prevents multiple references

Include a library file

Include a local file

Variables and functions accessible from anywhere

Variables and functions accessible only from within this class

Creating an instance

Stack allocation

```
Image myImage;
myImage.SetAllPixels(ClearColor);
```

Heap allocation

```
Image *imagePtr;
imagePtr = new Image();
imagePtr->SetAllPixels(ClearColor);

...

delete imagePtr;
```

Organizational Strategy

image.h

Header file: Class definition & function prototypes

```
void SetAllPixels(const Vec3f &color);
```

image.C

.C file: Full function definitions

```
void Image::SetAllPixels(const Vec3f &color) {
    for (int i = 0; i < width*height; i++)
      data[i] = color;
}
```

main.C

Main code: Function references

```
myImage.SetAllPixels(clearColor);
```

Constructors & Destructors

```
class Image {
public:
  Image(void) {
    width = height = 0;
    data = NULL;
  }

 ~Image(void) {
    if (data != NULL)
      delete[] data;
  }

  int width;
  int height;
  Vec3f *data;
};
```

Constructor:
Called whenever a new instance is created

Destructor:
Called whenever an instance is deleted

Constructors

Constructors can also take parameters

```
Image(int w, int h) {
  width = w;
  height = h;
  data = new Vec3f[w*h];
}
```

Using this constructor with stack or heap allocation:

```
Image myImage = Image(10, 10);
```

stack allocation

```
Image *imagePtr;
imagePtr = new Image(10, 10);
```

heap allocation

The Copy Constructor

```
Image(Image *img) {
  width = img->width;
  height = img->height;
  data = new Vec3f[width*height];
  for (int i=0; i<width*height; i++)
    data[i] = new data[i];
}
```

A default copy constructor is created automatically, but it is usually not what you want:

```
Image(Image *img) {
  width = img->width;
  height = img->height;
  data = img->data;
```

Passing Classes as Parameters

If a class instance is passed by value, the copy constructor will be used to make a copy.

```
bool IsImageGreen(Image img);
```

Computationally expensive

It's much faster to pass by reference:

```
bool IsImageGreen(Image *img);
```

or

```
bool IsImageGreen(Image &img);
```

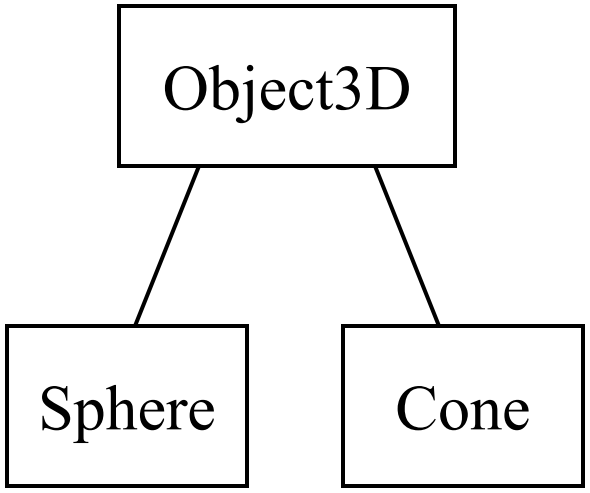
Class Hierarchy

Child classes inherit parent attributes

```
class Object3D {
  Vec3f color;
};

class Sphere : public Object3D {
  float radius;
};

class Cone : public Object3D {
  float base;
  float height;
};
```



Class Hierarchy

Child classes can *call* parent functions

```
Sphere::Sphere() : Object3D() {
  radius = 1.0;
}
```

Call the parent constructor

Child classes can *override* parent functions

```
class Object3D {
  virtual void setDefaults(void) {
    color = RED; }
};

class Sphere : public Object3D {
  void setDefaults(void) {
    color = BLUE;
    radius = 1.0 }
```

Virtual Functions

A superclass pointer can reference a subclass object

```
Sphere *mySphere = new Sphere();
Object3D *myObject = mySphere;
```

If a superclass has virtual functions, the correct subclass version will automatically be selected

Superclass

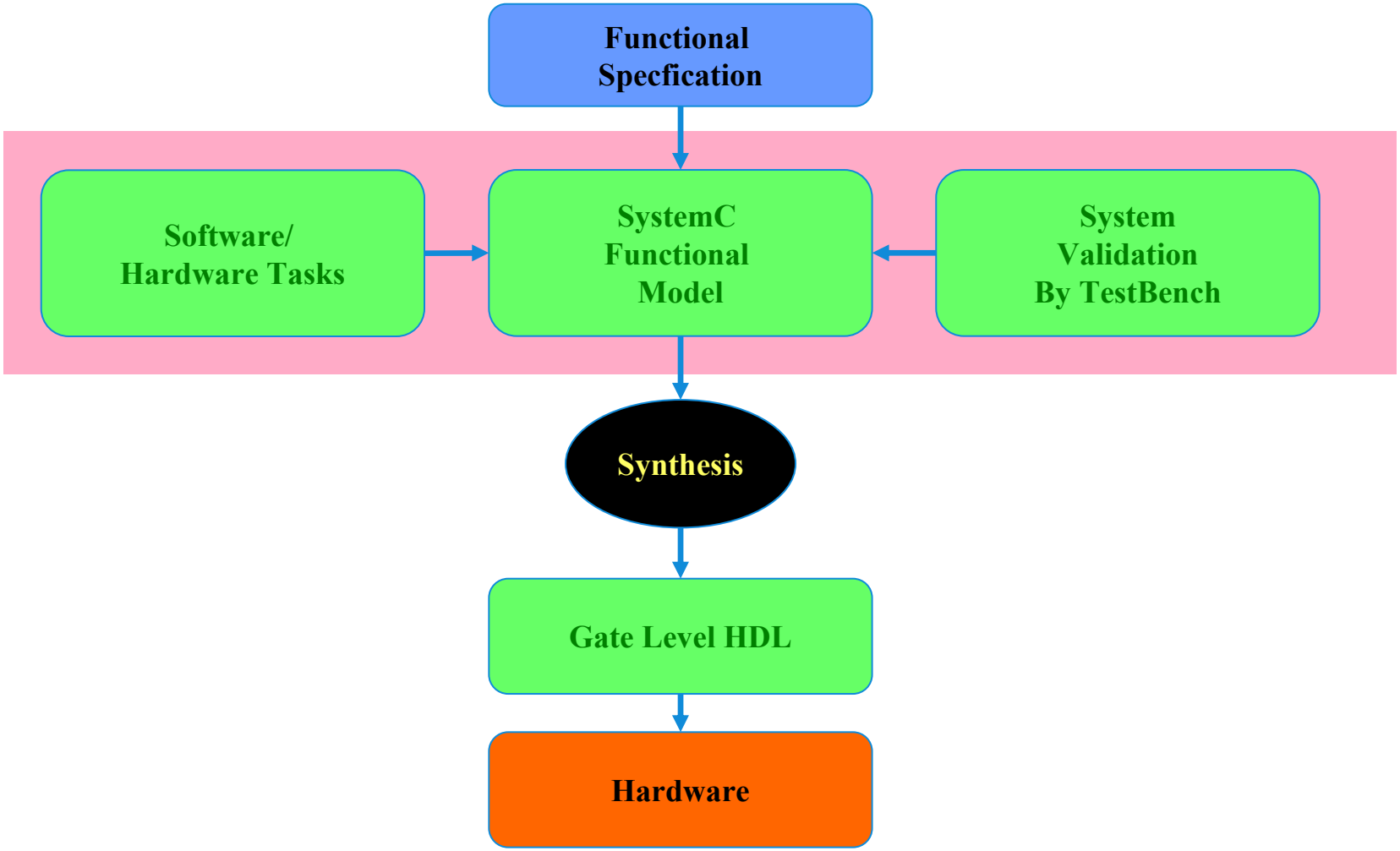
```
class Object3D {
  virtual void intersect(Vec3f *ray, Vec3f *hit);
};
```

Subclass

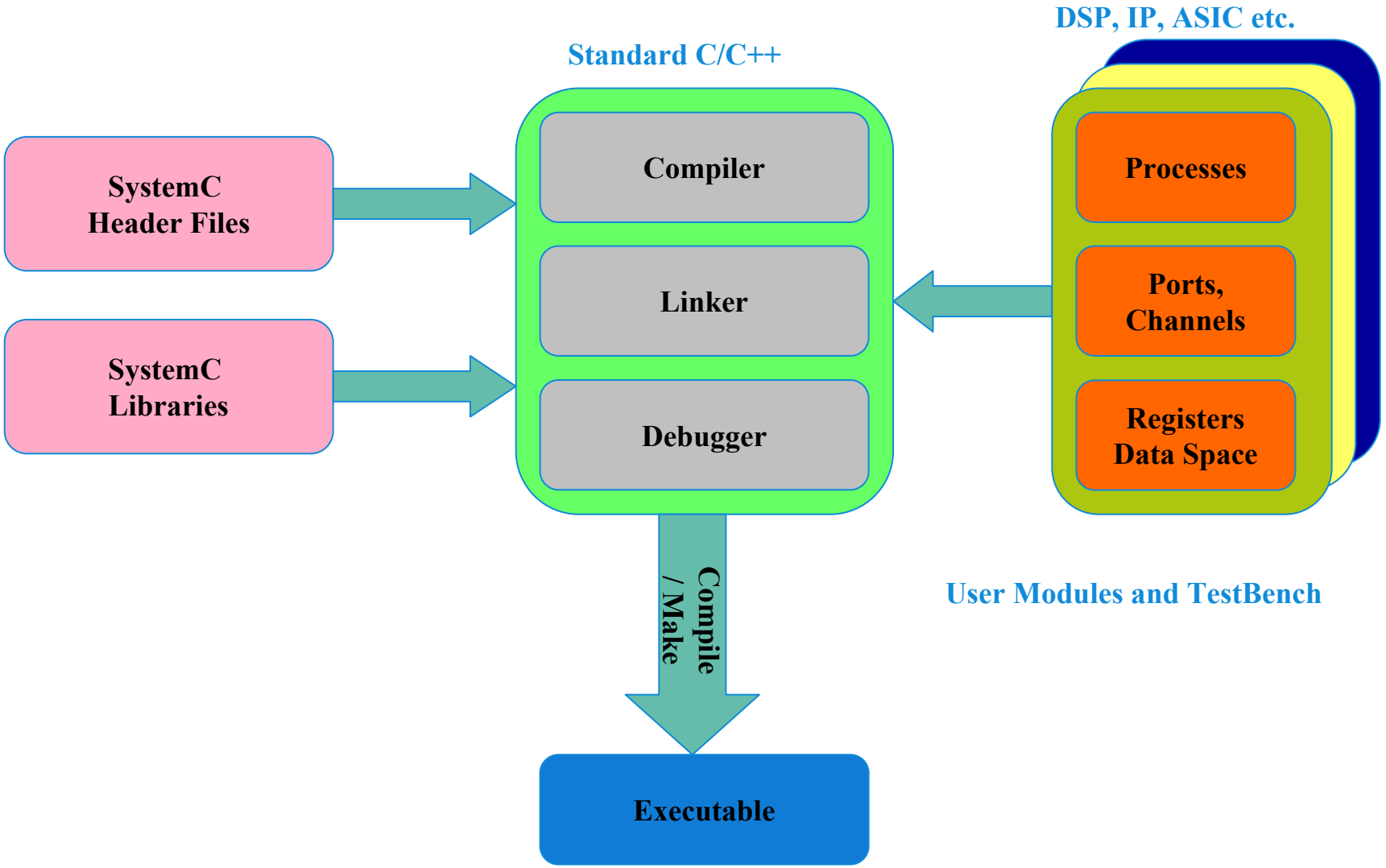
```
class Sphere : public Object3D {
  virtual void intersect(Vec3f *ray, Vec3f *hit);
};
myObject->intersect(ray, hit);
```

Actually calls `Sphere::intersect`

SystemC Design Flow



How SystemC works..?



SystemC and VHDL



SystemC Module

- Modules are basic building block
- Constructors are needed to create an instance of module
- Processes registered within Constructors



VHDL Entity

- Entity is the basic building block
- No Constructor needed.
- Processes registered within Architectures

2.

Modules and hierarchy

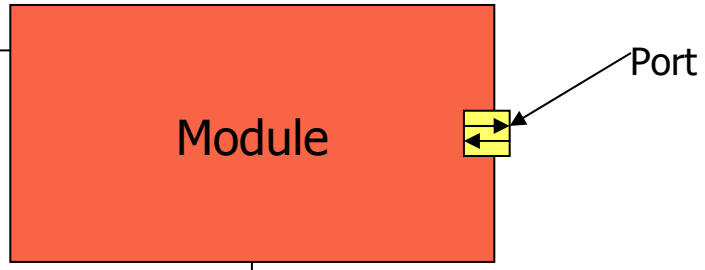
Modules and hierarchy

- Modules are the basic building blocks for partitioning a design
- A module is a structural entity, which can contain *processes*, *ports*, *channels*, *member functions not registered as processes* and *instances of other modules*
- A module is the foundation of structural hierarchy
- Modules allow designers to hide internal data representation and algorithms from other modules
- Designers are forced to use public interfaces to other modules, thus making the entire system easier to change and maintain → reusability = lower design time

- How to declare a module:
  - Either by using the macro

    SC_MODULE (transmit) {…};
  - or with explicit use of classes

    class transmit : public sc_module {…};



Module instantiation

- Declaration

```
SC_MODULE(ex2)
{
  . . .
  ex1 ex1_instance;

  SC_CTOR(ex2) :
  ex1_instance("ex1_anyname")
 {
   //body
 }

};
```

- Pointer

```
SC_MODULE(ex2)
{
  . . .
  ex1 *ex1_instance;

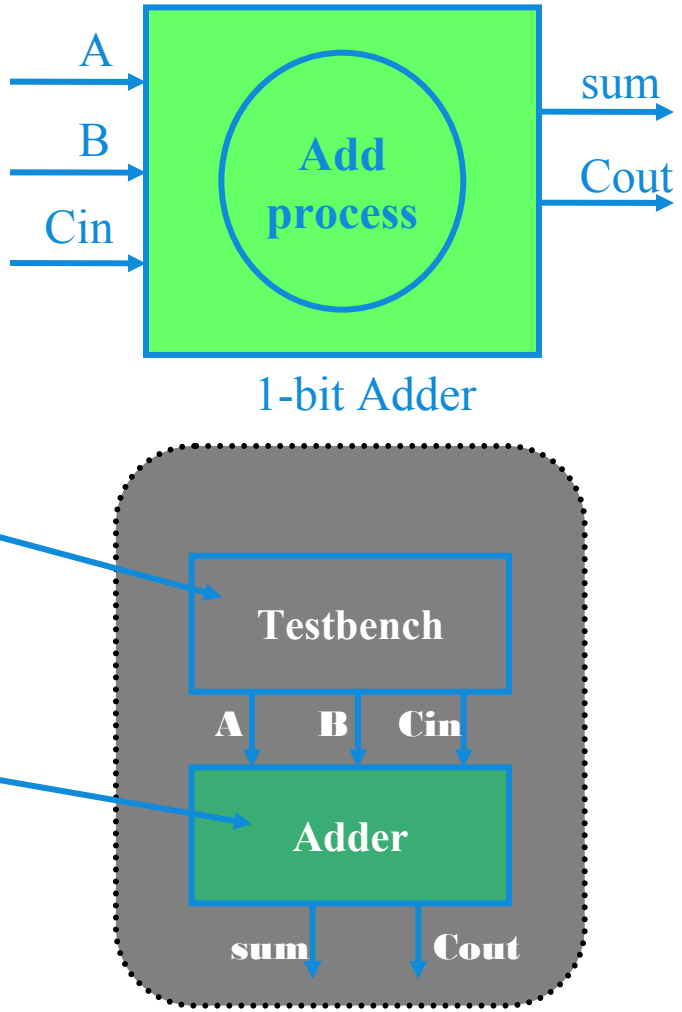
  SC_CTOR(ex2)
{
   ex1_instance = new
  ex1("ex1_anyname");
   //body
 }
};
```

Processes in SystemC

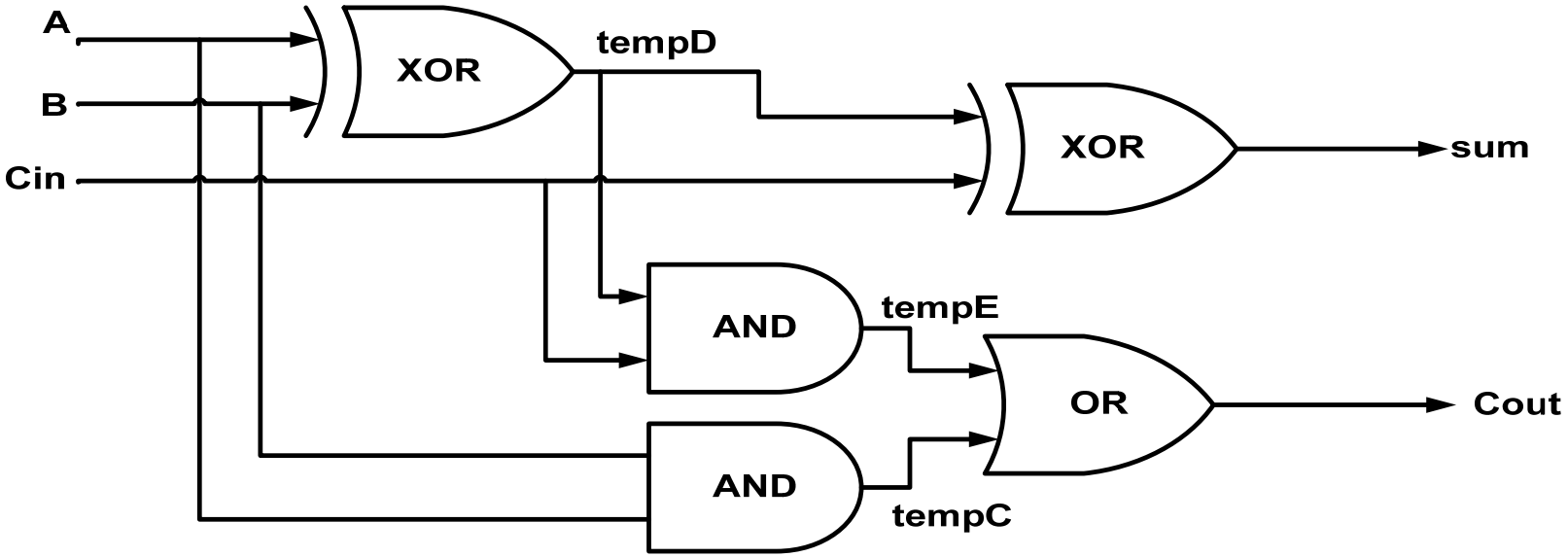
- An argument that begs the question:
- Processes are small pieces of code that run concurrently with other processes.
- All high-level system level design (SLD) tools use an underlying model of a network of processes.
- Functionality is described in processes. Processes must be contained within a module.
- They are registered as processes with the SystemC kernel, using a process declaration in the module constructor.
- Processes accept no arguments and produce no output

Example System: 1-bit Adder

- Basic 1-bit Adder Module
- Testbench
- Top-level Module



Example: A Structural Model for 1-bit Adder



Modeling the Adder circuit

- An **Adder** is a combinatorial circuit. Its output is purely a function of its input.
- Use the simplest kind of SystemC process to model it: **SC_METHOD**.
- SC_METHODs are simply C++ functions. Therefore, the SystemC class library must make them behave like processes, in particular:
  - The SystemC class library contains a simulation kernel – a piece of code that models the passing of time, and calls functions to calculate their outputs whenever their inputs change.
  - The function must be declared an SC_METHOD and made sensitive to its inputs.

Basic Module Structure Adder Module

```
#include header files

SC_MODULE(module_name){
   Port or Signal Declarations;

   Clock (if synch.);

   Process/Functions Declarations;

   SC_CTOR(module_name){
            Process Registration;
            Static Sensitivity List;

   }
};
```

```
//1-Bit Adder, Adder.h
#include "systemc.h"

SC_MODULE(Adder){
   sc_out<sc_logic> sum, Cout;
   sc_in<sc_logic> A, B, Cin;

   void add(){
        sc_logic tempC, tempD, tempE;
        tempC = A.read() & B.read();
        tempD = A.read() ^ B.read();
        tempE = Cin.read() & tempD.read();
        sum.write(tempD ^ Cin.read());
        cout.write(tempC | tempE);
    }
   SC_CTOR(Adder){
            SC_METHOD(add);
            sensitive << A << B << Cin;
   }
};
```

VHDL versus SystemC

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Adder is
        port(A, B, Cin : in  STD_LOGIC;
             sum, Cout : out STD_LOGIC);
end Adder;

architecture ARCH of Adder is
begin
   add: process (A, B, Cin)
   begin
      tempC, tempD, tempE: STD_LOGIC;
      tempC<= (A and B);
      tempD<=(A xor B);
      tempE<=(Cin and tempD);
      sum<=(tempD xor Cin);
      cout<=(tempC or tempE);
   end process add;
end ARCH;
```

```
#include "systemc.h"

SC_MODULE(Adder){
   sc_out<sc_logic> sum, Cout;
   sc_in<sc_logic> A, B, Cin;

   void add(){
        sc_logic tempC, tempD, tempE;
        tempC = A.read() & B.read();
        tempD = A.read() ^ B.read();
        tempE = Cin.read() & tempD.read();
        sum.write(tempD ^ Cin.read());
        cout.write(tempC | tempE);
     }
   SC_CTOR(Adder){
            SC_METHOD(add);
            sensitive << A << B << Cin;
   }
};
```

What just happened?

- **SC_MODULE:**
  Hierarchy in SystemC is created by using a class sc_module. sc_module may be used directly, or may be "hidden" using the macro SC_MODULE. The example SC_MODULE above creates an sc_module class object called Adder.
- **Ports:**
  SystemC numerous predefined ports such as sc_in<>, sc_out<>, sc_inout<>. The language also allows the definition of user defined ports through the use of the sc_port<> class.

```
sc_port<sc_signal_in_if<bool>, 1> clk;
```

Constructor for sc_module Instance Adder

SystemC provides a shorthand way of writing the constructor using a macro SC_CTOR.

This constructor does the following:

- Create hierarchy (none in this case)
- Register functions as processes with the simulation kernel
- Declare sensitivity list for processes
- Accepts one argument only, which is the module name

- In this example, the constructor declares that **add** is a SC_METHOD, and says that any event on ports A, B and Cin must make the kernel run the function (calculate new values for sum and Cout).

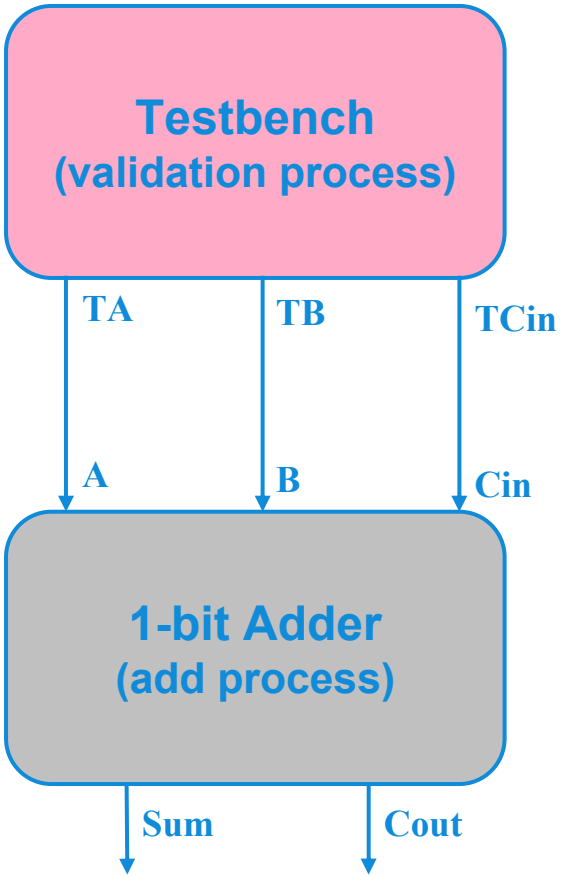
Alternative constructor: SC_HAS_PROCESS

- The SC_HAS_PROCESS can be used in 2 situations:
  - If we need arguments beyond just the instance name string passed into SC_CTOR (i.e. to provide configurable modules)
  - If we want to place the constructor into the implementation (i.e. *.cpp file)

```
//FILE: module_name.h
SC_MODULE(module_name) {
  SC_HAS_PROCESS(module_name);
  module_name(sc_module_name instname[, other_args…])
   : sc_module(instname)
   [, other_initializers]
   {
     CONSTRUCTOR_BODY
   }
};
```

Figure 5-12. Syntax of SC_HAS_PROCESS in the Header

TestBench




```
#include "systemc.h" //Testbench.h

SC_MODULE(Testbench){
   sc_out<sc_logic> TA, TB, TCin;

   void testprocess()
{
            TA.write(SC_LOGIC_0);
            TB.write(SC_LOGIC_0);
            TCin.write(SC_LOGIC_0);
            wait(5, SC_NS);
            TA.write(SC_LOGIC_1);
            TB.write(SC_LOGIC_1);
            TCin.write(SC_LOGIC_0);
            wait(10, SC_NS);

            sc_stop(); //Stop simulation
   }

   SC_CTOR(Testbench){
            SC_THREAD(testprocess);
   }
};
```

Top-Level Module

```
Include module headers
Include testbench header(s)

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

   Module Instantiations;
   Binding signals with modules;

   Testbench Module Instantiation;
   Binding signals with testbench;

   sc_start(time);
    //sc_start(-1) infinite simulation
   return 0;
}
```

```
//AdderMain.cpp
#include "Adder.h"
#include "Testbench.h"

int main(int argc, char *argv[]){
   sc_signal<sc_logic> A, B, Cin;
   sc_signal<sc_logic> sum, Cout;

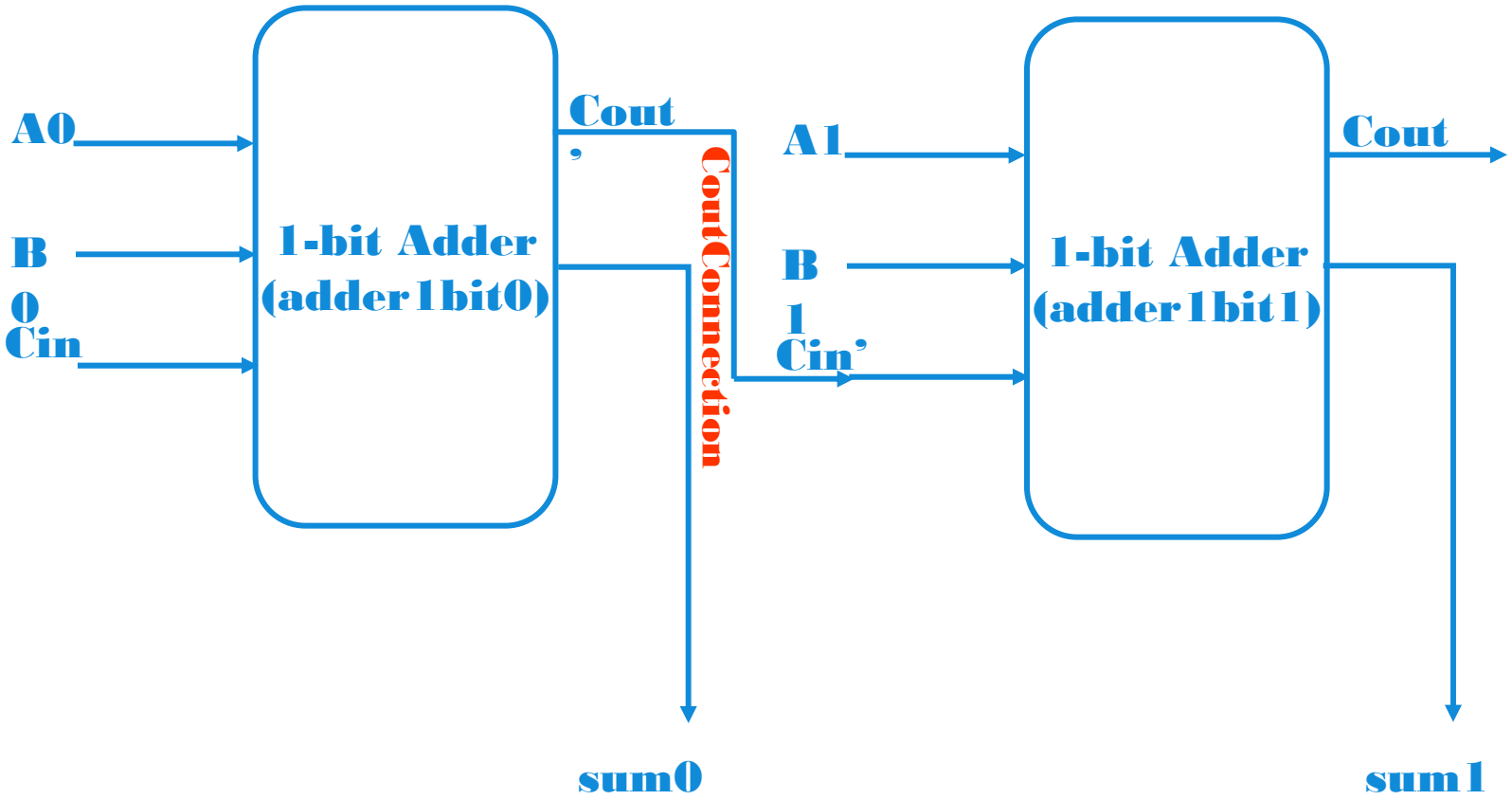
   Adder adder1bit("BitAdder");
   Testbench tb("Testbench");

   adder1bit.A(A);
   adder1bit.B(B);
   adder1bit.Cin(Cin);
   adder1bit.sum(sum);
   adder1bit.Cout(Cout);

    tb.TA(A);
    tb.TB(B);
    tb.TCin(Cin);

   //Start the simulation for 200ns
   sc_start(200, SC_NS);
   return 0;
}
```

Hierarchy example: A simple 2-bit Adder



2-Bit Adder Basic Module

```
#include "Adder.h"
SC_MODULE(Adder2Bits) {
    sc_in<sc_logic> A0, B0, A1, B1,Cin;
    sc_out<sc_logic> sum0, sum1, Cout;
    sc_signal<sc_logic> CoutConnection;

   Adder adder1bit0, adder1bit1;

   SC_CTOR(Adder2Bits):
    adder1bit0("Adder0"),
    adder1bit1("Adder1")
    {
    adder1bit0.A(A0);
    adder1bit0.B(B0);
    adder1bit0.Cin(Cin);
    adder1bit0.sum(sum0);
    adder1bit0.Cout(CoutConnection);
    adder1bit1.Cin(CoutConnection);

    adder1bit1.A(A1);
    adder1bit1.B(B1);
    adder1bit1.sum(sum1);
    adder1bit1.Cout(Cout);
  }
};
```

- **Note the basic 1-bit Adder is now being used within *Adder1Bits* Module as a sub-module**
- **1-bit Adders provide the basic functionality of an adder**

- **Interconnections between sub-module components.**
- **Notice that the interconnections are carried out port-to-port**

What just Happened II?

- **Header Files:**

Note the inclusion of Adder.h. This allows access to the module containing the **Adder** circuit.

- **Ports:**

It is permitted to reuse the names A, B, Cin and sum, Cout as this a different level of the hierarchy.

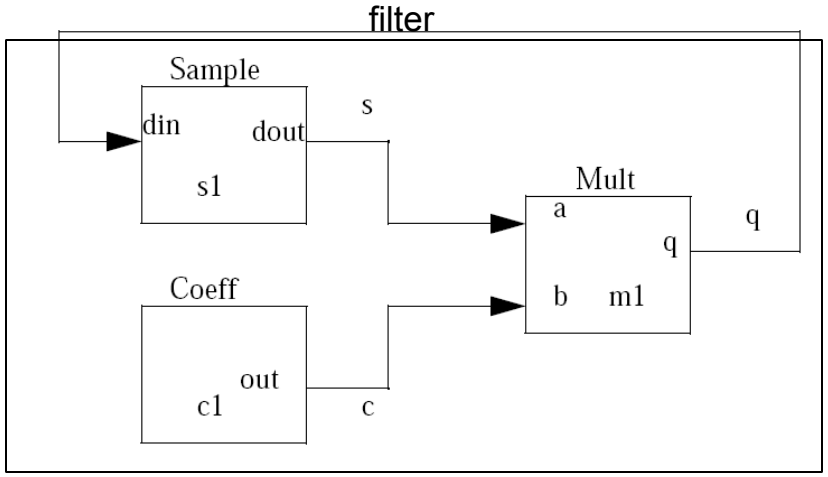
- **Declaring Intermediate Wires:**

They are declared using sc_signal. sc_signal is a class with a template parameter specifying the type of data the signal can hold – sc_logic in this example. It is a primitive, built-in channel with the SystemC library.

Adder2Bits Constructor

- Must have two instances of Adder. After the port declarations, they are declared. And a label must be declared for each instance.
- The two labels, Adder0, Adder1 are passed to the constructors of the instances of Adder by using an initializer list on the constructor of Adder2Bits.
- Lastly, the ports are wired up. This can be done by **position** or by **name**

Module signals



- Signals are used to connect ports of lower-level modules together
- Signal can be local to a module
- Signals carry data, while ports determine the direction of data
- Signals can be connected to ports in 2 ways:
  - Positional connection
  - Named connection

Module signals (cont'ed)

- Positional connection

```
// filter.h
#include "systemc.h"
#include "mult.h"
#include "coeff.h"
#include "sample.h"

SC_MODULE(filter) {

  sample *s1;
  coeff  *c1;
  mult   *m1;
  sc_signal<sc_uint<32> > q, s, c;

  SC_CTOR(filter) {
    s1 = new sample ("s1");
    (*s1)(q,s);

    c1 = new coeff ("c1");
    (*c1)(c);

    m1 = new mult ("m1");
    (*m1)(s,c,q);
  }
}
```

Include files containing the systemC classes and the declarations of the instantiated modules

Pointer variables are declared for each object that will be instantiated later

Local signals are declared

•The 3 modules are instantiated in the constructor SC_CTOR:
• The first line creates a new object and a pointer to it
•The second line uses the pointer to map signals to the object ports. The order is important!!

- Such a connection works well for instantiations with few ports

Module signals (cont'ed)

Named connection

```
#include "systemc.h"
#include "mult.h"
#include "coeff.h"
#include "sample.h"

SC_MODULE(filter) {
  sample *s1;
  coeff  *c1;
  mult   *m1;

  sc_signal<sc_uint<32> > q, s, c;

  SC_CTOR(filter) {
    s1 = new sample ("s1");
    s1->din(q);
    s1->dout(s);

    c1 = new coeff ("c1");
    c1->out(c);

    m1 = new mult ("m1");
    m1->a(s);
    m1->b(c);
    m1->q(q);
  }
}
```

- The 3 modules are instantiated in the constructor SC_CTOR
- Using named connection allows creating connections in any order

- Such a connection works well also for instantiations with many ports

More on Modules

- They map functionality of HW/SW blocks
- The basic building block of every system
- Can contain a whole hierarchy of sub-modules and provide private variable/signals.
- Modules can interface to each other via ports/interfaces/channels
- Module functionality is achieved by means of processes

3.

Types

SystemC Data Types

Type	Description
sc_logic	Simple bit with 4 values(0/1/X/Z)
sc_int	Signed Integer from 1-64 bits
sc_uint	Unsigned Integer from 1-64 bits
sc_bigint	Arbitrary size signed integer
sc_biguint	Arbitrary size unsigned integer
sc_bv	Arbitrary size 2-values vector
sc_lv	Arbitrary size 4-values vector
sc_fixed	templated signed fixed point
sc_ufixed	templated unsigned fixed point
sc_fix	untemplated signed fixed point
sc_ufix	untemplated unsigned fixed point

SystemC data types are (along with C/C++ types):

- **bool** 2 value single bit type [0 or 1]
  - bool A, B;
  - sc_in<bool > input;
- **sc_logic** 4 value single bit type [0, 1, X or Z]
  - sc_logic C, D;
  - sc_out<sc_logic > E;
- **sc_int** 1 to 64 bit signed integer type
  - sc_int<16> x, y;
  - sc_out<sc_int<16> > z;
- **sc_time** time (units: SC_PS, SC_NS, SC_MS etc.)
  - sc_time t1(10, SC_NS)

Fast Fixed-Point Data Types

- Arbitrary Precision vs. Simulation Speed
- Achieving Faster Speed
  - Use ***double*** as underlying data type
  - Mantissa limited to 53 bits
  - Range limited to that of ***double***
- Fast Fixed-Point Types
  - **sc_fixed_fast, sc_ufixed_fast**
  - **sc_fix_fast, sc_ufix_fast**
- Exactly the same declaration format and usage as before
- All fixed-point data types, can be mixed freely

Fast Fixed-Point Data Types (2)

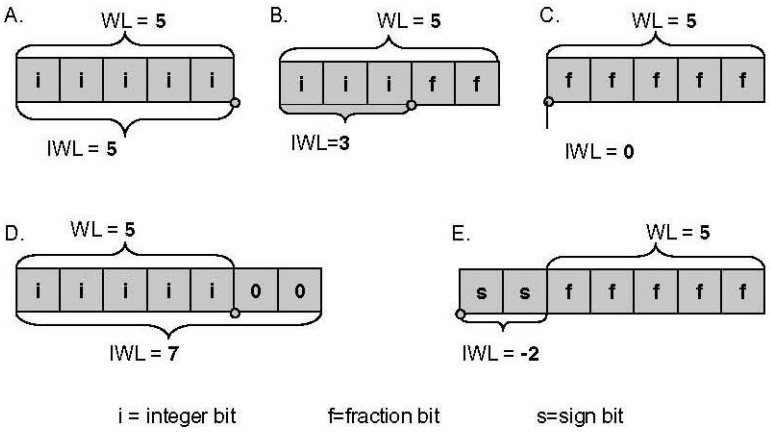
sc_fixed_fast <wl, iwl [, quant [, ovflw [, nbits]>

sc_ufixed_fast <…> ! stat. limited to 53 bits

sc_fix_fast, sc_ufix_fast ! dyn. limited to 53 bits

sc_fixed, sc_ufixed ! static precision

sc_fix, sc_ufix ! dynanmic precision



Name	Overflow Meaning
SC_SAT	Saturate
SC_SAT_ZERO	Saturate to zero
SC_SAT_SYM	Saturate symmetrically
SC_WRAP	Wraparound
SC_WRAP_SYM	Wraparound symmetrically

Name	Quantization Mode
SC_RND	Round
SC_RND_ZERO	Round towards zero
SC_RND_MIN_INF	Round towards minus infinity
SC_RND_INF	Round towards infinity
SC_RND_CONV	Convergent rounding[13]
SC_TRN	Truncate
SC_TRN_ZERO	Truncate towards zero

4.

Concurrency
Processes and events

Processes

- They describe the functionality of the system, thus expressing concurrency and parallel activities in the system
- They are contained in *modules*, and they access external *channel interfaces* through the *ports* of a module
- There are 3 different types of processes, depending on how they are called and executed:
  - *Method processes*
  - *Thread processes*
  - *Clocked thread processes (deprecated)*
- Processes are not hierarchical, so no process will call another process directly
- Processes can call methods and functions that are not processes
- Processes take no arguments and return none
- Processes are fired according to their sensitivity list, that can be both *dynamic* and *static*

SC_METHOD

- **SC_METHOD** is a SystemC process with following properties
  - SC_METHOD is executed **repeatedly** as one of its sensitivity signals changes state/value
  - SC_METHOD cannot be suspended during execution: they cannot call *wait()*
  - Such processes must also avoid using calls to blocking methods
  - They run completely and then return
- SC_METHOD processes are registered as

  SC_METHOD(process_name);

  sensitivity << signal1 << signal2 << …. ;

SC_THREAD

- **SC_THREAD** is a SystemC process with following properties
  - SC_THREAD is executed **only once** as one of its sensitivity signals changes state/value.
  - SC_THREAD can be suspended during execution using
    - *wait()*, generally for sensitivity based execution
    - *wait(time)*, generally for testbenches
    - *wait(event),* for dynamic sensitivities [to be shown later]
- SC_THREAD processes are registered as

  SC_THREAD(process_name);

  sensitivity << signal1 << signal2 << …. ;

Thread processes: SC_THREAD (blocking)

- They are started once and only once by the simulator
- Once started, they gain complete control on the simulation until they choose to return control to the simulator
- They have 2 ways to pass control back to simulator:
  - Simply exit: the thread is terminated for the rest of simulation
  - Invoke the module *wait()* method
- They can contain *wait()* functions that suspend process execution until an event occurs on one of the signals the process is sensitive to
- The process will continue execution until the next *wait()*

Method processes vs. thread processes

- A thread process is the most general process
- A thread process can be used to model nearly anything
- A thread process is slower than a method process, as *wait()* induces a context switch in the SystemC scheduler
- If simulation speed is important, try to prefer method processes, which are faster

dont_initialize()

- By default, all processes are executed initially, during the initialization phase
- It may be useful to have no initialization for some processes
  - i.e. when we know that a process needs a request from somewhere else
- Use *dont_initialize()*
- The use of dont_initialize() requires a static sensitivity list otherwise there would be nothing to start the process

```
…
SC_METHOD(attendant_method);
   sensitive(fillup_request);
   dont_initialize();
…
```

DFF Example: Basic Module with SC_METHOD

```
#include"systemc.h"

SC_MODULE(DFF) {
   sc_in<sc_bit> Data;
   sc_out<sc_bit> Q, Qbar;
   sc_in_clk clk;

   void store(){
            Q.write(Data.read());
            Qbar.write(~Q.read());
   }

   SC_CTOR(DFF){
            SC_METHOD(store);
            sensitive << clk.pos();
   }
};
```

Clock Inlet
For synchronous designs

DFF process
"Repeats execution when *clk* has positive edge"

Process Registration with SC_METHOD

DFF Example: Testbench with SC_THREAD

```
#include"systemc.h"

SC_MODULE(Testbench) {
   sc_out<sc_bit> TData;
   sc_in_clk Tclk;

   void testprocess(){
            TData.write(sc_bit('0'));
            wait();
            TData.write(sc_bit('1'));
            wait();
            TData.write(sc_bit('1'));
            wait(10, SC_NS);
            sc_stop();
   }

   SC_CTOR(Testbench){
            SC_THREAD(testprocess);
            sensitive << Tclk.pos();
   }
};
```

Clock Inlet
For synchronous designs

Validation process
"Executes only once"

Stop Simulation

Process Registration with SC_THREAD

DFF Example: Top-Level Module

```
#include"DFF.h"
#include"Testbench.h"

int sc_main(int argc, char *argv[]) {
   sc_signal<sc_bit> Data, Q, Qbar;
   sc_clock sysclock("clk", 20, SC_NS);

   DFF dff1("DFF1");
   Testbench test1("Test1");

   dff1.Q(Q);
   dff1.Data(Data);
   dff1.clk(sysclock);
   dff1.Qbar(Qbar);

   test1.TData(Data);
   test1.Tclk(sysclock);

   sc_start(400, SC_NS);
};
```

System Clock With period of 20ns

Interconnections With signals

Start Simulation For 400ns

Parameterized Modules: *Bi-directional Counter*

```
#include header files

SC_MODULE(module_name){
   Port or Signal Declarations;
   Clock (if synch.);
   Local Channel/Module/Variable;
   Process/Functions Declarations;
   SC_HAS_PROCESS(module_name);
   module_name(sc_module_name module_name,
   .....initialisations): sc_module(module_name){
            Process Registration;
            Static Sensitivity List;
            Module Instance;
            Internal Channel Binding;
            Initializations;
   }
};
```

```
#include "systemc.h" //Counter.h
SC_ MODULE(Counter){
   sc_out<sc_int<8> > out;
   sc_in_clk clk;
   sc_in<bool> reset. enable;
   sc_int<8> count;
   bool dir;
   void do_count(){
        while(true){
            if(dir == 0){
               if(reset.read()) count = 0;
               if(enable.read()) count++;
            }
            else{
               if(reset.read()) count = 255;
               if(enable.read()) count--;
            }
            wait();
        }
   }
   SC_HAS_PROCESS(Counter) ;
   Counter (sc_module_name Counter, int init_dir =
   0): sc_module(Counter){
        SC_THREAD(do_count);
        sensitive << clk.pos();
        dir = init_dir;
   }
} ;
```

Events

- Events are occurrence of signal values and changes.
- Events are meant to **trigger processes** of modules.
- An event has no duration or value. It can be used for
  - static sensitivity of processes, or
  - dynamic sensitivity of processes

Triggering events: *event*.notify()

- Events occur explicitely by using the ***notify()*** method
- If a nearer notification is scheduled, the previous outstanding scheduled event is canceled
- Events can be canceled with the **.cancel()** method

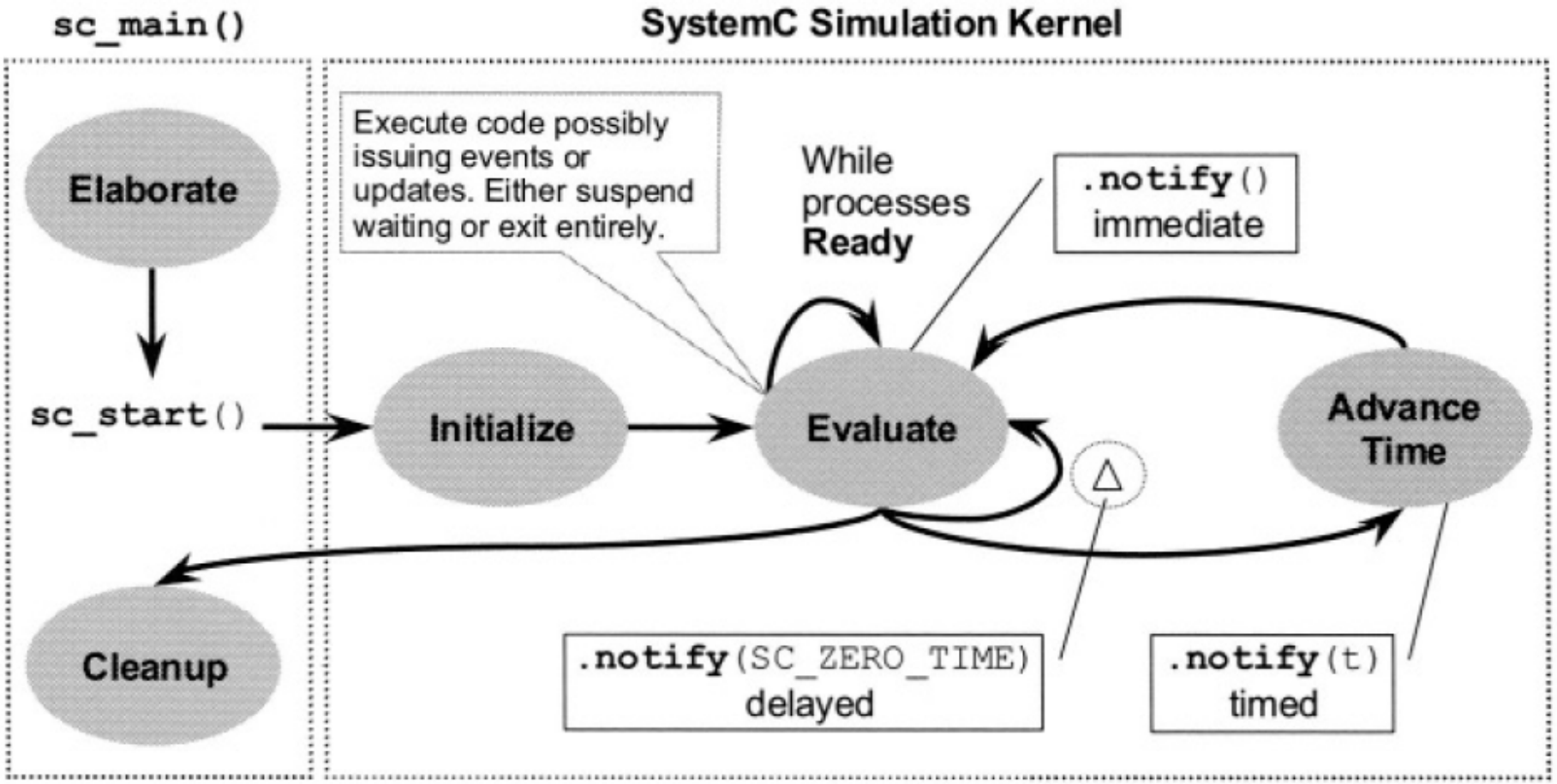
```
sc_event action;
sc_time now(sc_time_stamp()); //observe current time
//immediately cause action to fire
action.notify();
//schedule new action for 20 ms from now
action.notify(20, SC_MS);
//reschedule action for 1.5 ns from now
action.notify(1.5,SC_NS);
//useless, redundant
action.notify(1.5, SC_NS);
//useless preempted by event at 1.5 ns
action.notify(3.0,SC_NS);
//reschedule action for next delta cycle
action.notify(SC_ZERO_TIME);
//useless, preempted by action event at SC_ZERO_TIME
action.notify(1, SC_SEC);
//cancel action entirely
action.cancel();
//schedule new action for 1 femtosecond from now
action.notify(20,SC_FS);
```

sc_event queue

- sc_event_queue lets a single event be scheduled multiple times even for the same time
- when events are scheduled for the same time, each happens in a separate delta cycle
- sc_event_queue objects do not support immediate notification
- .cancel() is replaced with .cancel_all()

```
sc_event_queue  action;
sc_time now(sc_time_stamp()); //observe current time
action.notify(20, SC_MS);//schedule for 20 ms from now
action.notify(1.5,SC_NS);//another for 1.5 ns from
                         //now
action.notify(1.5,SC_NS);//another identical action
action.notify(3.0,SC_NS);//another for 3.0 ns from
                          //now
action.notify(SC_ZERO_TIME);//for next delta cycle
action.notify(1,SC_SEC);//for 1 sec from now
action.cancel_all(); // cancel all actions entirely
```

SystemC simulation kernel



Static Sensitivity

- Static sensitivity provides the parameters, which would trigger a process **statically**
  - Static sensitivity is specified during design.
- For example, change of any of **A, B** and **Cin** in 1-bit Adder are events that trigger **add**() process in 1-bit Adder.
  - For static sensitivity, the process registration needs:

```
SC_METHOD(add);
sensitive << A << B << Cin;
```

- An example demonstrating static sensitivity follows.

Static sensitivity: Example 1

```
// rcv.h

#include "systemc.h"
#include "frame.h"

SC_MODULE(rcv) {
  sc_in<frame_type> xin;
  sc_out<int>       id;

  void extract_id();

  SC_CTOR(rcv) {
    SC_METHOD(extract_id);
    sensitive(xin);
  }
};
```

```
// rcv.cc

#include "rcv.h"
#include "frame.h"

void rcv::extract_id() {
  frame_type frame;

  frame = xin;
  if(frame.type == 1) {
    id = frame.ida;
  } else {
    id = frame.idb;
  }
}
```

Example2

```
// traff.h

#include "systemc.h"

SC_MODULE(traff) {

  // input ports
  sc_in<bool>   roadsensor;
  sc_in<bool>   clock;

  // output ports
  sc_out<bool>  NSred;
  sc_out<bool>  NSyellow;
  sc_out<bool>  NSgreen;
  sc_out<bool>  EWred;
  sc_out<bool>  EWyellow;
  sc_out<bool>  EWgreen;

  void control_lights();
  int i;

  // Constructor
  SC_CTOR(traff) {

    SC_THREAD(control_lights);  // Thread Process


    sensitive << roadsensor;
    sensitive_pos << clock;
  }
};
```

```
// traff.cc

#include "traff.h"

void traff::control_lights() {
  NSred = false;
  NSyellow = false;
  NSgreen = true;
  EWred = true;
  EWyellow = false;
  EWgreen = false;

  while (true) {
    while (roadsensor == false)
      wait();
    NSgreen = false; // road sensor triggered
    NSyellow = true; // set NS to yellow
    NSred = false;
    for (i=0; i<5; i++)
      wait();

    NSgreen = false;   // yellow interval over
    NSyellow = false;  // set NS to red
    NSred = true;      // set EW to green
    EWgreen = true;
    EWyellow = false;
    EWred = false;
    for (i= 0; i<50; i++)
      wait();
    ...
```

Dynamic sensitivity for SC_METHOD: next_trigger()

- *next_trigger()* does **not** suspend the process
- temporarily sets a sensitivity list only for next time the process executes again
- next_trigger() may be called repeatedly, and each invocation encountered overrides the previous
- Without a next_trigger() or a static sensitivity, such process will never be executed again

```
next_trigger(time);
next_trigger(event);
next_trigger(event1 | eventi…); //any of these
next_trigger(event1 & eventi…); //all of these
                                //required
next_trigger(timeout, event);   //event with timeout
next_trigger(timeout, event1 | eventi…); //any + timeout
next_trigger(timeout, event1 & eventi…); //all + timeout
next_trigger(); //re-establish static sensitivity
```

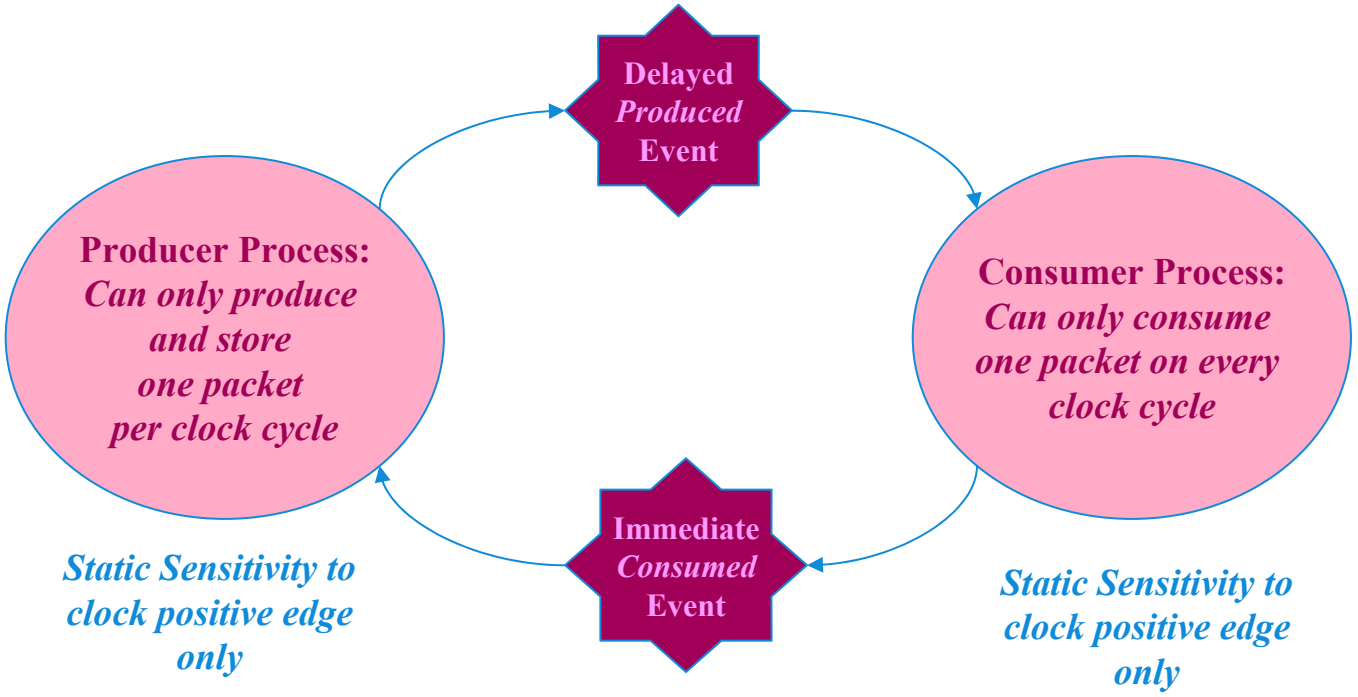
Dynamic sensitivity for SC_THREAD::wait()

- These processes rely on *wait()* method to suspend their execution
- When the suspended process is reactivated, it resumes execution at the statement after *wait()*

```
wait(time);
wait(event);
wait(event_1 | event_n...);  // any of these
wait(event_1 & event_n...);  // all of these
wait(timeout, event);   // event with timeout
wait(timeout, event_1 | event_n...);  //any event with
                                      // timeout
wait(timeout,event_1 & event_n...);  //all events with
                                     // timeout
wait(); // static sensitivity
```

Dynamic Sensitivity

- Using events process can be made dynamically sensitive.
- Look at the example:



Dynamic Sensitivity: ***A Producer Consumer Example***

```
#include "systemc.h"
SC_MODULE (Dynamic) {
    sc_in_clk clk; sc_out<int> outport;
    sc_event p_event, c_event; int pkt;

    int produce_packet(){
        return ++pkt;
    }
    void consume_packet(int packet){
       cout << "@ " << sc_time_stamp() << ":
          Packet written out: " << packet << endl;
       outport.write(packet);
    }
    void produce() {
        while (true) {
            pkt = produce_packet();
            cout << "@ " << sc_time_stamp() << ":
                Packet produced: " << pkt << endl;
            p_event.notify(5, SC_NS);
            wait();
            wait(c_event);
        }
    }
```

```
void consume() {
        while (true) {
            wait(p_event);
            consume_packet(pkt);
            cout << "@ " << sc_time_stamp() << ":
                Packet consumed: " << pkt << endl;
            wait(3);
            c_event.notify();
        }
    }

    SC_CTOR(Dynamic) {
        SC_THREAD(produce);//Static
        sensitive << clk.pos();
        SC_THREAD(consume);
        sensitive << clk.pos();//Static

        pkt = 0;
    }
};
```

5.

Interfaces, channels and ports

Introduction

- In classical hardware modeling, the hardware signal is the medium for communication and synchronization between processes
  - Signals are at a very low level of abstraction
- Interfaces, ports and channels are the way through which SystemC implements synchronization and communication at system level
  - Interfaces: they define a set of methods, but do not implement these methods
  - Channels: they implement one or more interfaces
  - Ports: they enable a module, and hence its processes, accessing a channel's interface

Interfaces

- An interface provides a set of method declarations, but provides no method implementations
  - Interfaces are abstract classes
  - All interface methods are pure virtual methods, i.e. they do not have an implementation
- Interfaces are used to declare sets of methods that channels must implement
- Use of interfaces is useful when modeling layered communication
- All interfaces are directly or indirectly derived from the base class sc_interface

Interface examples

```
// ----------------------------------------
//  An example read interface: sc_read_if
//  this interface provides a 'read' method
// ----------------------------------------

template <class T>
class sc_read_if
: virtual public sc_interface
{
public:

  // interface methods
  virtual const T& read() const = 0;
};
```

1

```
// ----------------------------------------
//  An example write interface: sc_write_if
//  this interface provides a 'write' method
// ----------------------------------------

template <class T>
class sc_write_if
: virtual public sc_interface
{
public:

  // interface methods
  virtual void write( const T& ) = 0;
};
```

2

```
//  An example read/write interface: sc_read_write_if
// ----------------------------------------------------

template <class T>
class sc_read_write_if
: public sc_read_if<T>,
  public sc_write_if<T>
{};
```

3

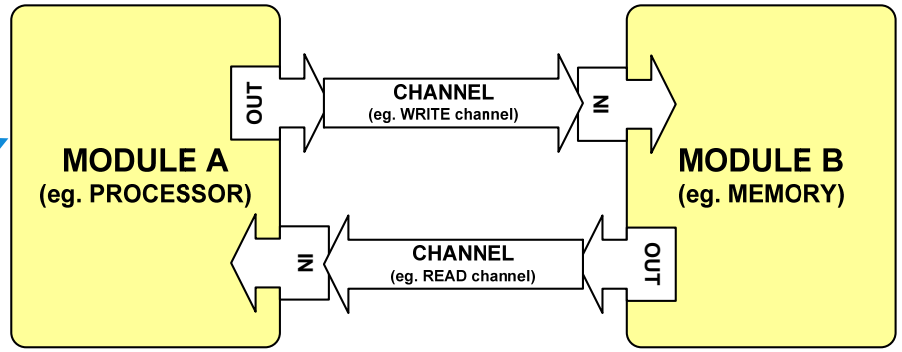
1-2: they define pure virtual methods
1-2: read & write interface derive from the interface base class sc_interface
3: interface derived from 1-2

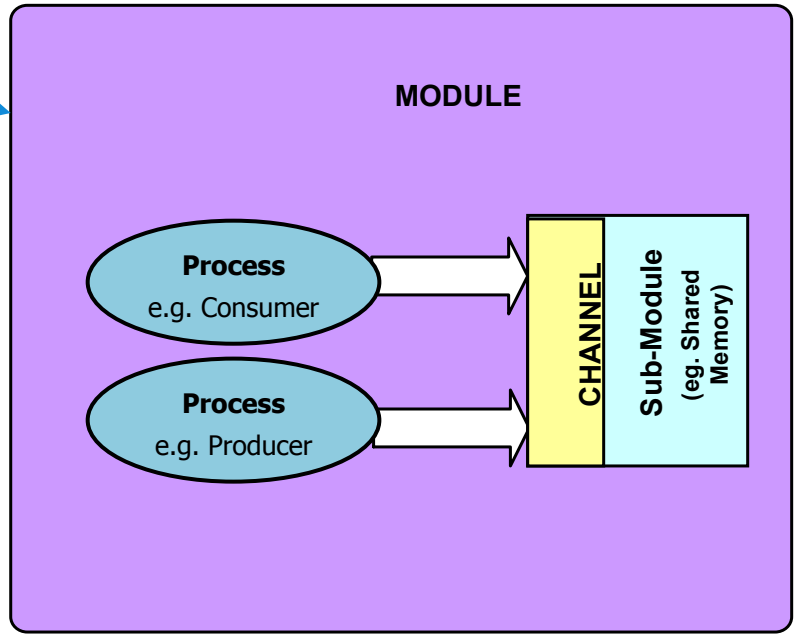
Channels

- Channels provide communication for
  - inter- or
  - intra-module

  data transfer.

- Channels encapsulate
  - communication behaviour
  - access
  - associated buffering





Channels

- A channel implements one or more interfaces, and serves as a container for communication functionality
- Different channels may implement the same interface in different ways
- A channel may implement more than one interface
- A channel may be connected to more than two modules
- A distinction is made between *primitive channels* and *hierarchical channels*

Primitive channels

- A channel is primitive when it does not contain any hierarchy or process
- All primitive channels are derived from the base class called *sc_prim_channel*
- SystemC contains several built-in channels.

  The simplest are: **sc_signal**, **sc_fifo, sc_mutex**, and **sc_semaphore**

- A primitive channel is one that supports the *evaluate-update* method of access:
  - evaluate-update method is designed for simulating concurrency
  - When simultaneous actions occur on the channel (ex. concurrent *write & read* from a channel), they have to be serialized in reality. However, updates to the channel will take effect only after all the currently active processes are finished or have come to a synchronization point

sc_signal

- It is the most classical and simplest channel

```
// ---------------------------------
//  INTERFACE : sc_signal_in_if<T>
// ---------------------------------

template <class T>
class sc_signal_in_if
: virtual public sc_interface
{
public:

    // read the current value
    virtual const T& read() const = 0;
};
```

```
// ------------------------------------
//  INTERFACE : sc_signal_inout_if<T>
// ------------------------------------

template <class T>
class sc_signal_inout_if
: public sc_signal_in_if<T>
{
public:

    // write the new value
    virtual void write( const T& ) = 0;
};
```

sc_signal (cont'ed)

```
// ---------------------------------
//  PRIMITIVE CHANNEL : sc_signal<T>
// ---------------------------------

template <class T>
class sc_signal
: public sc_prim_channel,
  public sc_signal_inout_if<T>
{
public:


// interface methods

virtual const T& read() const
    { return m_cur_val; }

virtual const sc_event& default_event() const
    { return m_value_changed_event; }

virtual void write( const T& value )
{
    // dynamic design rule checking
    if( current_process() != m_driver ) {
        if( m_driver == 0 )
            m_driver = current_process();
        else
            error( "more than one driver\n" );
    }
    // do the write
    m_new_val = value;
    if( m_new_val != m_cur_val )
        request_update();
}
```

```
// ---------
//  EXAMPLE
// ---------

SC_MODULE( writer )
{
    // port(s)
    sc_out<int> out;
    sc_in_clk   clk;

    // process(es)
    void main_action()
    {
        int val = 0;
        while( true ) {
            wait();       // wait for a positive edge of the clock
            out = val ++; // shortcut for: out->write( val ++ );
        }
    }

    SC_CTOR( writer )
    {
        SC_THREAD( main_action );
        sensitive_pos << clk;
    }
};
```

sc_signal (cont'ed)

```
SC_MODULE( reader )
{
    // port(s)
    sc_in<int> in;

    // process(es)
    void main_action()
    {
        int val;
        while( true ) {
            wait();   // wait for a change of value on in
            val = in; // shortcut for: val = in->read();
            cout << val << endl;
        }
    }

    SC_CTOR( reader )
    {
        SC_THREAD( main_action );
        sensitive << in; // shortcut for: sensitive << in.default_event()
    }
};
```

```
int sc_main( int, char*[] )
{
    // declare channel(s)
    sc_signal<int> sig;

    // create clock(s)
    sc_clock clock( "clock", 10, SC_NS );

    // instantiate block(s) and connect to channel(s) and clock(s)
    writer w( "writer" );
    reader r( "reader" );

    w.out( sig );
    r.in( sig );

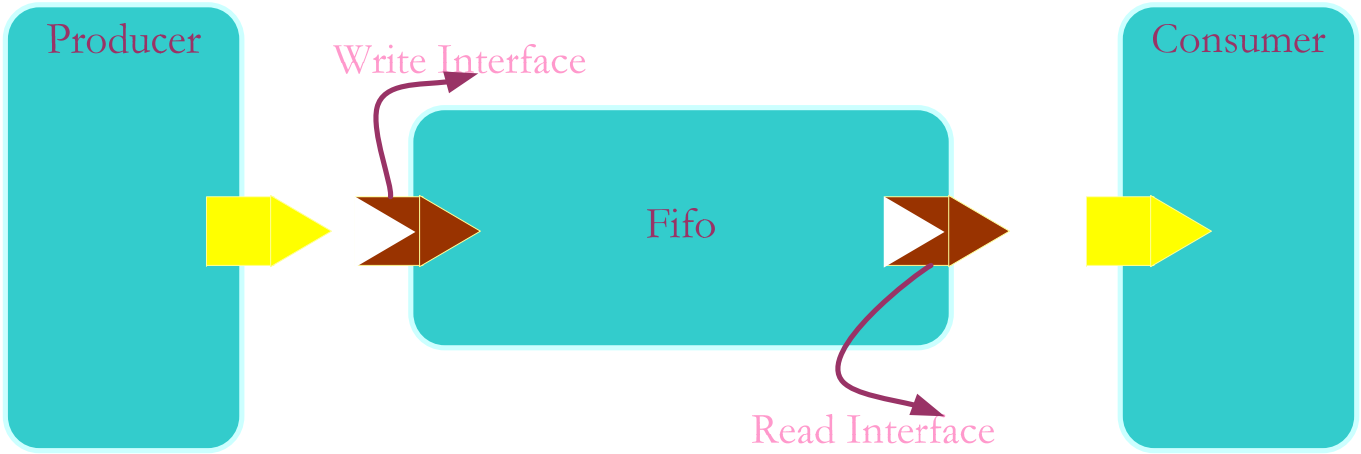
    w.clk( clock );

    // run the simulation for a while
    sc_start( 1000, SC_NS );
    return 0;
}
```

SC_FIFO Channels

- **sc_fifo** is a predefined SystemC primitive channel.
  - Default object created with this type has size of 16.
- **sc_fifo** has following pre-defined methods:
  - **write()** — *if not full, write to buffer; if full, wait for availability.*
  - ***nb_write()*** — *return true, when fifo has space; returns false (does not wait).*
  - **read()** — *if not empty, read; if empty, wait until available and then read.*
  - ***nb_read()*** — *returns false, when fifo is empty (does not wait) otherwise reads and returns true*
  - **num_available()** — *returns number of elements.*

Fifo Example



FIFO: Declaration of Interfaces

```
class write_if : public sc_interface
{
  public:
    virtual void write(char) = 0;
    virtual void reset() = 0;
};

class read_if : public sc_interface
{
  public:
    virtual void read(char&) = 0;
    virtual int num_available() = 0;
};
```

FIFO: Declaration of Channel

```
class fifo: public
sc_channel,
  public write_if,
  public read_if
{
  private:
    int max_elements=10;
    char
data[max_elements];
    int num_elements,
first;
    sc_event write_event,
             read_event;
    bool fifo_empty() {…};
    bool fifo_full() {…};
  public:
    fifo() :
num_elements(0),
             first(0);
```

```
void write(char c)
{
  if (fifo_full())
    wait(read_event);
  data[ <you calculate> ]
= c;
  ++num_elements;
  write_event.notify();
}

void read(char &c)
{
  if (fifo_empty())
    wait(write_event);
  c = data[first];
  --num_elements;
  first = ...;
  read_event.notify();
}
```

FIFO: Declaration of Channel

```
void reset()
{

num_elements = first = 0;

}

int num_available()
{

return num_elements;

}

}; // end of class
declarations
```

FIFO: Producer & Consumer

```
SC_MODULE(producer)
{
  public:
    sc_port<write_if> out;

  SC_CTOR(producer)
{
    SC_THREAD(main);
}

  void main()
  {
    char c;
    while (true)
    {
      out.write(c);
      if(…)
      out.reset();
    }
  }
};
```

```
SC_MODULE(consumer)
{
  public:
    sc_port<read_if> in;

    SC_CTOR(consumer)
  {
    SC_THREAD(main);
  }

  void main()
  {
    char c;
    while (true)
    {
      in.read(c);
      cout<<
in.num_available();
    }
  }

};
```

FIFO Completed

```
SC_MODULE(top)
{
  public:
    fifo afifo;
    producer *pproducer;
    consumer *pconsumer;
  SC_CTOR(top)
  {
    pproducer=new
producer("Producer");
    pproducer->out(afifo);

    pconsumer=new
consumer("Consumer");
    pconsumer->in(afifo);
  }
};
```

sc_mutex

- A mutex is an object used to let multiple program threads share a common resource without colliding
- Any process that needs the resource must lock the mutex to get exclusive access to the resource. The process unlocks the mutex when the resource is available again
- An example of application is arbitration for a shared bus
- Drawback: lack of an event that tells when an sc_mutex is freed. This requires using **trylock** repeatedly based on some other event or time-based delay

sc_mutex

- **sc_mutex** is a primitive channel intended to model the behavior of a mutual exclusion lock to control access to a shared resource
- **sc_mutex** has two states: *unlocked* or *locked*. Only one process can lock a given mutex at one time
- **sc_mutex** methods:
  - ***int lock()*** : *Lock the mutex if it is free, else **wait** till mutex gets free.*
  - ***int unlock()*** : *Unlock the mutex, if lock is in. Else do nothing*
  - ***int trylock()*** : *Check if mutex is free, if free then lock it else return -1. **Does not wait indefinitely***

sc_semaphore

- It is a channel used to handle the case when there can be more copies or owners for one resource
- A mutex can be considered a semaphore with a count of one
- When creating an sc_semaphore object, it is necessary to specify how many are available
- An *sc_semaphore* access consists of waiting for an available resource and then posting notice when finished with the resource (compare to sc_mutex)

Hierarchical channels

- Hierarchical channels can have a structure, contain processes and access directly other channels
- All hierarchical channels are derived from the base class called **sc_channel**
- As sc_channel is just a redefinition of the class sc_module, so from a language point of view a hierarchical channel is nothing but a module

Primitive vs hierarchical channels

- Use primitive channels:
  - When you need to use the request-update scheme.
  - When channels are atomic and cannot reasonably by chopped into smaller pieces.
  - When speed is absolutely crucial (using primitive channels we can often reduce the number of delta cycles).
  - When it doesn't make any sense trying to build up a channel (such as a semaphore or a mutex) out of processes and other channels.
- Use hierarchical channels:
  - When channels are truly hierarchical and users would want to be able to explore the underlying structure.
  - When channels contain processes.
  - When channels contain other channels.
- Both primitive and hierarchical channels can be refined. Refinement has to do with interfaces, not with whether a channel is primitive or hierarchical.

Ports

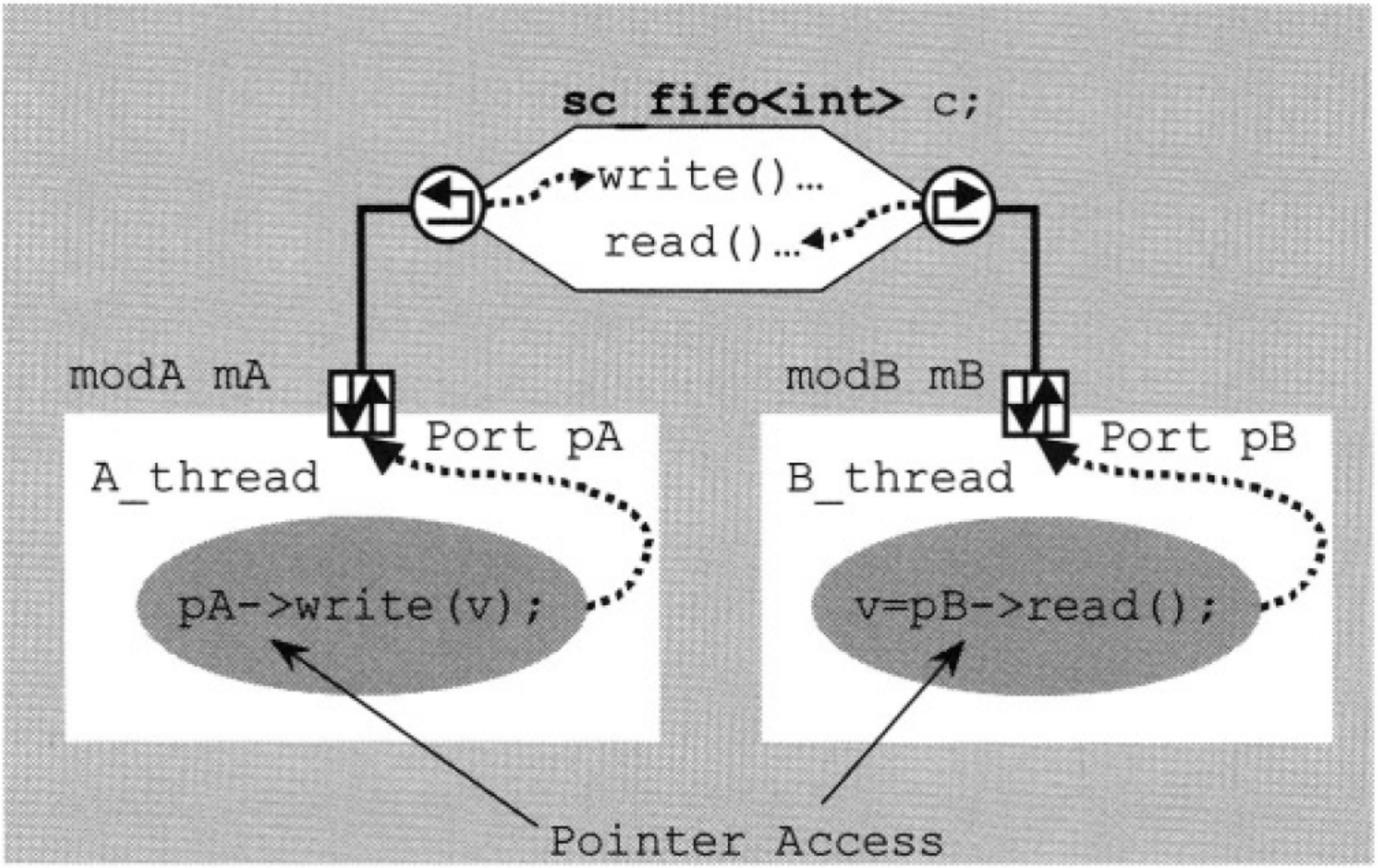
- A port is an object through which a module, and hence its processes, can access a channel's interface
- A port type assumes a certain interface. A channel cannot be connected to a port if it doesn't implement the port's interface

```
sc_port<interface<type>, N > p;
```

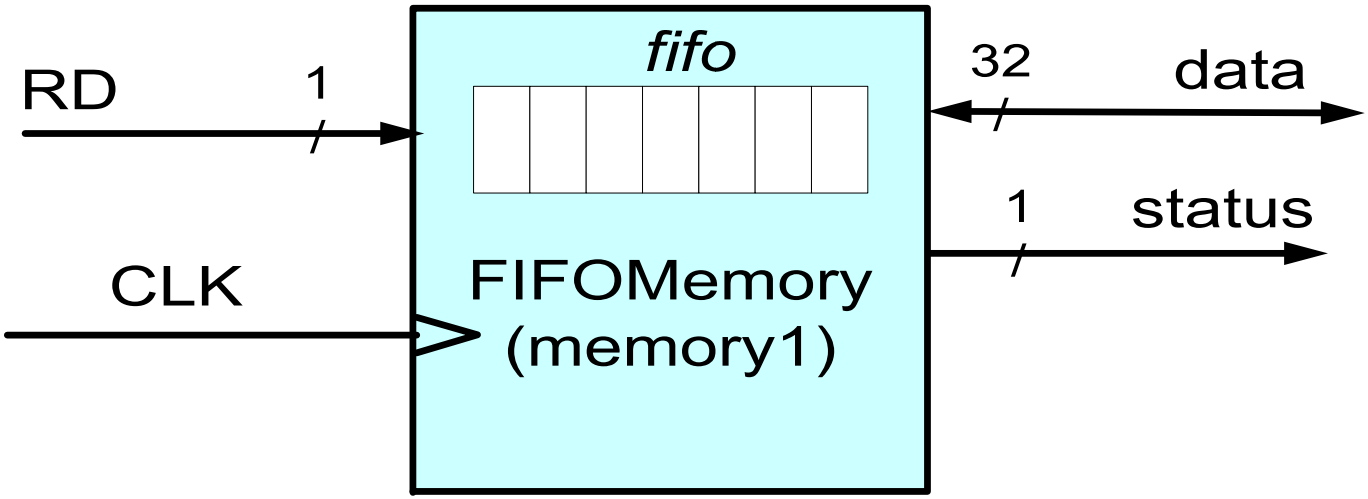
  - N = number of channels that can be connected to the port

- The port base class is called *sc_port*
- Specialized ports can be created by refining the port base class sc_port or one of the predefined port types

Communication via *sc_port*



Primitive Channel Example: A FIFO Memory



FIFO M emories are used for

- **delay or**
- **buffering purposes**

Primitive Channel Example: A FIFO Memory

```
#include "systemc.h"

SC_MODULE(FIFOMemory) {
  sc_in<bool> RD;   // ReadNotWrite
  sc_inout<int> data;
  sc_in_clk clock;
  sc_out<bool> status; // 0 - NOERROR, 1 -
    ERROR
  sc_fifo<int> buffer;
  const int size;
  void memory_process() {
      int temp;
      while(true) {
            wait();
            if (RD.read() == true) {
                if(buffer.nb_read(temp)) {
                   status.write(0);
                   data.write(temp);
                }
                else{
                   status.write(1);
                }
            }
            else {
                temp = data.read();
                if(buffer.nb_write(temp)) {
                   status.write(0);
                }
                else {
                    status.write(1);
                }
            }
      }
  }

  SC_HAS_PROCESS(FIFOMemory);

   FIFOMemory(sc_module_name name_, int size_=10):
                sc_module(name_), size(size_), buffer(size_)
   {
      SC_THREAD(memory_process);
      sensitive << clock.pos();
    }
};
```

Testbench

```
#include "systemc.h"
SC_MODULE(Testbench){
  sc_in_clk clock;    sc_inout<int> data;
  sc_out<bool> RD;    sc_in<bool> status;

  void testprocess(){
    wait();
    data.write(0x010);
    RD.write(0);
    wait();
    cout << "@: " << sc_time_stamp() << ":
  WR " << data.read() << ": STATUS: " <<
  status.read() << endl;

    data.write(0x020);
    RD.write(0);
    wait();
    cout << "@: " << sc_time_stamp() << ":
  WR " << data.read() << ": STATUS: " <<
  status.read() << endl;
    //also write 0x030, 0x040, 0x050, 0x060
  not shown
    RD.write(1);
    wait();
    cout << "@: " << sc_time_stamp() << ":
  RD " << data.read() << ": STATUS: " <<
  status.read() << endl;

    //also read four more times
  }

  SC_CTOR(Testbench){
    SC_THREAD(testprocess);
    sensitive << clock.pos();
  }
};
```

Top-level Module

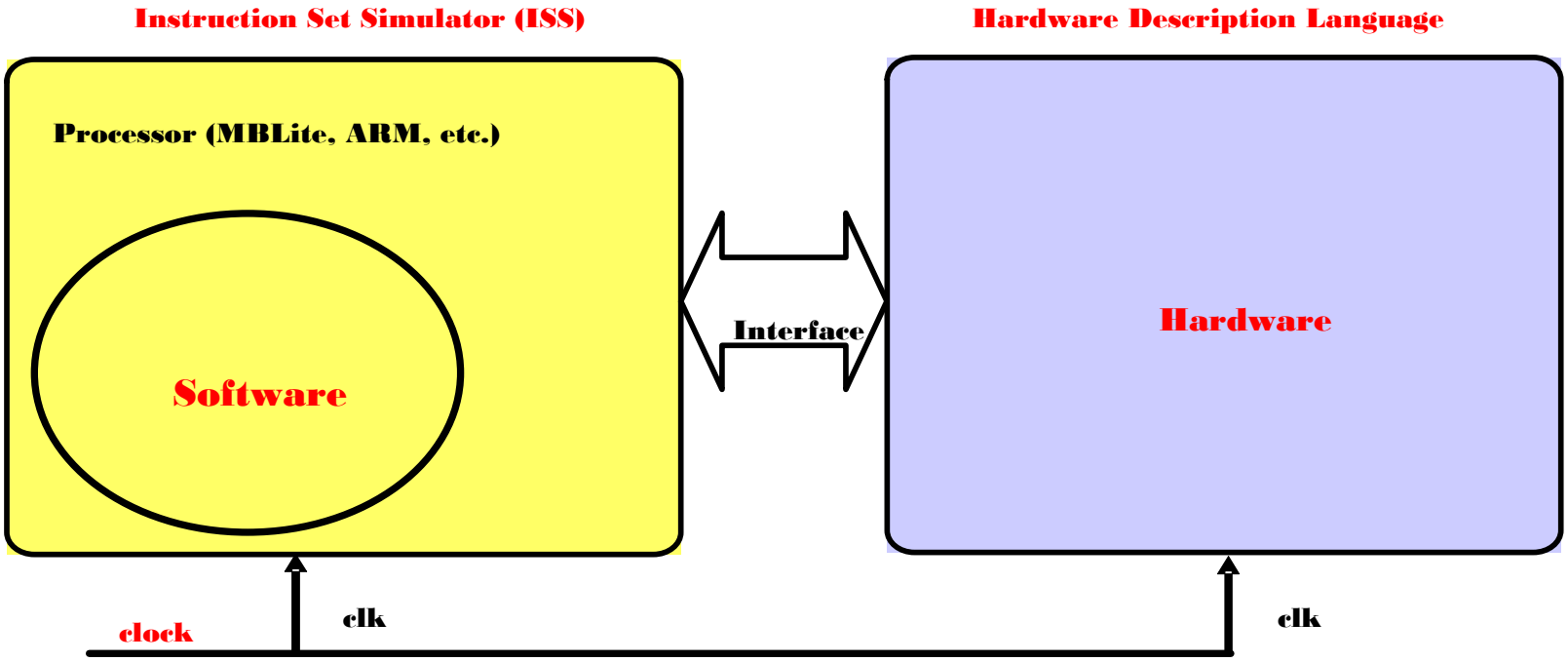
```
#include "FIFOMemory.h"
#include "Testbench.h"

Int sc_main(int argc, char *argv[]){
   sc_signal<int> data;
   sc_signal<bool> RD, status;
   sc_clock sysclock("clock1", 20, SC_NS);
   FIFOMemory memory1("Memory1");
   Testbench test1("Test1");

   memory1.clock(sysclock);
   memory1.data(data);
   memory1.RD(RD);
   memory1.status(status);
   test1.clock(sysclock);
   test1.data(data);
   test1.RD(RD);
   test1.status(status);
}
```

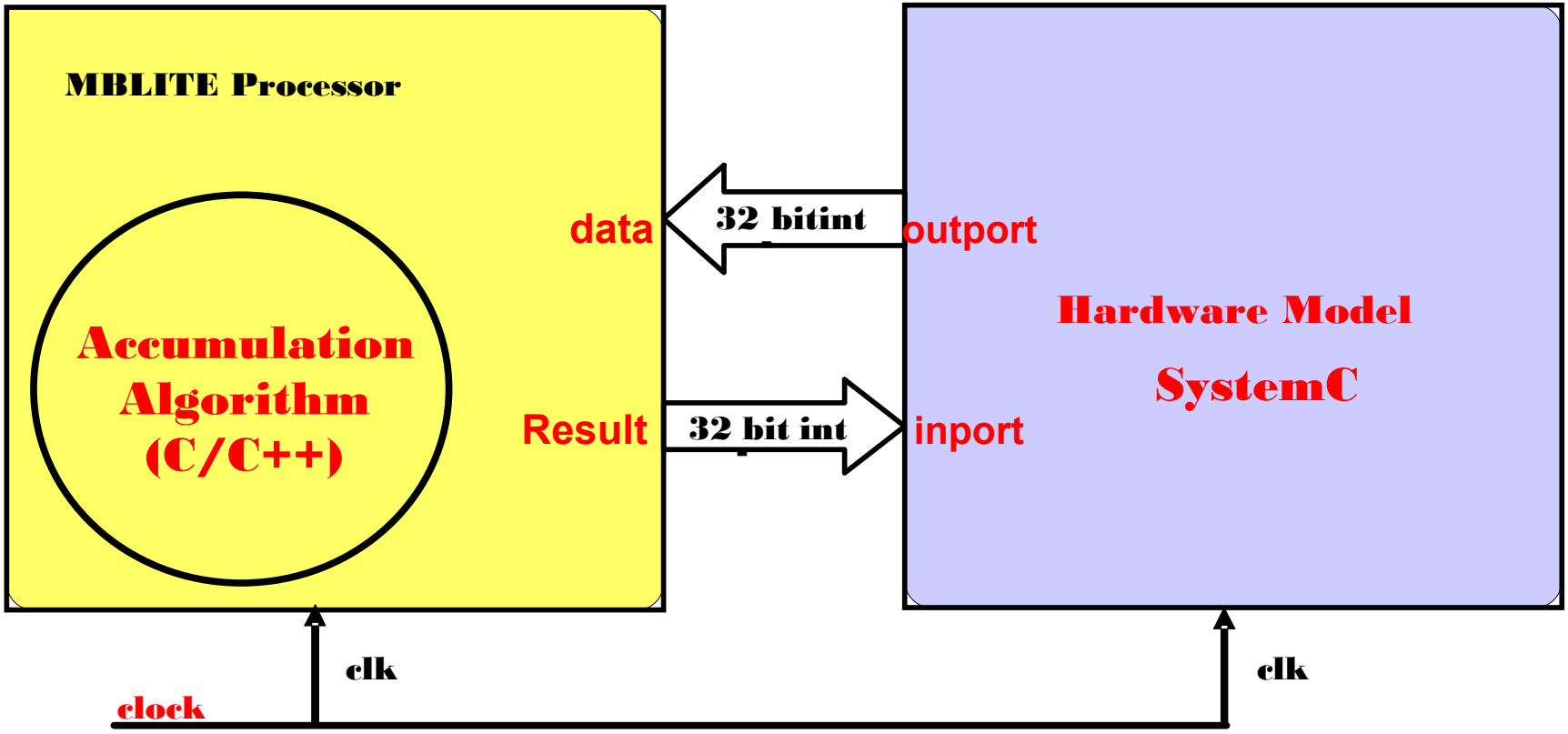
Interconnections between *memory1* and *test1*

HW/SW Model

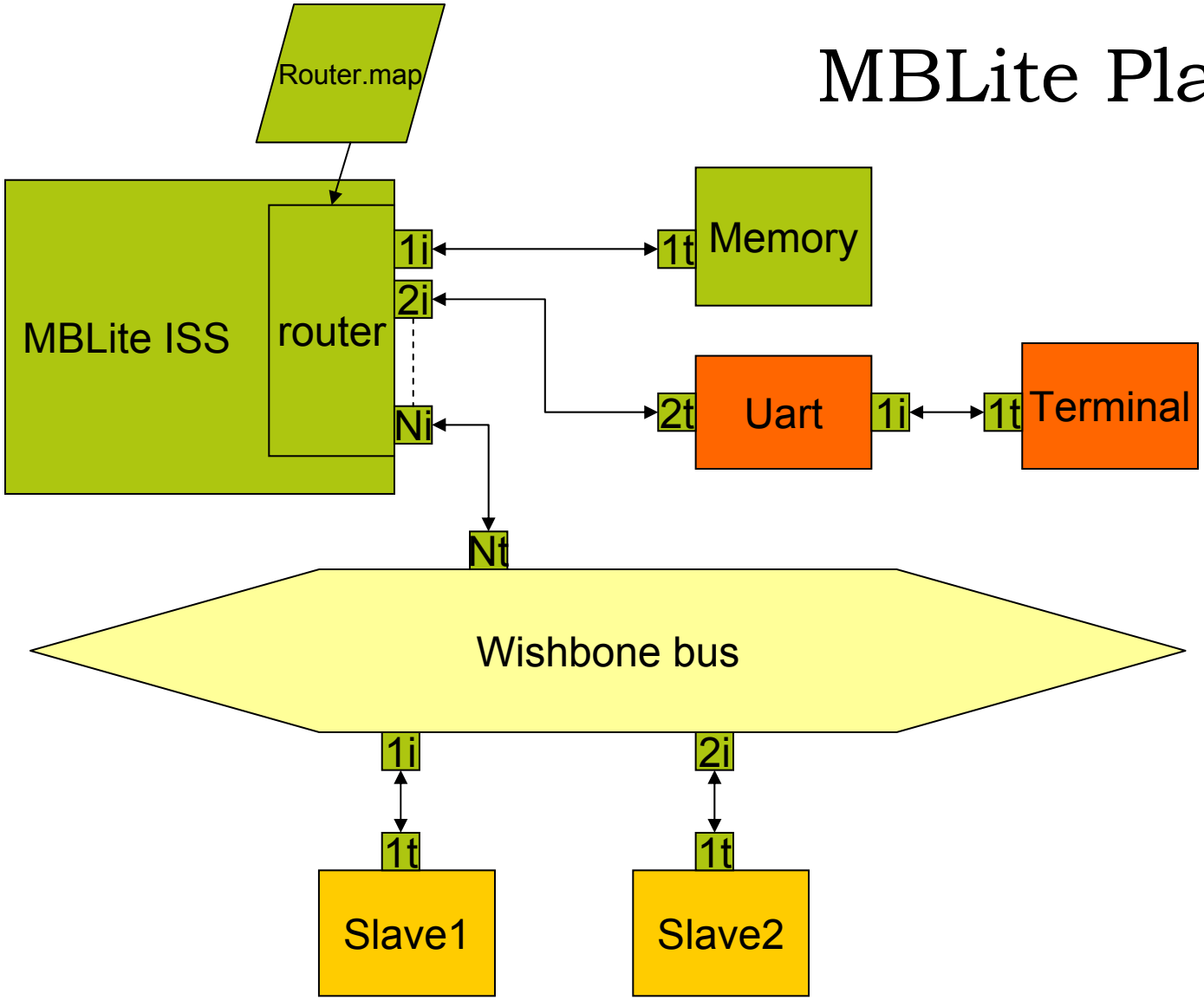

Instruction Set Simulator (ISS)
Hardware Description Language
Processor (MBLite, ARM, etc.)
Software
Interface
Hardware
clock
clk
clk

A HW/SW co-design example

The HW block is supplying 10 numbers each 32-bit wide, the SW block is performing the accumulation of the numbers supplied and sent back to the HW



MBLite Platform

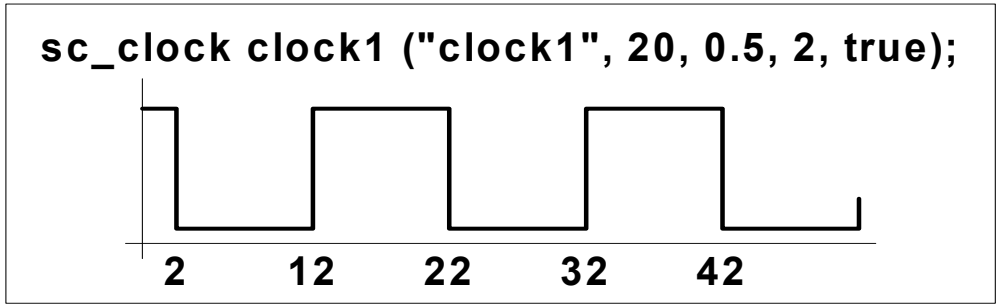


Some considerations on clock

- Clocks are commonly used when modelling low-level hardware where clocked logic design currently dominates
- Clocks add many events, and much resulting simulation activity is required to update those events -> clocks can slow down simulation significantly
- SystemC provides a built-in hierarchical channel called **sc_clock**
- How to create ?

  sc_clock ***clock_name*** ("***clock_label***", ***period***
  [, duty_ratio, offset, initial_value]);

- Example



Debugging and signal tracing

- Signal tracing can be done in SystemC by using a so called VCD (Value Change Dump) format file
- A VCD file is a text format file containing the activity of the signals which need to be traced
- A VCD file is created in 4 steps:
  - Open the VCD file
  - Select the signals to be traced
  - Start simulation: VCD file is written
  - Close the trace file

```
sc_trace_file* tracefile;
tracefile = sc_create_vcd_trace_file(tracefile_name);
if (!tracefile) cout <<"There was an error."<<endl;
…
sc_trace(tracefile,signal_name,"signal_name");
…
sc_start(); // data is collected
…
sc_close_vcd_trace_file(tracefile);
```

Example

```
//FILE: wave.h
SC_MODULE(wave) {
  sc_signal<bool> brake;
  sc_trace_file*  tracefile;
  …
  double temperature;
};
```

```
//FILE: wave.cpp
wave::wave(sc_module_name nm) //Constructor
: sc_module(nm) {
  …
  tracefile = sc_create_vcd_trace_file("wave");
  sc_trace(tracefile,brake,"brake") ;
  sc_trace(tracefile,temperature,"temperature");
}//endconstructor
wave::~wave() {
    sc_close_vcd_trace_file(tracefile);
    cout << "Created wave.vcd" << endl;
}
```

SystemC Community and Tools

- SystemC is an **open source** and the latest versions can be downloaded free from **www.systemc.org**
- SystemC can be installed in various platforms, including **ModelSim**, VC++, Linux G++, Cygwin etc. SystemC community at www.systemc.org discusses various issues with different tools and operating systems.
- SystemC designs can be **synthesized to gate level HDLs** using various synthesis tools, examples **Celoxica Agility Compiler**, Forte Cynthesizer , Mentor Catapult C++, Cadence CtoS etc.
- SystemC can be debugged using **gdb** and other C++ debugging tools. Waveform based debugging tools are also available in some IDEs (eg. ModelSim).

References

- OSCI SystemC Developers, SystemC User Guide, 2005. Website, www.systemc.org
- Black, D.C., Donovan, J., "SystemC : from the Ground Up", Kluwer Academics, 2004.
- Website, www.doulos.com
- Website, www.forteds.com
- C++ tutorial: www.cplusplus.com/doc/tutorial