SystemC: an overview

ET 4351

Alexander de Graaf, EEMCS/ME/CAS

5/9/14

Acknowledgement

- These slides are based upon:

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. *C++ Basics*
3. *Modules and hierarchy*
4. *Processes, Events, Time*
5. *Types*
6. *Channels, Interfaces and ports*
7. *Tools (simulation, synthesis)*

1.

Introduction

Motivations

- 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

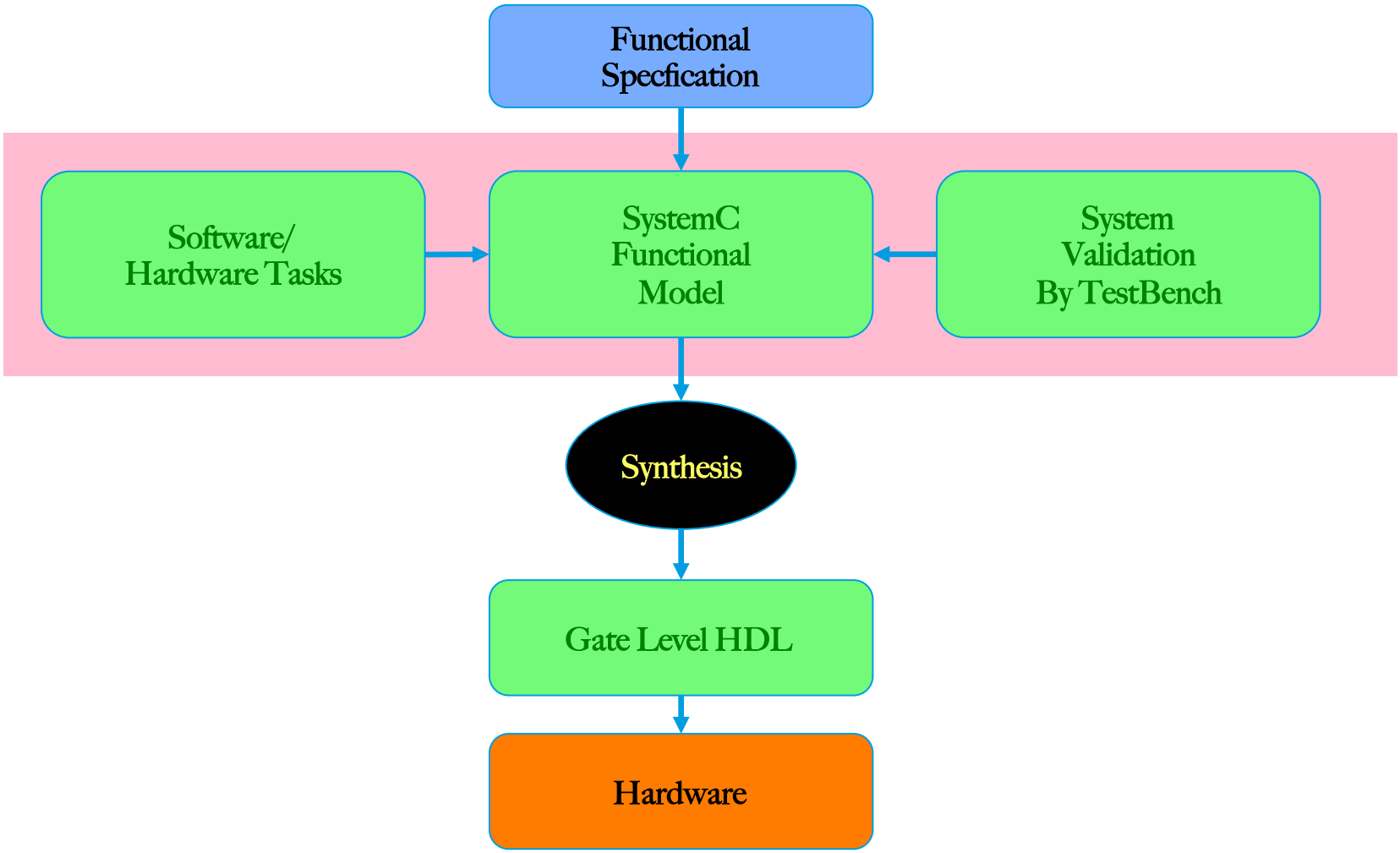
What is SystemC..?

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

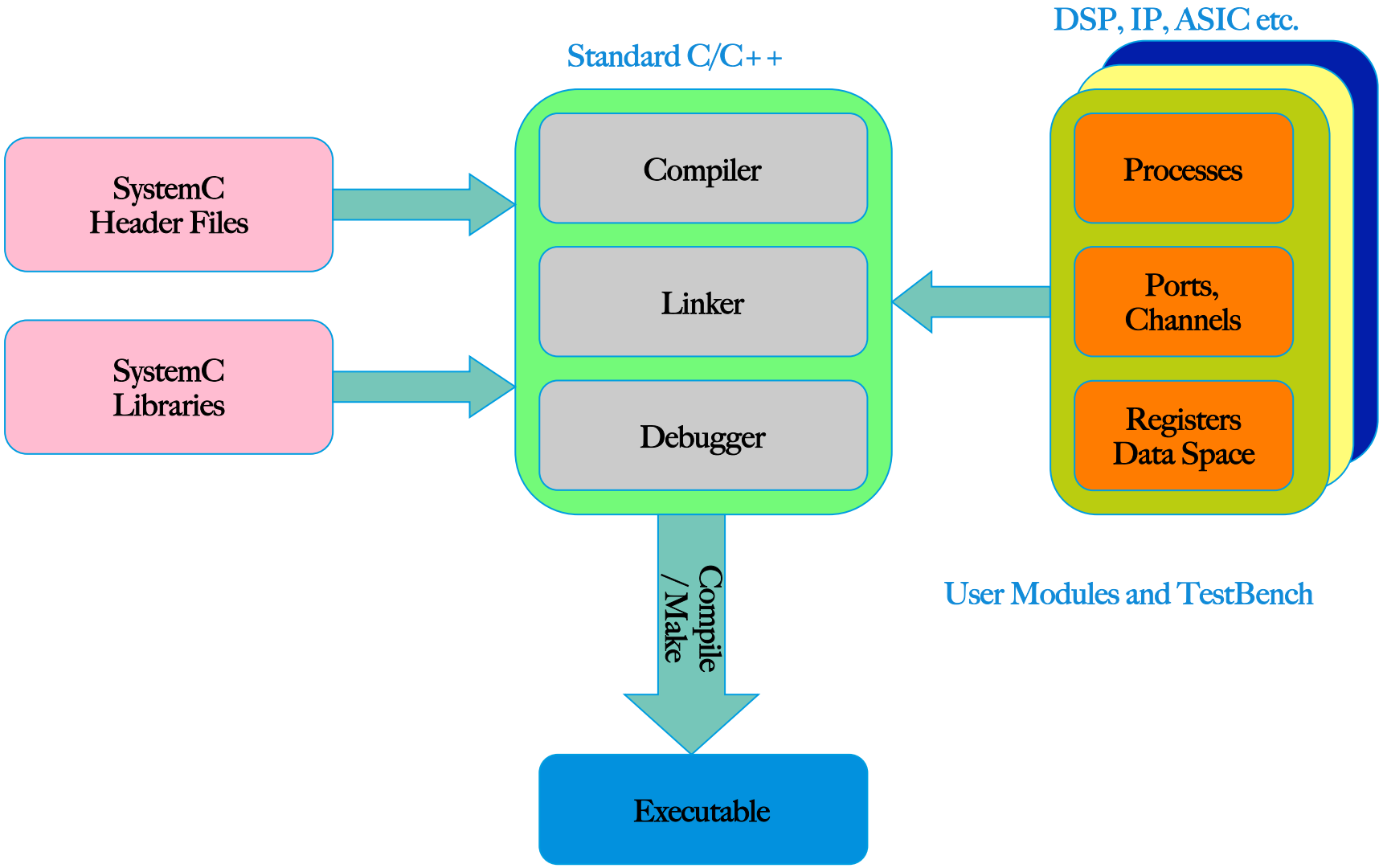
What is SystemC?

- 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

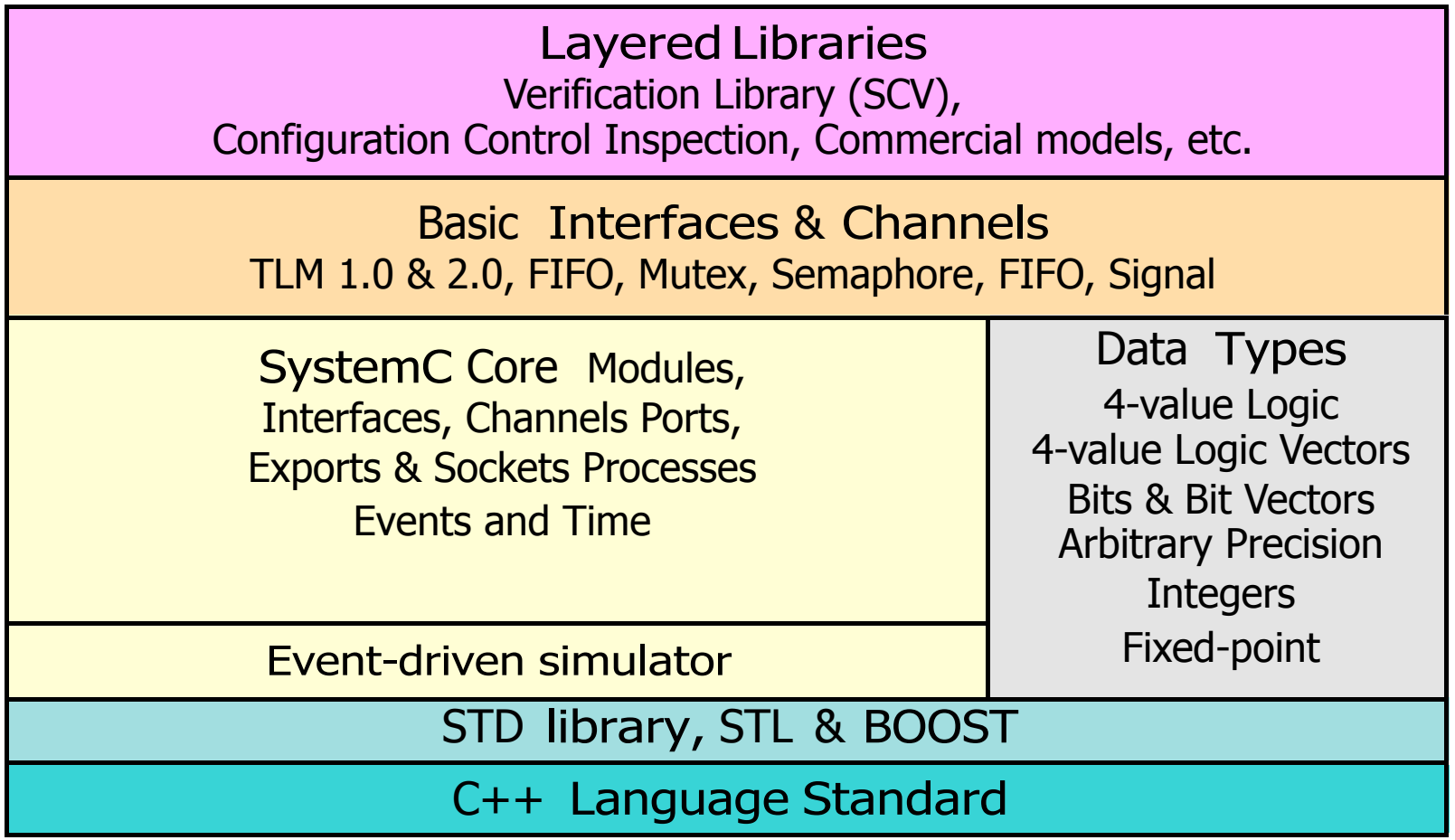
SystemC Design Flow



How SystemC works..?



SystemC Architecture



2.

C++ Basics

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

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

Creating an instance

Stack allocation

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

Heap allocation

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

...

delete imagePtr;
```

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] = img->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);
```

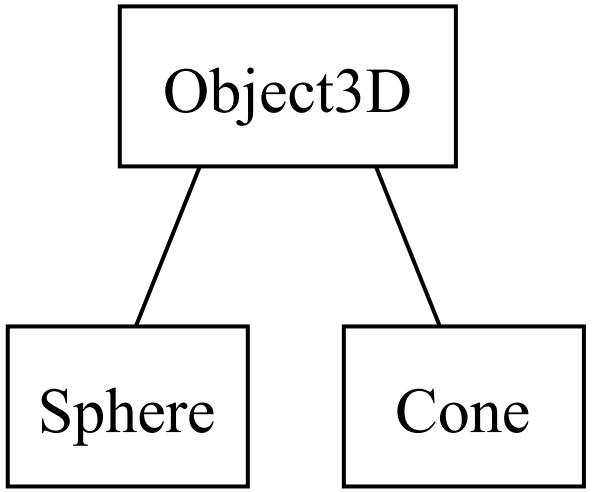
or

```
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);
```

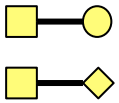
Actually calls `Sphere::intersect`

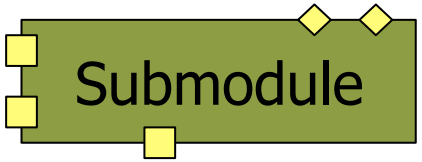
3.

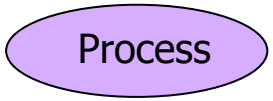
Modules and hierarchy

SystemC Components

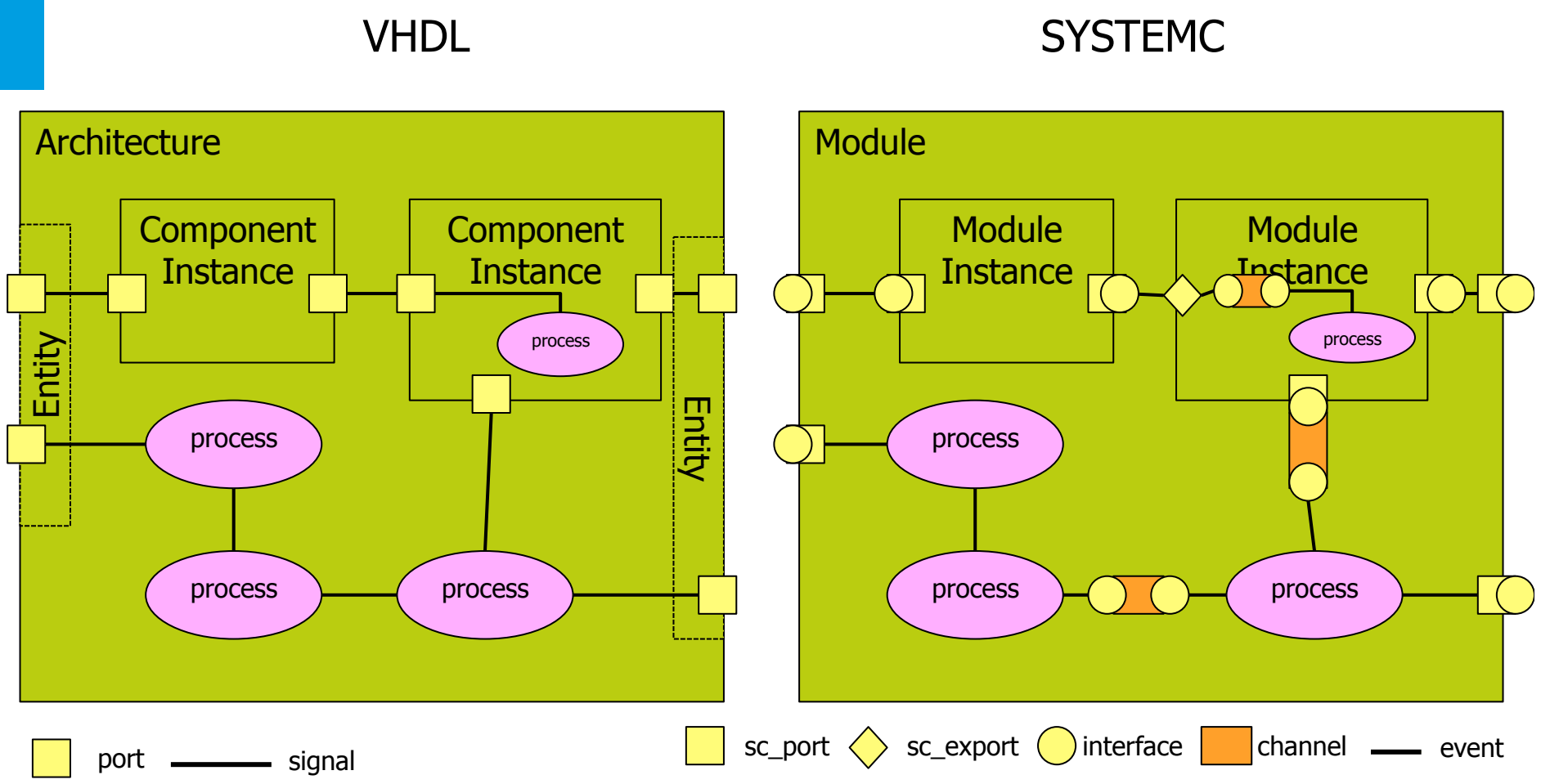
- sc_module – A hierarchy containing
  - Ports bound to interfaces
    - For external communication
  - Instances of other "sub" modules
    - Creates hierarchy by composition
  - Instances of channels implementing interfaces
    - For communication between modules
    - Implement interfaces for use with ports
- Processes
  - Providing concurrent behaviors
- Local data and helper methods as needed







VHDL/SystemC Components



General SystemC Requirements

- Declarations (entity) in header file (.h)
- Definitions (architecture) in implementation file (.cpp)
- One module/class per file
- Include **<systemc>** to access SystemC
- Namespace **sc_core** and **sc_dt** hide all the details
  - Explicit scoped name in headers (never **using**)
  - May have **using namespace** directives in implementation

Anatomy of a MODULE (Overview)

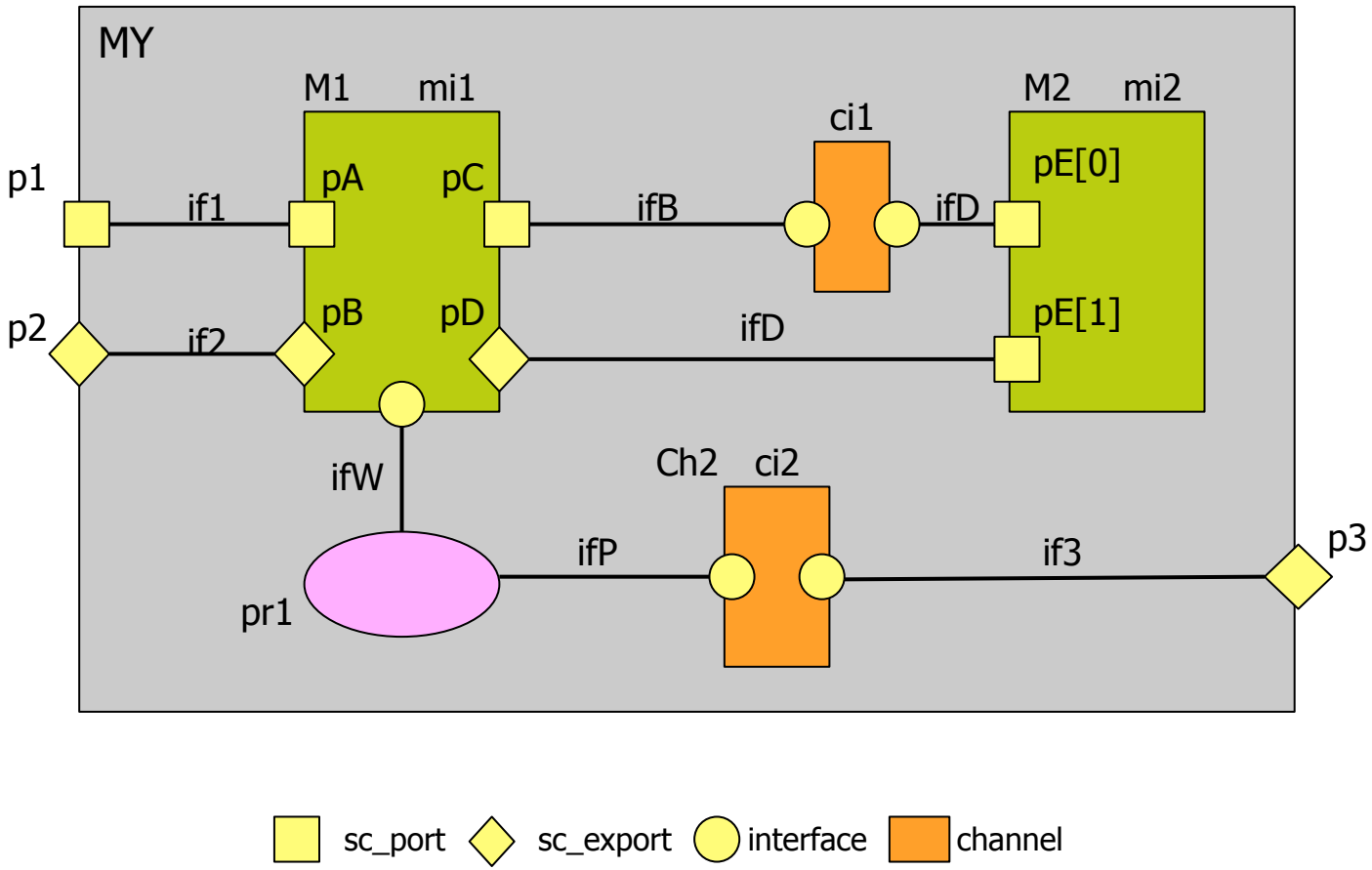
Entity/Header MDL_NAME.h

```
#ifndef _MDL_NAME_H_
#define _MDL_NAME_H_
#include <systemc>
struct MDL_NAME
: sc_core::sc_module {
//ports, exports, & sockets
//processes & overrides
//module instances
MDL_NAME(…);//Constructor
~MDL_NAME(void);//Destructor
private:
//channels
//variables
//events
//helpers

};
#endif /* _MDL_NAME_H_ */
```

Implementation MDL_NAME.cpp

```
#include "MDL_NAME.h"
using namespace sc_core;
MDL_NAME::MDL_NAME(//Constructor
sc_module_name instance, …
)
: sc_module(instance), …
{
//dynamic instantiation
//connectivity
SC_HAS_PROCESS(MDL_NAME);
//process registration
//static sensitivity
}
MDL_NAME::~MDL_NAME(void) {}
//methods (e.g. processes)
```

SC_MODULE Graphical View



SC_MODULE Text View

MY.h

```
#ifndef _MY_H_
#define _MY_H_
struct M1; struct M2;
SC_MODULE(MY) {
  sc_port<if1>     p1;   //Ports
  sc_export<if2> p2;
  sc_export<if3> p3;
  Ch1 ci1; Ch2 *ci2;  //Channels
  M1* mi1; M2* mi2; //Sub-modules
  void pr1(void);   //Processes
  SC_CTOR(MY);   //Constructor
};
#endif
```

MY.cpp

```
#include <systemc>
#include "M1.h"
#include "M2.h"
#include "MY.h"
// continued…
```

MY.cpp

```
// continued…
void MY::pr1(void) {
  while(true) {
    wait(mi1.ready());
    ci2->process(mi1.v());
  }
}//end MY::pr1()
MY::MY(sc_module_name nm)
: sc_module(nm) {
  mi1 = new M1("mi1");
  mi2 = new M2("mi2");
  SC_HAS_PROCESS(MY);
  SC_THREAD(pr1); //Register
  mi1->pA(p1);          //Connect
  mi1->pB(p2);
  mi1->pC(ci1);
  mi2->pE[0](ci1);
  mi2->pE[1](mi1->pD);
}//end MY::MY()
```

Behavior

Constructor

SystemC Macros

- Following three macros are fairly common
  - We consistently use only **SC_HAS_PROCESS()**

```
#define SC_MODULE(user_module_name)        \
        struct user_module_name : ::sc_core::sc_module

#define SC_CTOR(user_module_name)                    \
        typedef user_module_name SC_CURRENT_USER_MODULE; \
        user_module_name( ::sc_core::sc_module_name )

// SC_HAS_PROCESS macro call must be followed by a ;
#define SC_HAS_PROCESS(user_module_name)             \
        typedef user_module_name SC_CURRENT_USER_MODULE
```

```
SC_MODULE(my_module) {
SC_CTOR(my_module);
};
```

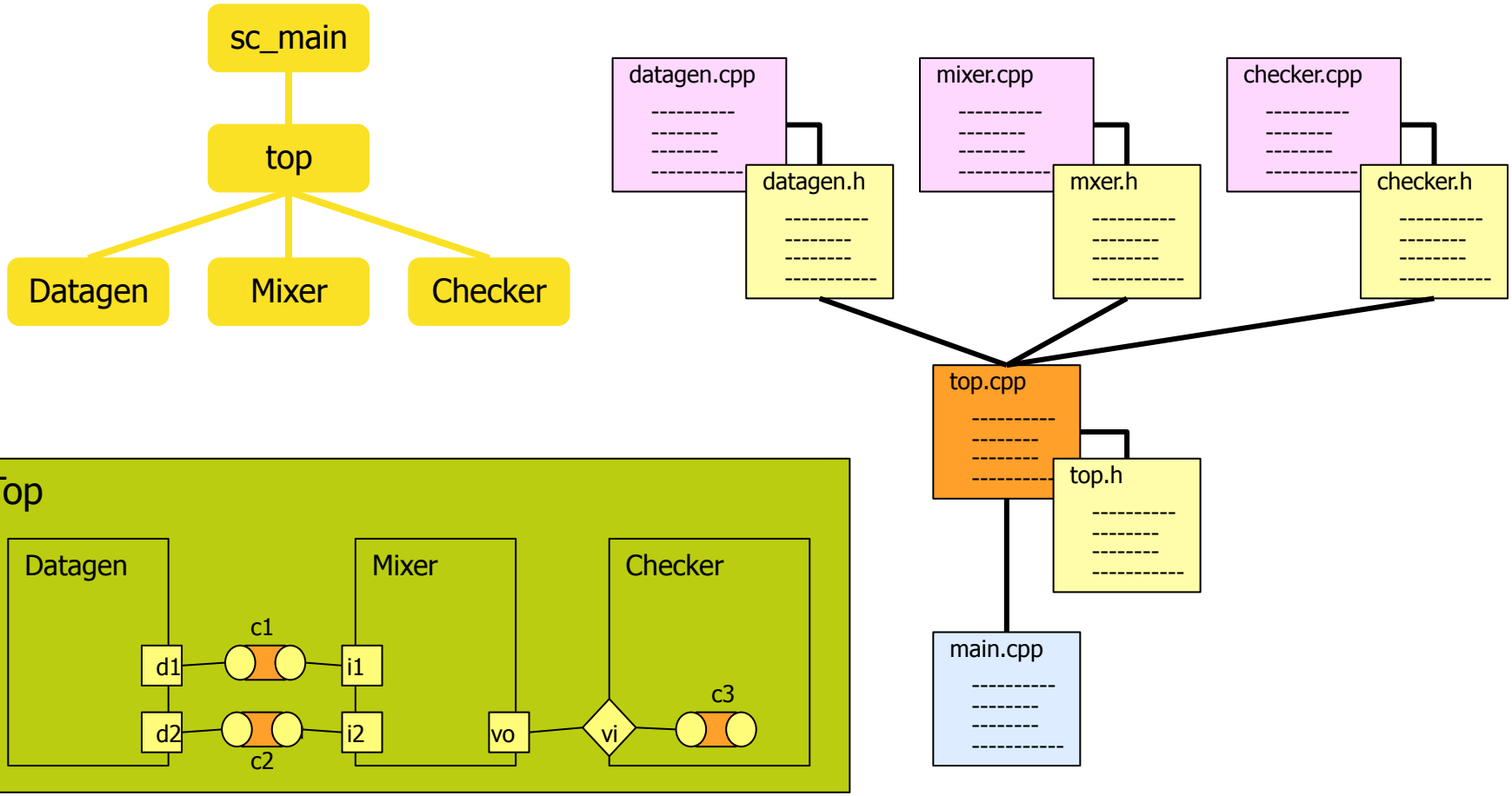
Anatomy Module.h (header)

```
#ifndef _MODULE_H_
#define _MODULE_H_
#include <systemc>
// HEADING TEXT (EXTERNAL_DESCRIPTION, COPYRIGHT)
struct SUBMODULE;              // Forward declaration - no need for headers
struct MODULE: public sc_core::sc_module {
   sc_core::sc_port <INTERFACE> PORT;        // Ports
   sc_core::sc_export<INTERFACE> XPORT;      // Ports
   MODULE(sc_module_name inst);              // Constructor
   ~MODULE(void);                            // Destructor
Private:
   SUBMODULE*     SUBMOD_iptr;               // Submodules
   CHANNEL*       CH_chan_iptr;              // Subchannels
   TYPE_NAME      MODULE_DATA;               // Private data
   void   PROC1_thread(void);                // Process(es)
   void   PROC3_method(void);                // …
   RET_TYPE FUNC(ARG_TYPE…);                 // Helper methods (member funcs)
};
#endif /* _MODULE_H_ */
```

Anatomy of Module.cpp (implementation)

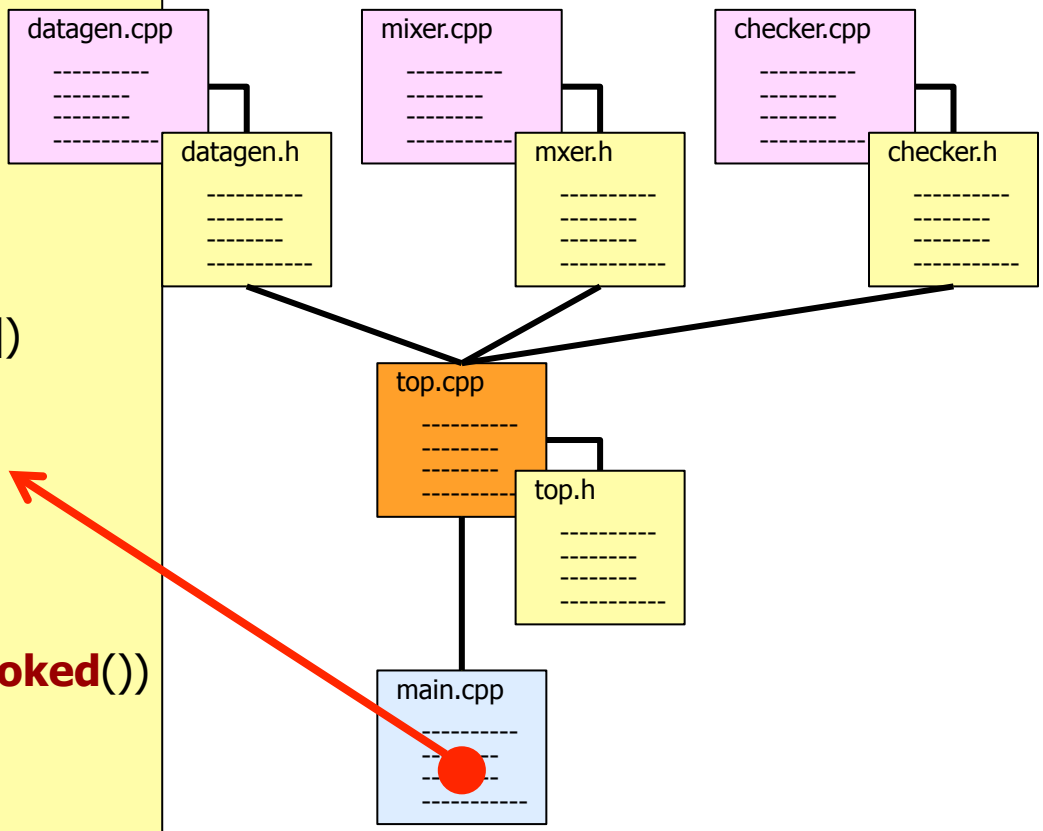
```
// HEADING TEXT (INTERNAL_DESCRIPTION, COPYRIGHT)
#include "MODULE.h"                    // and other includes (SUBMODULE.h,…)
using namespace sc_core;               // and other using's
MODULE::MODULE(sc_module_name instance_name)
: sc_module(instance_name)                                  // INITIALIZATION
, PORT("PORT")                                              // - Port names
, SUBMOD_iptr(new SUBMODULE("SUBMOD_iptr"))                 // - Instance
, CH_chan_iptr(new CHANNEL("CH_chan_iptr"))                 // - Instance
{ SUBMODULE->PORT(CHANNEL_OR_PORT);                         // - Connect
    XPORT(CHANNEL_OR_XPORT);                                // - Connect
   SC_HAS_PROCESS(MODULE);
   SC_THREAD(PROC1_thread);                                 // - Register
   SC_METHOD(PROC2_method);                                 // - Register
      sensitive << EVENT_OR_PORT;                           // - Static
}
void MODULE::PROC1_thread(void) { for(;;) {/* BEHAVIOR */}; }
void MODULE::PROC2_method(void) { /* BEHAVIOR */; return; }
RET_TYPE MODULE::FUNC(ARG_TYPE VAR) { /* BEHAVIOR */; return VALUE; }
```

Simple example – main.cpp



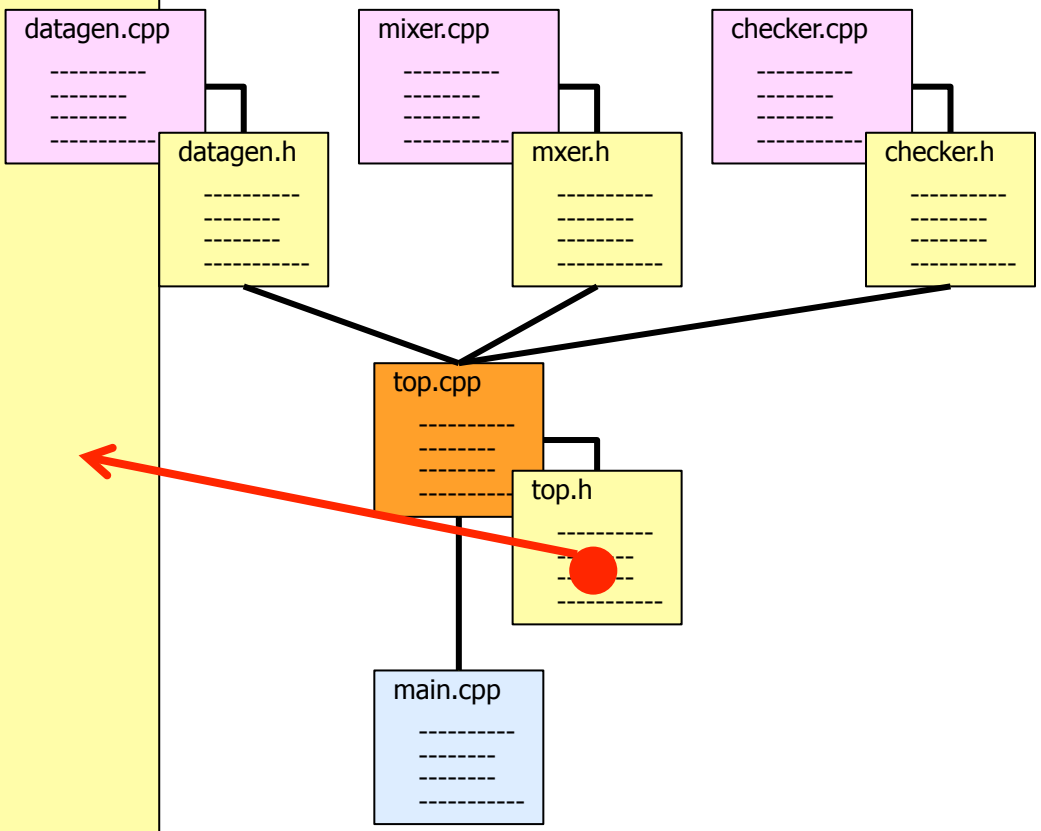
Simple example – main.cpp

```
#include "top.h"
using namespace sc_core;

int exit_status = 0; // global

int sc_main(int argc, char *argv[])
{
    top top_i("top_i");
    sc_start(1000,SC_NS);

    if(!sc_end_of_simulation_invoked())
        sc_stop();
    return exit_status;
}
```



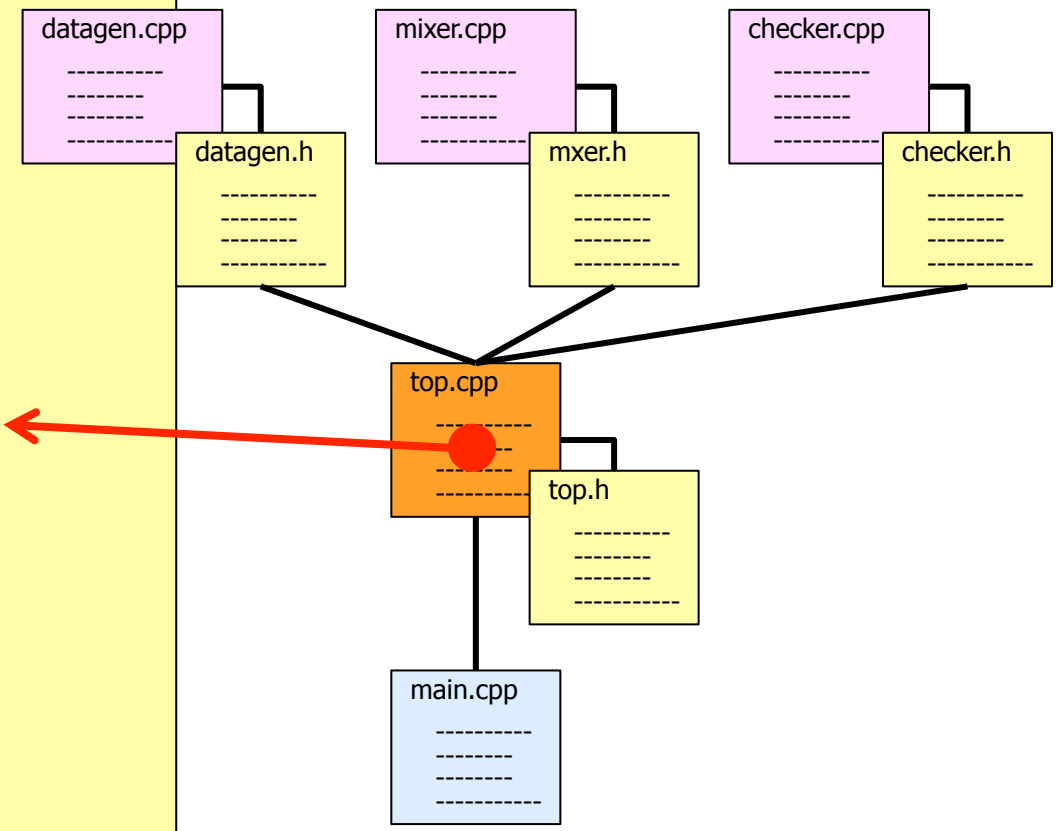
Simple example – top.h

```
#ifndef TOP_H
#define TOP_H
#include <systemc>
struct datagen;
struct mixer;
struct checker;

SC_MODULE(top) {
  SC_CTOR(top); //< Constructor
private:
   // Instantiate fifo channels
   sc_core::sc_fifo<int> c1;
   sc_core::sc_fifo<int> c2;
  // Declare modules
  datagen* gen;
  mixer*     mix;
  checker*  chk;
};
#endif
```



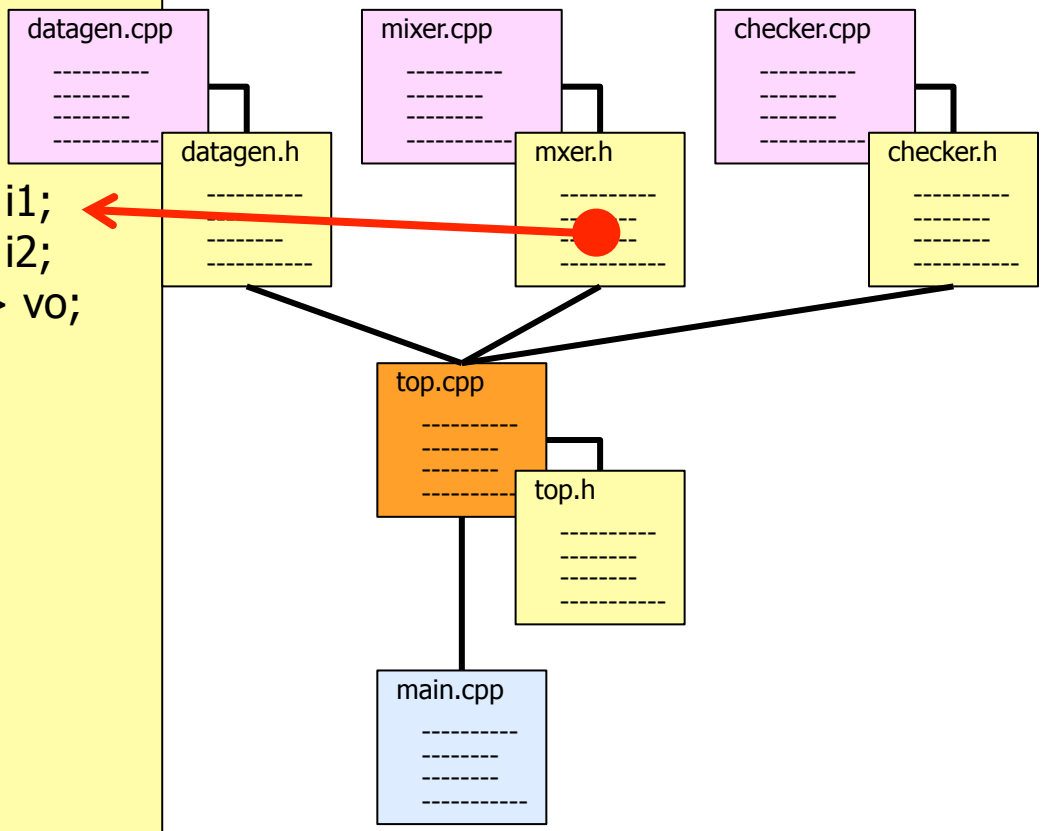
Simple example – top.cpp

```
#include "top.h"
#include "datagen.h"
#include "mixer.h"
#include "checker.h"
using namespace sc_core;

top::top(sc_module_name inst)
: sc_module(inst)
, gen (new datagen("gen"))
, mix (new mixer  ("mix"))
, chk (new checker("chk"))
{  // Instantiate modules
   // Connectivity
   gen->d1(c1);
   gen->d2(c2);
   mix->i1(c1);
   mix->i2(c2);
   mix->vo(chk->c3);
};
```



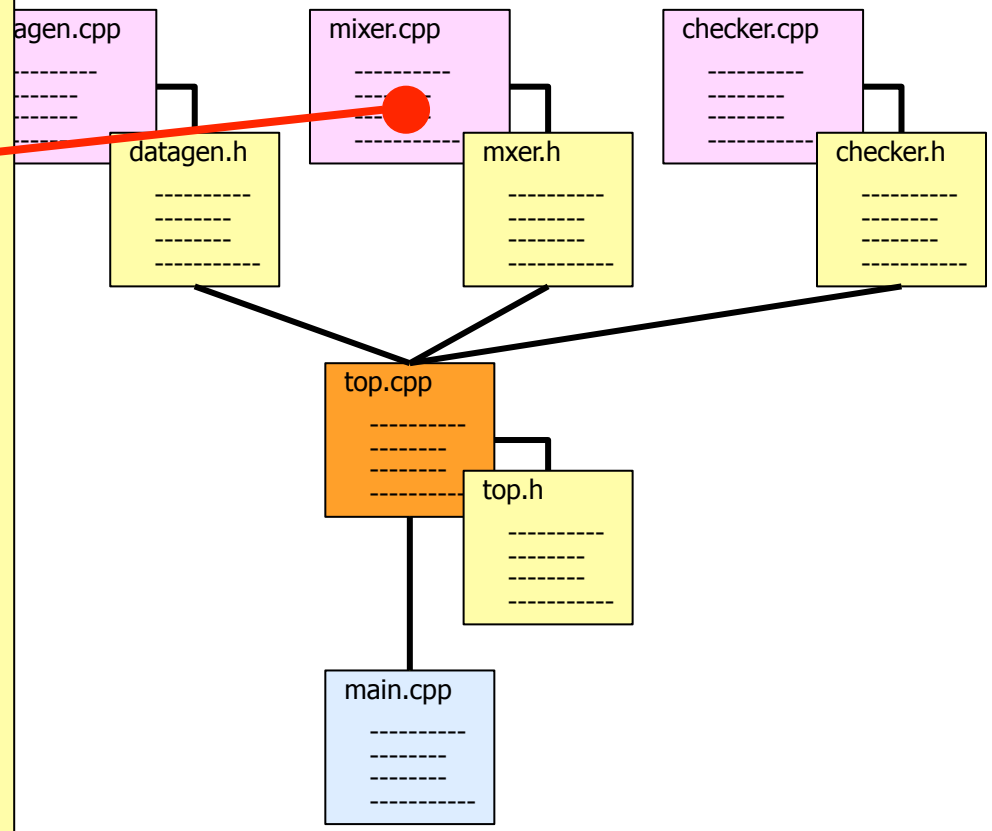
Simple example – mixer.h

```
#ifndef MIXER_H
#define MIXER_H
#include <systemc>
SC_MODULE(mixer) {
  // Ports
  sc_port<sc_fifo_in_if<int> >  i1;
  sc_port<sc_fifo_in_if<int> >  i2;
  sc_port<sc_fifo_out_if<int> > vo;
  // Module constructor
  SC_CTOR(mixer);
private:
  // Process declaration
  void mixer_thread(void);
};
#endif
```



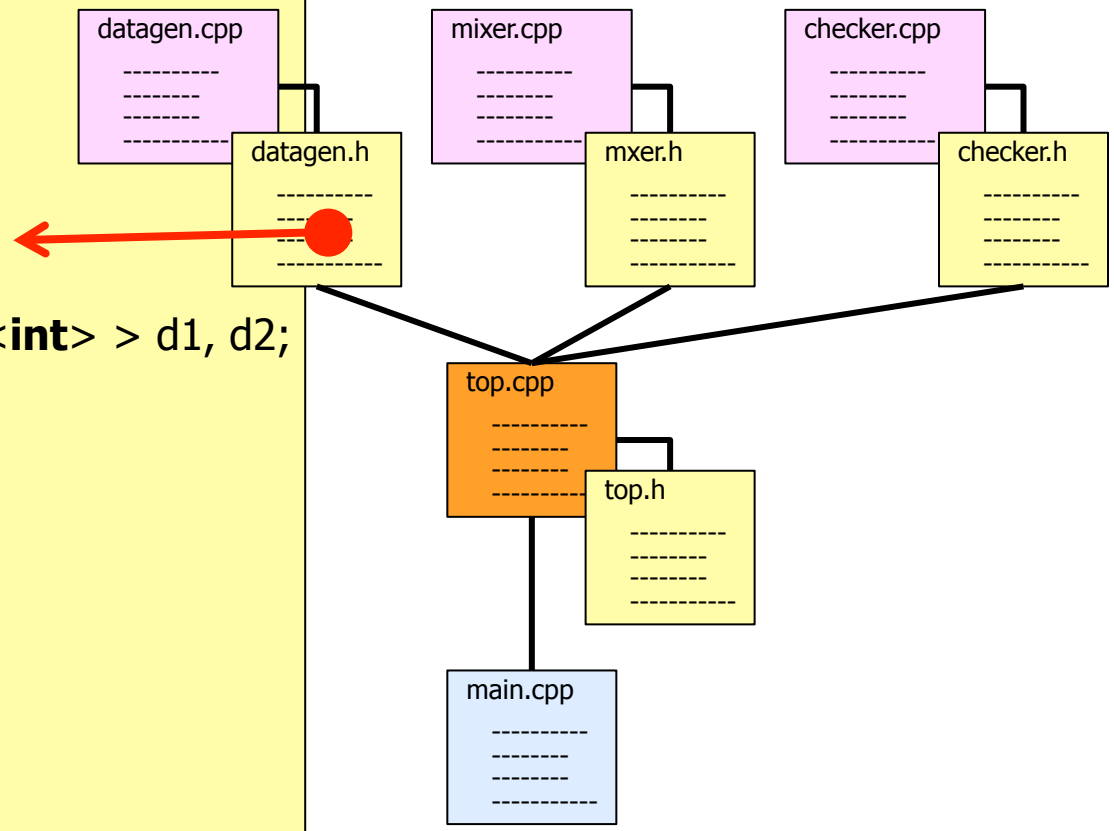
Simple example – mixer.cpp

```
#include "mixer.h"
using namespace sc_core;
// Constructor
mixer::mixer(sc_module_name inst)
: sc_module(inst)
{  // Register processes
    SC_HAS_PROCESS(mixer);
    SC_THREAD(mixer_thread);
}
// Processes
void mixer::mixer_thread(void)  {
   int k1, val1, k2, val2;
   while (true) {
      k1   = i1->read();
      val1 = i1->read();
      k2   = i2->read();
      val2 = i2->read();
      vo->write(k1*val1 + k2*val2);
   }
}
```



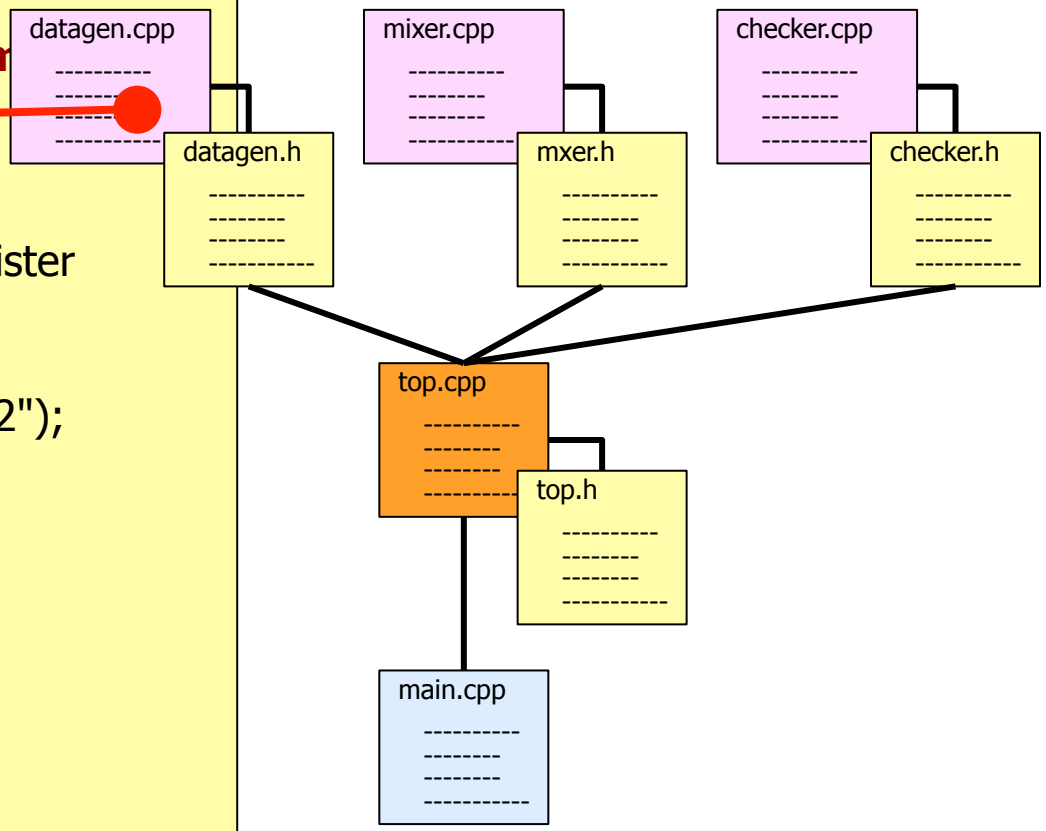
Simple example – datagen.h

```
#ifndef DATAGEN_H
#define DATAGEN_H
SC_MODULE(datagen) {
   // Ports
   sc_core::sc_port
   < sc_core::sc_fifo_out_if <int> > d1, d2;
   // Process declaration
   void stim_thread(void);
   // Module constructor
   SC_CTOR(datagen);
};
#endif
```



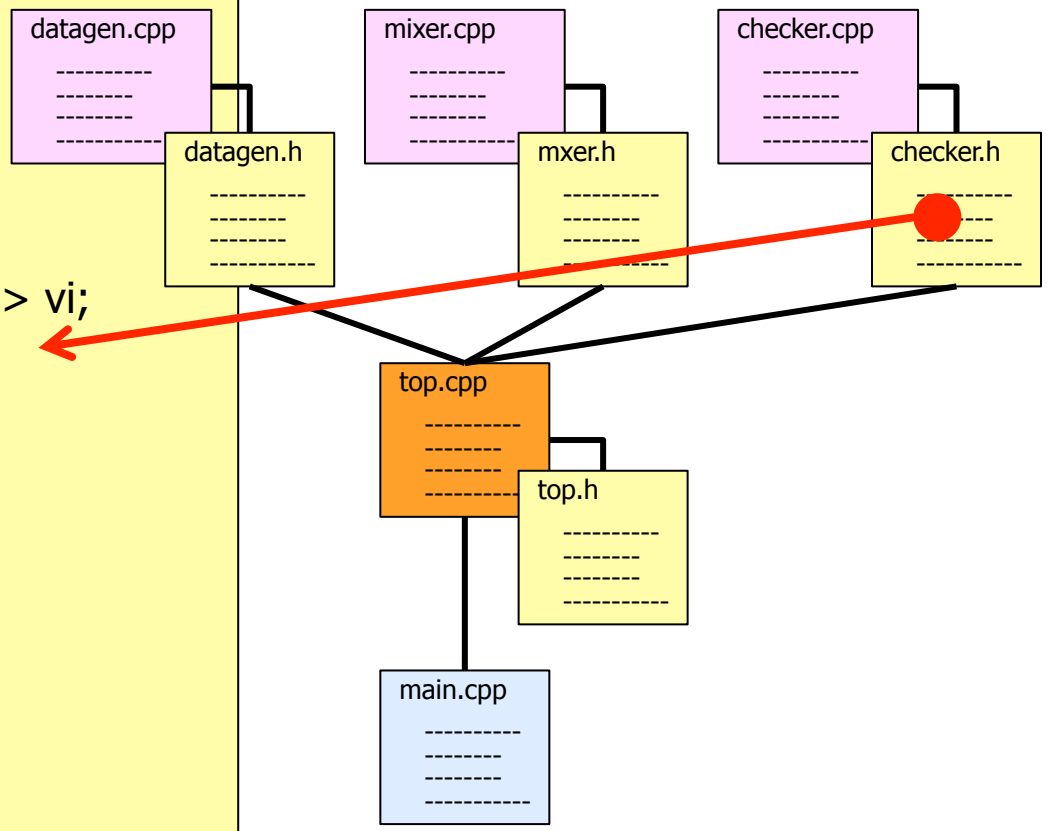
Simple example – datagen.cpp

```
#include <fstream>
#include "datagen.h"
using namespace sc_core;
// Constructor
datagen::datagen(sc_module_nam
: sc_module(nm)
{ // Register
   SC_HAS_PROCESS(datagen);
   SC_THREAD(stim_thread);//Register
}
void datagen::stim_thread(void) {
   std::ifstream f1("img1"),f2("img2");
   do {
      int data1, data2;
      f1 >> data1; f2 >> data2;
      d1->write(data1);
      d2->write(data2);
   } while (!f1.eof() && !f2.eof());
   // What happens when this exits?
}
```



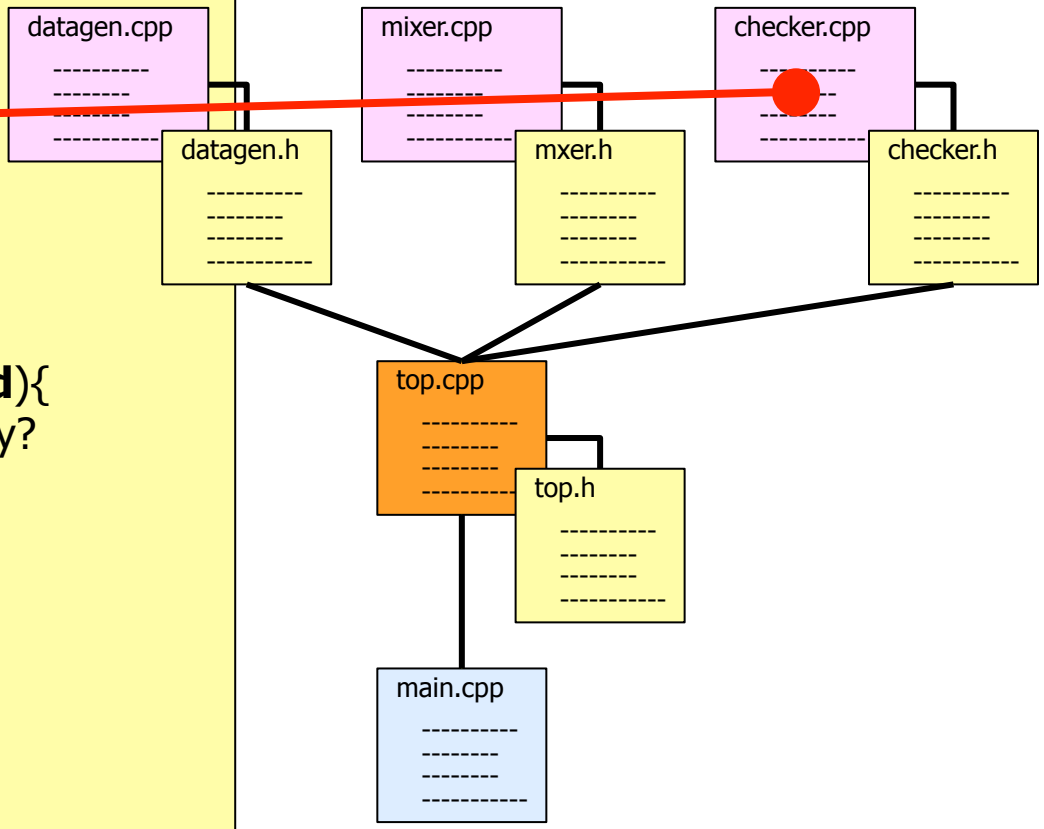
Simple example – checker.h

```
#ifndef CHECKER_H
#define CHECKER_H
#include <systemc>
SC_MODULE(checker) {
  // Ports
  sc_core::sc_export
  < sc_core::sc_fifo_in_if<int> > vi;
  sc_core::sc_fifo<int> c3;
  // Module constructor
  SC_CTOR(checker);
private:
  int const wid;   // row width
  int       cnt;   // pixel count
  int       pel;   // pixel element
 // Process declaration
  void checker_method(void);
};
#endif
```



Simple example – checker.cpp

```
#include "checker.h"
#include <iostream>
using namespace sc_core;
checker::checker(sc_module_name nm)
: sc_module(nm)
, cnt(0), wid(8)
{ // Register processes
    SC_HAS_PROCESS(checker);
    SC_METHOD(checker_method);
    vi.bind(c3);
}
void checker::checker_method(void){
   next_trigger(                    // why?
     c3.data_written_event());
   while (c3.nb_read(pel)) {
     if (pel == 0) pel = int('.');
     std::cout << char(pel) << ' ';
     if ((cnt++ % wid) == (wid-1))
       std::cout << std::endl;
   }
}
```



4.

Simulation
Processes , Events, Time

Characteristics of SystemC Simulation

- Use "simulation processes" to implement concurrency
- Same as **process** in VHDL or **always**/**initial** block in Verilog
- Important to realize concurrency is simulated
- Simulation operates in a single OS process/thread
- Simulated processes are never preemptively interrupted
- Simulated processes cooperatively yield control to others
- Typical OS locking not needed
- Could change in a future multi-core implementation

The SystemC Processes

- Simulation Processes the main behavior of the simulation
- Majority of functionality is described in processes
- Or invoked as a result of function calls from processes
- Simulation Processes are normal C++ functions except
- Simulation Processes are registered with the SystemC kernel
- SystemC kernel runs and manages these functions
- Never call these "simulation process functions" directly
- Only the SystemC kernel calls simulation process functions

Simulation Process Characteristics

- Processes are not hierarchical
  - Cannot have a process inside another process
    (use module for hierarchical design)
- Processes may spawn other processes (see dynamic below)
- Processes use events and timing controls to synchronize
  - Synchronization is essential for safe communication
- Processes can be static or dynamic
  - Processes registered during elaboration are called static processes
  - Processes registered during run-time are called dynamic processes
- Modules may have many processes
  - Okay to register multiple different functions as processes in a module
  - Only dynamic process functions may be registered more than once

Simulation Process Types

- SystemC (v2.x) supports two primary process types:
  - Method processes (SC_METHOD) – similar to Verilog always block
  - Thread processes (SC_THREAD) – similar to Verilog initial block
  - CThread processes (SC_CTHREAD) – similar to SC_THREAD but sensitive to a particular clock edge
- Not the same as software OO "method" or "thread"
  - Although names were derived for reasons to become clearer
- Can have different process types inside the same module

Process Usage:

System Algorithmic models tend to use Threads or untimed Methods

System Performance models tend to use both

Transaction Level Models tend to use both

Behavioral/Algorithmic Synthesis models tend to use clocked Threads

RTL models use Methods

Test benches may use all process types (Threads and Methods)

SC_METHOD

- **SC_METHOD** is a SystemC process with following properties
  - Called at initialization
  - They run completely and then return
  - Must not have infinite loop
  - Must **return** to allow other processes to proceed
  - Executes **repeatedly** as one of its sensitivity signals changes value
  - Such processes must also avoid using calls to blocking methods
- Does not have saved local variables –use **sc_module** private data for this purpose
  - **static** doesn't work due to possible multiple instantiations of modules
- 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.
  - Local variables preserved as expected
  - 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 << …. ;

SC_CTHREAD

- **SC_CTHREAD** is a SystemC process with following properties
  - SC_CTHREAD is executed **only once** when the first single edge (pos/neg) of the clock signal sensed occurs.
  - Local variables preserved as expected
  - SC_CTHREAD can be suspended during execution using
    - *wait()*, suspended till next clock edge
    - *wait(int)*, suspended for *int* clock_edges
  - SC_CTHREAD provides for a function **reset_signal_is(reset_name, bool)** to express the reset condition for a clocked thread
- SC_CTHREAD processes are registered as
  SC_CTHREAD(process_name, clock_name.[pos() | neg()]);

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();
…
```

Creating static Process types

- Member functions must return void with no arguments
  - What args would SystemC kernel supply?

- Macros register the simulation process with the SystemC simulation kernel inside constructor

```
// example.h
SC_MODULE(example) {
      void my_thread(void);
      void my_thread2(void);
      void my_method(void);
      SC_CTOR(example);
};
```

```
// example.cpp
example::example(sc_module_name nm)
: sc_module(nm)
{
      SC_HAS_PROCESS(example);
      SC_THREAD(my_thread); // Register
      SC_THREAD(my_thread2);
      SC_METHOD(my_method);
}
```

Time

- Simulated time is managed by the SystemC kernel
  - Use previously discussed **sc_time** data type
- Internally, a global 64-bit unsigned number
- Resolution defaults to picoseconds
  - 2^{64} ps = 30 weeks 3 days 12 hours 5 min 44 sec
- Use sc_time_stamp() to access

```
sc_time my_time;
double d = 5.0;
// Assume it is currently 25 ns
cout << "Time is " << sc_time_stamp() << endl;
my_time = sc_time_stamp()/2 + sc_time(d, SC_US);
cout << "Calculated" << mytime << endl;
```

```
Time is 25 ns
Calculated 5012500 ps
```

Time

- To delay an SC_THREAD to wait specified amount of time
  - Use **wait**(**sc_time**) or **wait**(amt,units)
  - Examples: **wait**(5*period) or **wait**(1,SC_SEC)
- Can similarly invoke delay to future with SC_METHOD
  - Use **next_trigger**(**sc_time**) or **next_trigger**(amt,units)
  - Example: **next_trigger**(**sc_time**(2, SC_MS))

```
// Assume it is currently 25 ns
sc_time my_time = sc_time_stamp();
wait(5,SC_NS);
cout << "Time is " << sc_time_stamp() << endl;
wait(2*(sc_time_stamp()-my_time));
cout << "Delayed " << my_time
<< " to " << sc_time_stamp() << endl;
```

```
Time is 30 ns
Delayed 10 ns to 40 ns
```

Events (sc_event)

- SystemC simulator is event driven
  - similar to Verilog/VHDL
  - Event is a basic synchronization object
  - Use events to synchronize process communications
- Event has no data type, only control
  - An event is a singularity
  - Events "happen"

```
sc_event evt1, evt2;
```

notify() Method

- **sc_event**::**notify**() triggers an event
  - May be immediate or delayed
  - **notify**() method itself does not consume time

```
Syntax
void notify( void );              // immediate
void notify( const sc_time& );    // delayed/timed
void notify( double, sc_time_unit ); // convenience
```

```
sc_event evt1, evt2, evt3;
sc_time t15(15,SC_SEC);
// Assume it is currently 25 ns
cout << "Time is " << sc_time_stamp() << endl;
evt1.notify();
evt2.notify(t15);
evt3.notify(15,SC_SEC);
cout << "Time is " << sc_time_stamp() << endl;
```

```
Time is 25 ns
Time is 25 ns
```

Wait() Method

- **sc_module**::**wait**(args) triggers an event
  - Suspends thread process until specified event happens

Syntax (partial)

```
void wait( sc_event );
```

Other syntaxes will be discussed shortly.

```
// assume previous slide
extern sc_event my_event;
wait(5, SC_NS);
cout << "Time is " << sc_time_stamp() << endl;
wait(my_event);
cout << "Time is " << sc_time_stamp() << endl;
wait(my_event);
cout << "Time is " << sc_time_stamp() << endl;
```

```
Time is 5 ns
Time is 25 ns
Time is 40 ns
```

Complete wait & next_trigger syntax

```
sc_event e1, e2 ,e3;
sc_time t(200, SC_NS);

// first event of a list of events
wait(e1);
wait(e1 | e2 | e3);     // first one
wait( e1 & e2 & e3); // all in any order

// specific amount of time
wait(200, SC_NS);
wait(t); // wait for 200 ns

// events occur or after timeout
wait(200, SC_NS, e1 | e2 | e3);
wait(t, e1 | e2 | e3);
wait(200, SC_NS, e1 & e2 & e3);
wait(t, e1 & e2 & e3);

// next delta cycle (more later)
wait( 0, SC_NS );
wait( SC_ZERO_TIME );
```

```
sc_time t(200, SC_NS);

// first event of a list of events
next_trigger(e1);
next_trigger(e1 | e2 | e3);   // first one
next_trigger( e1 & e2 & e3);// any order

// specific amount of time
next_trigger(200, SC_NS);
next_trigger(t); // wait for 200 ns

// events occur or after timeout
next_trigger(200,SC_NS,e1 | e2 | e3);
next_trigger(t, e1 | e2 | e3);
next_trigger(200,SC_NS,e1 & e2 & e3);
next_trigger(t, e1 & e2 & e3);

// next delta cycle (more later)
next_trigger( 0, SC_NS );
next_trigger( SC_ZERO_TIME );
```

Practical application

- When using timeout form of wait
  - Capture and compare before/after times

```
const sc_time TIMEOUT(15,SC_SEC);
sc_event handshake_evt;
...
{
        sc_time before_timeout(sc_time_stamp());
        wait(TIMEOUT, handshake_evt);
        if (sc_time_stamp() == before_timeout+TIMEOUT) {
                SC_REPORT_ERROR(MSGID,"Timeout!");
        }
}
```

Multiple Notifications

- An sc_event may only have one notification scheduled at any given time
- If another notification is executed then the "soonest" notification is kept

```
Given:
sc_event event_a;

Then if at time t:
event_a.notify(10,SC_NS);// scheduled
event_a.notify( 5,SC_NS);// scheduled / previous tossed
event_a.notify(15,SC_NS);// not scheduled

Result of the above code is a notification scheduled for time t + 5ns
```

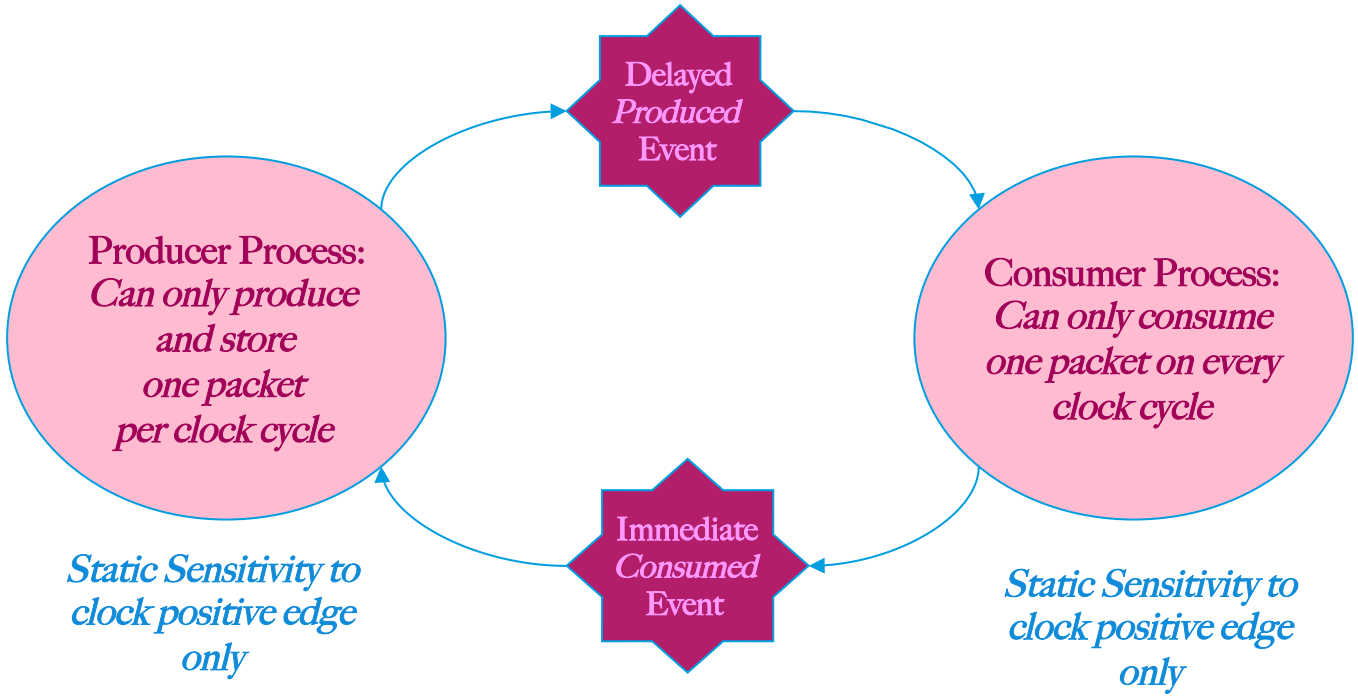
Other sc_event Methods

- void cancel() - cancels pending notifications for an event
  - Supported for delayed or timed notification
  - Not supported for immediate notification (does nothing)

```
Given!
sc_event a, b, c;              // event
sc_time t(10, SC_MS);          // variable t of type sc_time

a.notify();                    // current delta cycle
b.notify(SC_ZERO_TIME);        // next delta cycle
c.notify(t);                   // 10 ms delay

Cancel an event notification
a.cancel();        //Does nothing - Can't cancel immediate notification
b.cancel();        //Cancel notification on event b
c.cancel();        //Cancel notification on event c
```

Dynamic Sensitivity

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



Dynamic Sensitivity:
A Producer Consumer Example

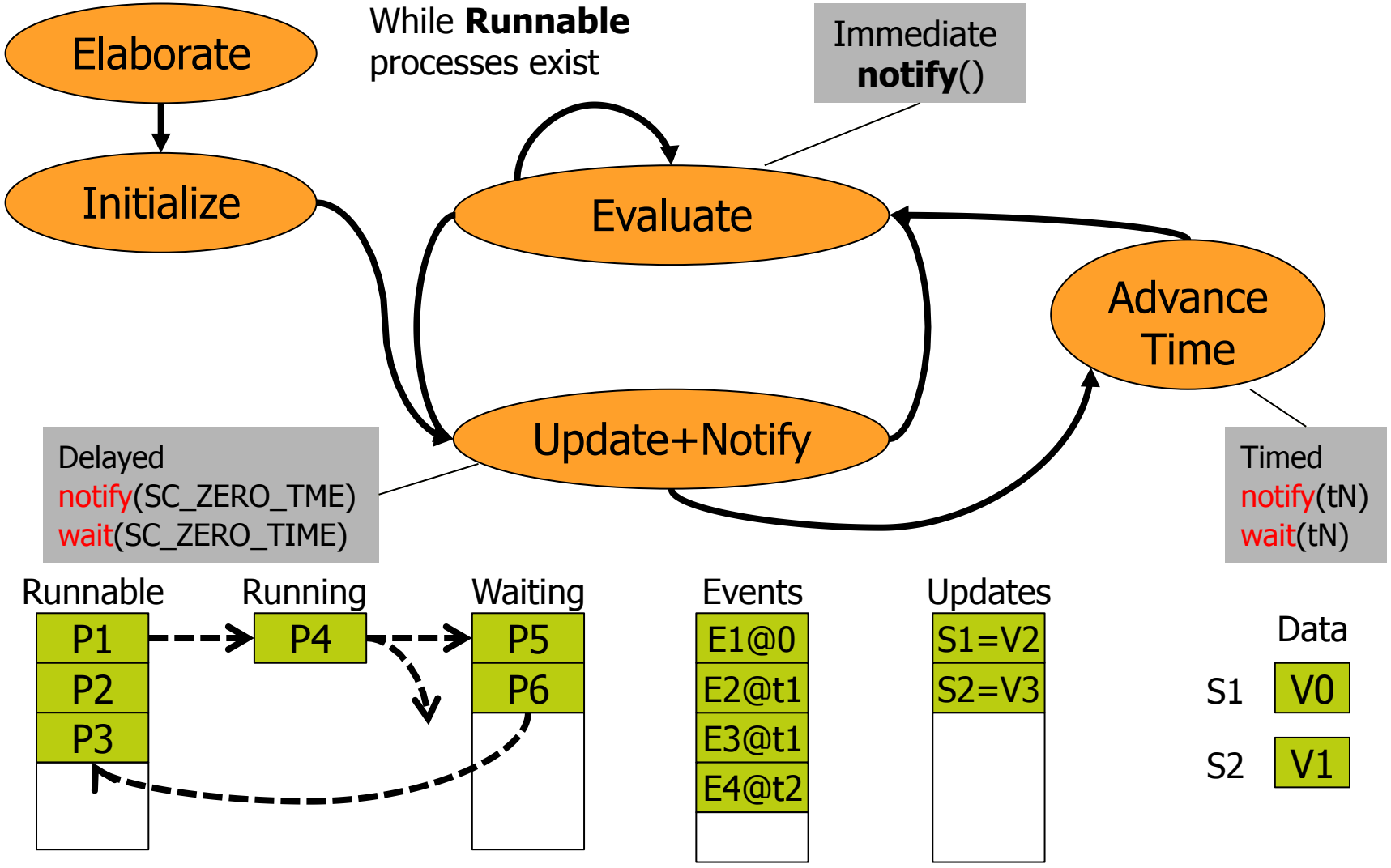
```
#include "systemc.h"
SC_MODULE (Dynamic) {
    sc_in_clk clk; sc_out<int> outport;
    sc_event p_event, c_event; int pkt;

    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;
    }
};
```

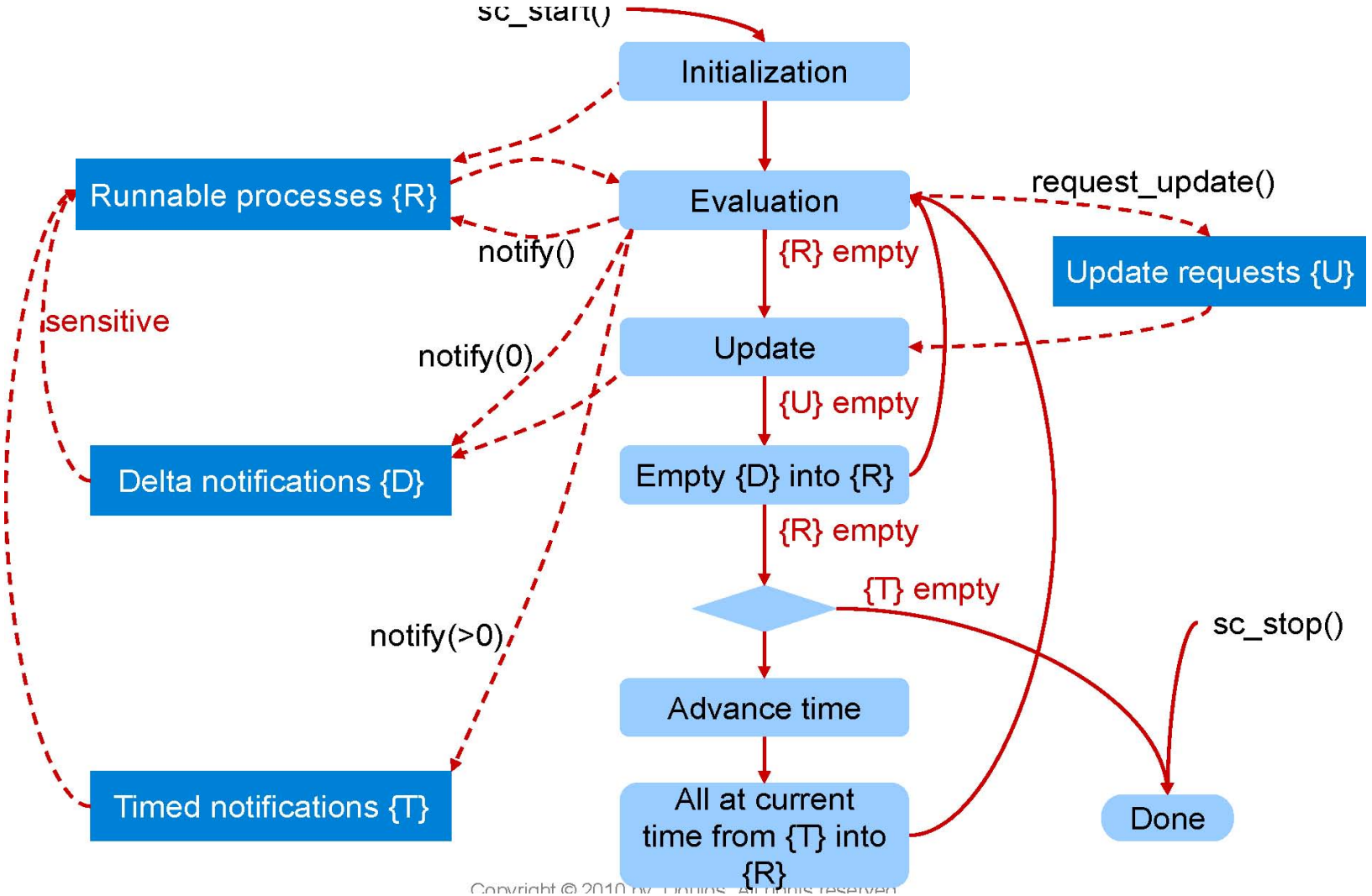
How Simulation Works (simplified)

- Event driven simulation kernel manages processes
  - Two major Phases
    - Elaboration Phase – constructs system to simulate
    - Simulation Phase – runs simulation processes
- During simulation phase
  - Scheduler selects when a simulation process may execute code
    - Processes execute code one at a time
  - Order of execution is not supposed to be determinable
    - Simulated concurrency
  - Behaves in the manner of cooperative multi-tasking
    - No preemptive interrupts to code
    - You control when & where to relinquish control

Simulation Engine



SystemC simulation kernel



5.

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
  - Uses ***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)

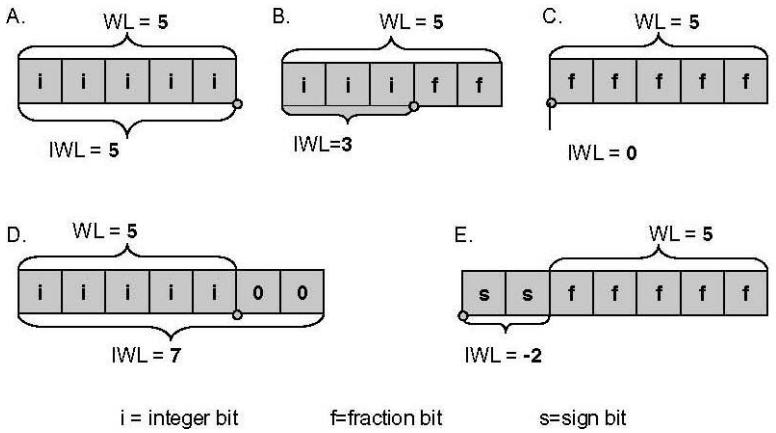
sc_fixed_fast <wl, iwl [, quant [, ovflw [, nbits]>

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 ! dynamic 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

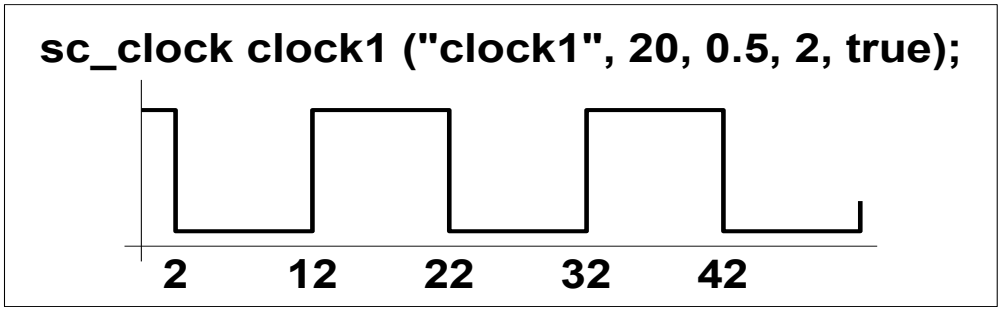
Name	Quantization Mode
SC_RND	Round
SC_RND_ZERO	Round towards zero
SC_RND_MIN_INF	Round towards minus infinity
SC_RND_INF	Round towards infinity
SC_RND_CONV	Convergent rounding[13]
SC_TRN	Truncate
SC_TRN_ZERO	Truncate towards zero

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

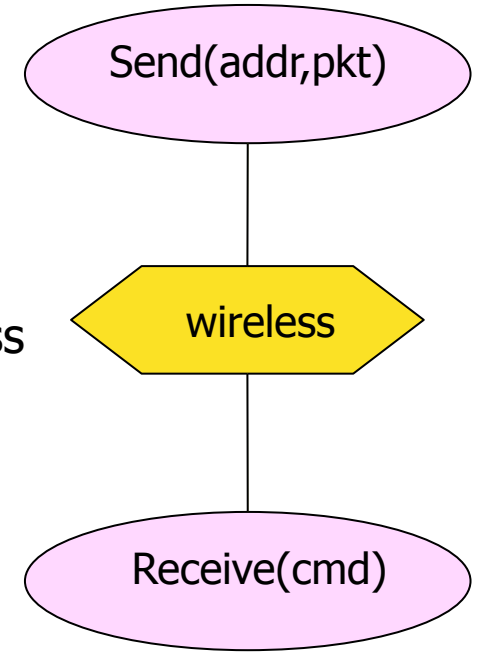


6.

Channels, Interfaces and ports

Channel

- Channels provide communications between processes
  - Transfer data & control
  - Take care of timing if modeled
  - Implemented as classes
- Examples
  - Simple: wire, FIFO, semaphore
  - Complex: AXI, ethernet, OCPIP, PCIe, wireless
- Provide
  - Safe communications
  - Ensure proper synchronization
  - Encapsulate details of communications
  - Separate algorithm from communications



SystemC Channel

SystemC channels are classes

- Inherit from one of two base classes
  - **sc_channel**
  - **sc_prim_channel**
- Also inherit from one or more interface classes
  - Defines what methods are supported
  - API (Application Programming Interface) of class

Interfaces

- C++ Interface Class
  - Defines an API
  - Defines signatures of functions
  - Contains only public pure virtual methods

```
class Bus_if {
public:
        virtual void send(unsigned,const packet&) = 0;
        virtual void receive(command& x) = 0;
};
```

- SystemC Interfaces
  - Same except additionally inherit from sc_interface
- Interfaces compel derived classes to implement methods

Types of Channels

- Two basic types
  - Hierarchical
  - Primitive
  - Not a critical distinction
- Primitive channels
  - Inherit sc_prim_channel
  - Intended to simulate fast
  - Not very complex
- Hierarchical channels
  - Inherit **sc_channel**
    - **sc_channel** is actually typedef of **sc_module**
    - Hierarchical channel is a module that implements an interface!
  - Can have processes, ports, contained channels and submodules
  - Used to model buses & may represent hardware

Standard Primitive Channels

- SystemC defines a few standard primitive channels
  - sc_fifo<T>
  - sc_mutex
  - sc_semaphore
  - sc_signal<T>
  - sc_buffer<T>
  - sc_signal_rv<T>
  - sc_event_queue
- Additionally, TLM 1.0 provides some example channels
  - tlm_fifo<T> - improves over sc_fifo in several ways
  - tlm_req_rsp_channel<T,T>
  - tlm_transport_channel<T>

Channel Observations

- Implements interfaces
  - Study the Interface classes to understand usage
- Two channels may implement the same interfaces
  - Different behaviors (levels of abstraction)
  - Swap for different degrees of accuracy and simulation speed
- Some act as adaptors
  - Interface different levels of abstraction
  - Usually hierarchical
- Most ESL designs involve design of channels
- Minimize primitive signal channels – can slow simulation
  - Used for interfacing to RTL (HDL's usually restricted to signals)
- FIFO's are useful

sc_fifo Interfaces

- sc_fifo channel, useful for modeling, has two interfaces:
- sc_fifo_in_if<T> (Read from FIFO)
  - void read(T&)
  - T read(void)
  - bool nb_read(T&)
  - int num_available(void) const
  - const sc_event& data_written_event() const
- sc_fifo_out_if<T> (Write to FIFO)
  - void write(const T&)
  - bool nb_write(const T&)
  - int num_free(void) const
  - const sc_event& data_read_event(void) const

sc_fifo<> Channel

- sc_fifo<T>
  - Implements sc_fifo_in_if & sc_fifo_out_if
  - Implements a FIFO (first-in first-out) behavior
  - Templatized as to type
  - As wide as the corresponding data type
  - Depth is user defined (default is 16 entries)

Declaration Syntax :

```
sc_fifo<T>   channel_name(depth=16);
```

T: C++, SystemC or user defined data type

Examples of sc_fifo< >

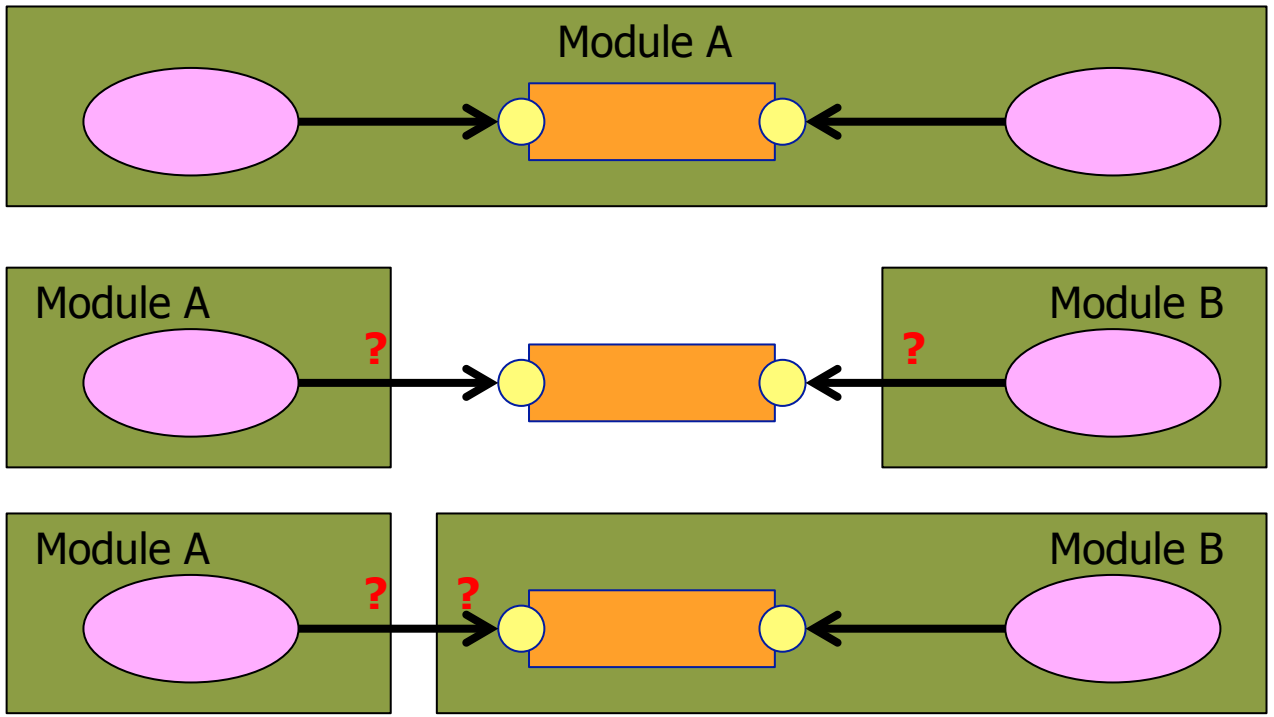
```
#ifndef _MYMOD_H_
#define _MYMOD_H_
SC_MODULE(mymod) {
   SC_CTOR(mymod);
   void my_thread(void);
   sc_fifo<int>  fifo1;
   sc_fifo<int>* fifo2_p;
};
#endif
```

```
##include <systemc>
using namespace sc_core;
#include "mymod.h"

// continued...
```

```
mymod::mymod(sc_module_name nm
): sc_module(nm)
, fifo1(256) // set depth
, fifo2_p (new sc_fifo<int>(5))
{
   SC_HAS_PROCESS(mymod);
   SC_THREAD(my_thread);
}
void my_mod::my_thread(void) {
  for(;;) {
    fifo1.write(42);
    int v = fifo1.read();
    while (!fifo2_p->nb_write(v)) {
      wait(
         fifo2_p->data_read_event());
    }//endwhile
  }//endforever
}
```

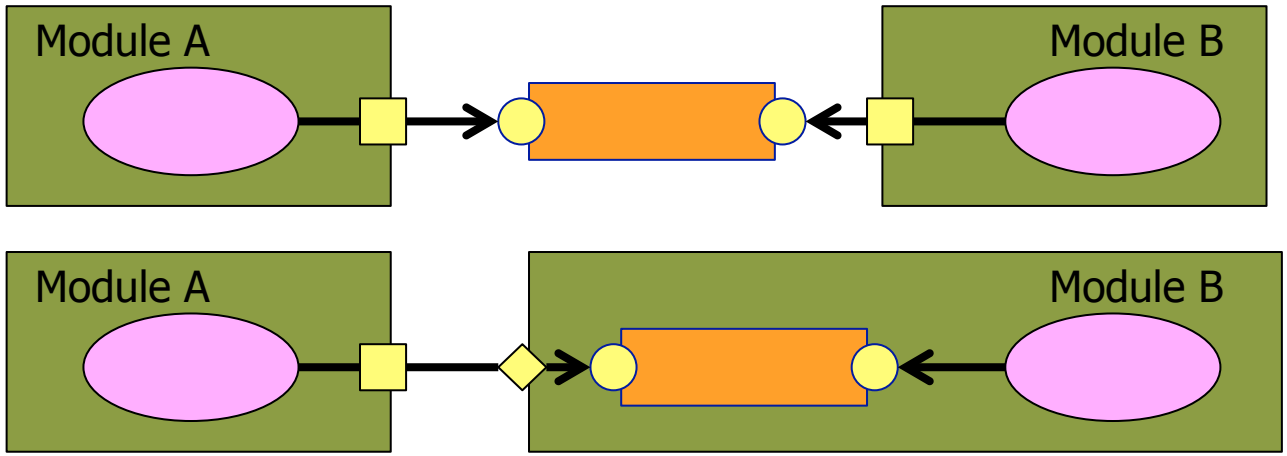
Communication Across Boundaries

- Communication is always between processes
  - How can a process inside a module object communicate with another embedded in an external module object



Communication with Ports

- Communication is always between processes
  - How can a process inside a module object communicate with another embedded in an external module object
- Use channels connected with one of two forms of ports
  - sc_port<INTERFACE> or sc_export<INTERFACE>

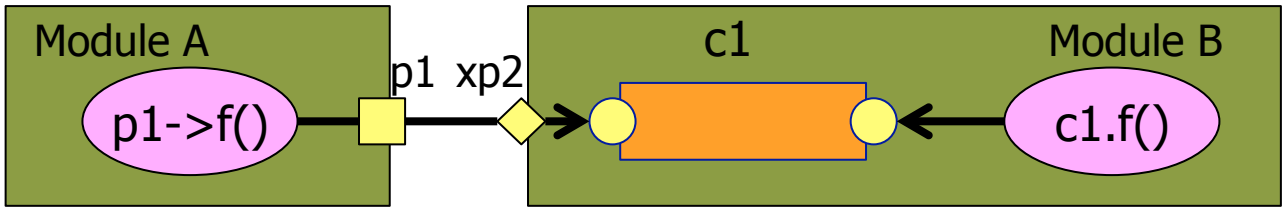


SystemC Ports and Exports

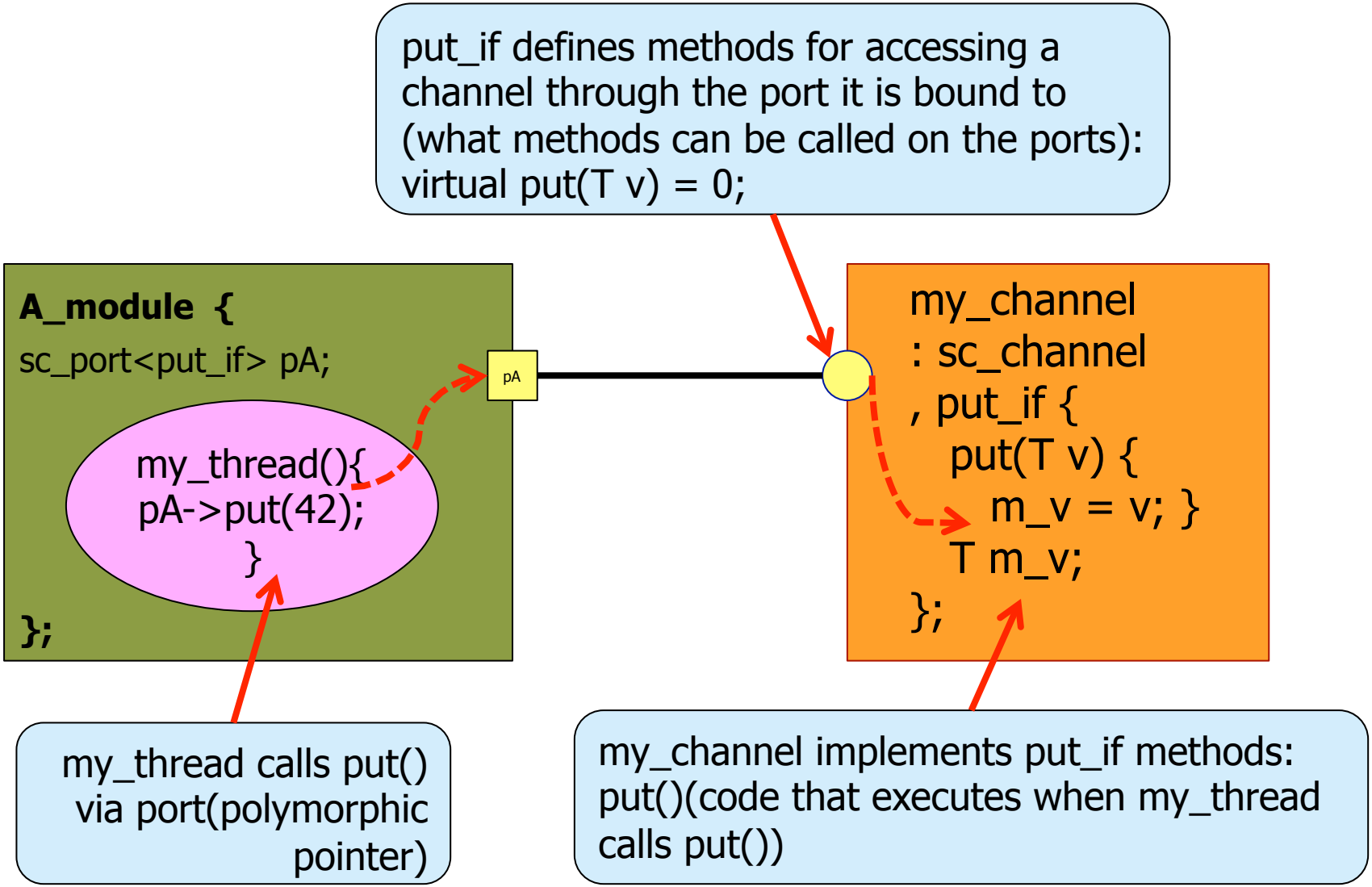
- Instantiate sc_[ex]ports directly in module declaration
  - Must be bound to a SystemC interface class

```
struct MODULE : sc_module {
sc_port<sc_fifo_out_if<int> >  p1;
sc_export<sc_fifo_in_if<int> > xp2;
...
};
```

- Ports are polymorphic pointers in disguise
  - Allow channels to access the member functions of channel
  - Always use **operator->** to access methods via a port
  - Port begins with P, Pointer begins with P

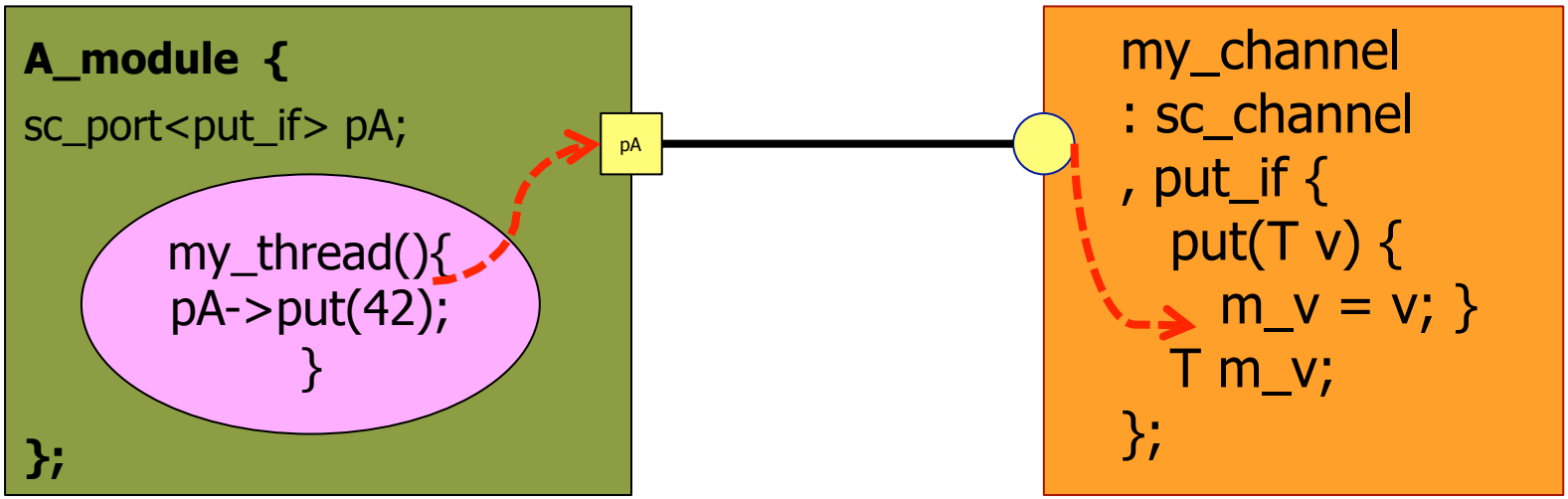


sc_port<>'s Connect to External Channels



sc_port<>'s Connect to External Channels

```
top_module {
A_module    a_inst;
My_channel ch_inst;
```




```
  top_module (…) {
     a_inst.pA.bind(ch_inst);
  }
};
```

sc_port<>'s Connect to Internal Channels

```
top_module {
  A_module* a_inst;
  B_module* ch_inst;

  top_module (...) {
     a_inst.pA.bind(b_inst.xpA);
  }
};
```

```
A_module {
sc_port<put_if> pA;

    my_thread(){
    pA->put(42);
        }

};
```

A

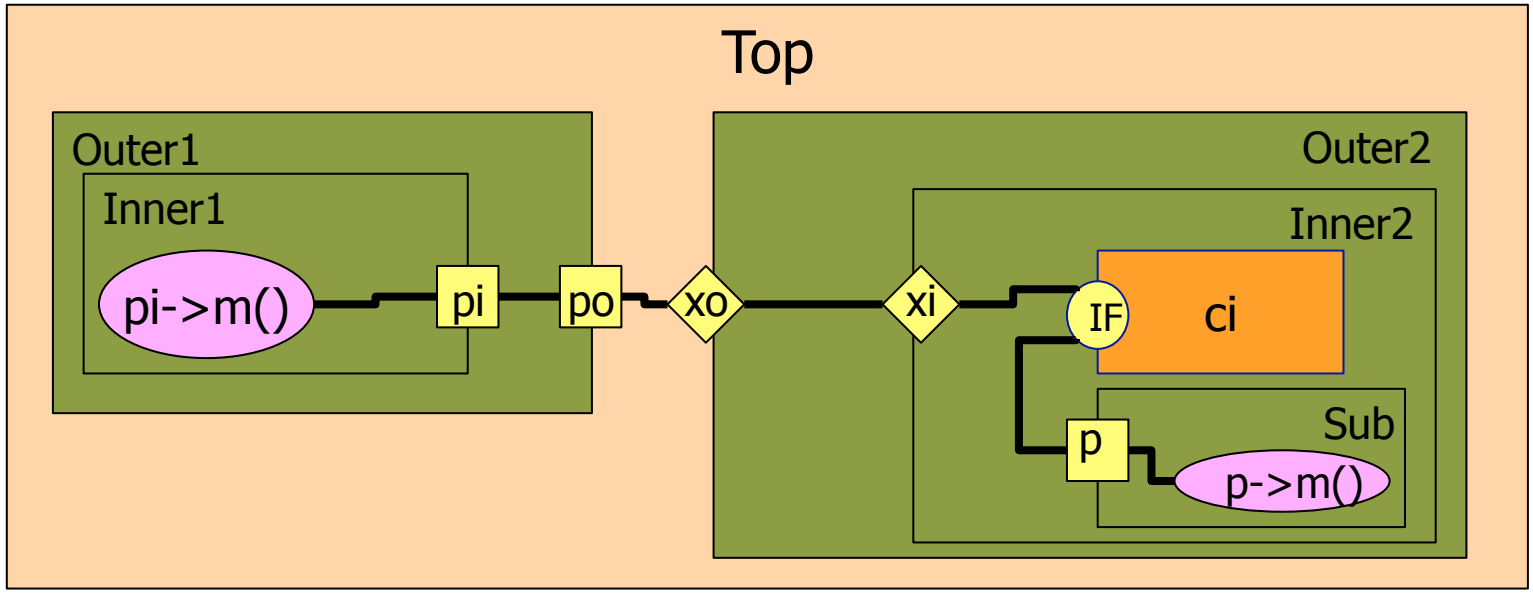
A

```
B_module : sc_module {
  sc_port<put_if> xpA;
  my_channel      ch_inst;

  B_module (...) {
    xpA.bind(ch_inst);
  }
};
```

```
my_channel
: sc_channel
, put_if {
   put(T v) {
     m_v = v; }
   T m_v;
};
```

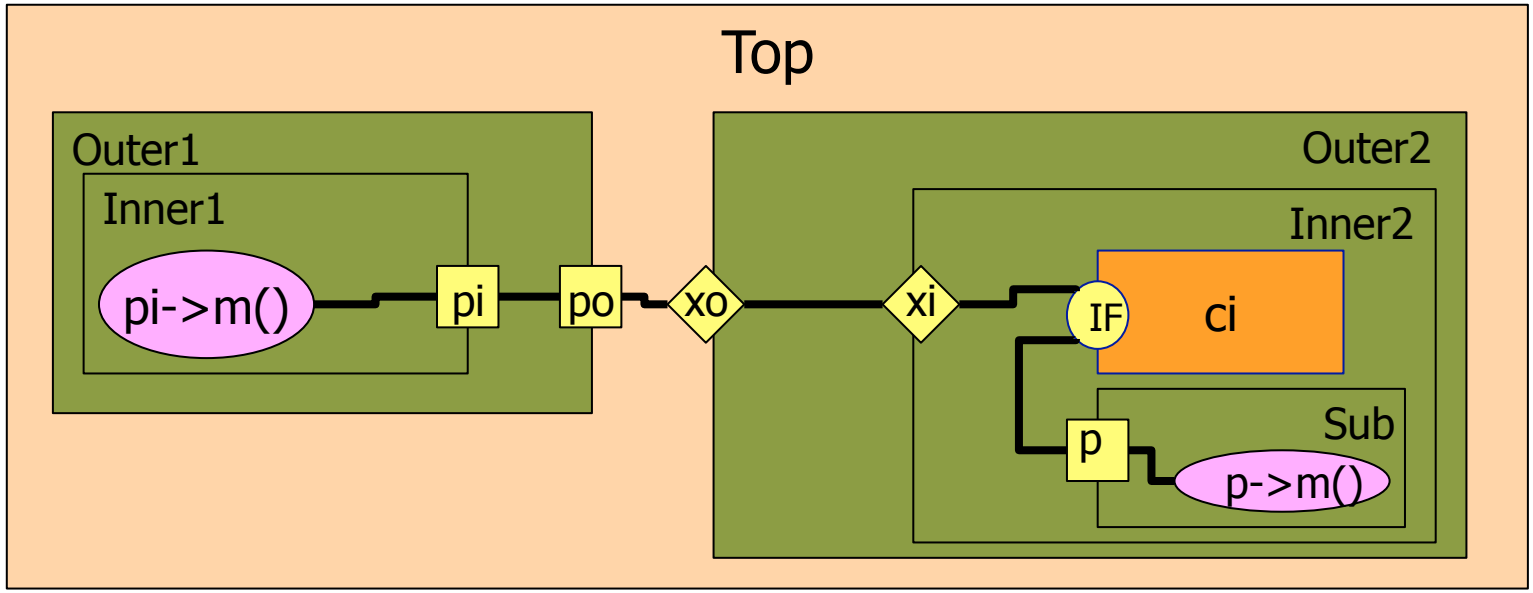
Where to connect

- Ports restricted as to what they can connect to
  - Process to port via method supported by port's interface
  - Port to port of same interface if hierarchically connected
  - Port to export of same interface
  - Export to export of same interface if hierarchically connected
  - Port externally or export internally to channel with interface



Connecting Ports

- Steps to using ports
  - **Declare** the port & channel
  - **Instantiate** the port & channel
  - **Bind** the port to the channel via bind() method
  - **Use** the port
- Let's examine each of these steps



Step 1 - Instantiate

```
SC_MODULE(Inner1) {
  SC_CTOR(Inner1);
  void Proc1(void);
  sc_port<IF> pi;
};
SC_MODULE(Outer1) {
  SC_CTOR(Outer1);
  Inner1 I1; //< Direct
  sc_port<IF> po;
};
SC_MODULE(Sub) {
  SC_CTOR(Sub);
  void Proc2(void);
  sc_port<IF> p;
};
```

```
SC_MODULE(Inner2) {
  SC_CTOR(Inner2);
  Chan c1;
  Sub* s1; //< Indirect
  sc_export<IF> xi;
};
SC_MODULE(Outer2) {
  SC_CTOR(Outer2);
  Inner2* I2;
  sc_export<IF> xo;
};
SC_MODULE(Top) {
  SC_CTOR(Top);
  Outer1* O1;
  Outer2* O2;
};
```

Step 2 - Bind

```
Outer1::Outer1(
sc_module_name inst_name
) : sc_module(inst_name)
, I1("I1")   // construct
{
   I1.pi.bind(po);  // connect
}
Top1::Top1(
sc_module_name inst_name
) :  (inst_name)
   ,O1(new Outer1("O1"))   //create
   ,O2(new Outer2("O2"))   //create
{
   O1->po.bind(O2->xo);  //connect
}
```

```
Inner2::Inner2(
sc_module_name inst_name
) : sc_module(inst_name)
,s1 ( new Sub("s1") )
{
   xi(c1);       // bind overload ()
   s1->p(c1); // bind overload ()
}
Outer2::Outer2(
sc_module_name inst_name
) : sc_module(inst_name)
,I2(new Inner2("I2"))   //create
{
   xo(I2->xi);  // bind
}
```

Step 3 - Use

```
SC_HAS_PROCESS(Inner1);
Inner1::Inner1(
   sc_module_name inst_name
) : sc_module(inst_name)
{
   SC_THREAD(Proc1);
}
void Inner1::Proc1(void) {
   p1->m();
}

struct IF : sc_interface
{
   virtual void m(void) = 0;
};
```

```
SC_HAS_PROCESS(Sub);
Sub::Sub(
sc_module_name inst_name
) : sc_module(inst_name)
{
   SC_METHOD(Proc2);
}
void Sub::Proc2(void) {
   p->m();
}
struct Chan
: sc_channel, IF
{
   Chan(
      sc_module_name inst_name);
   void m(void);
};
```

7.

Tools (simulation, synthesis)

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

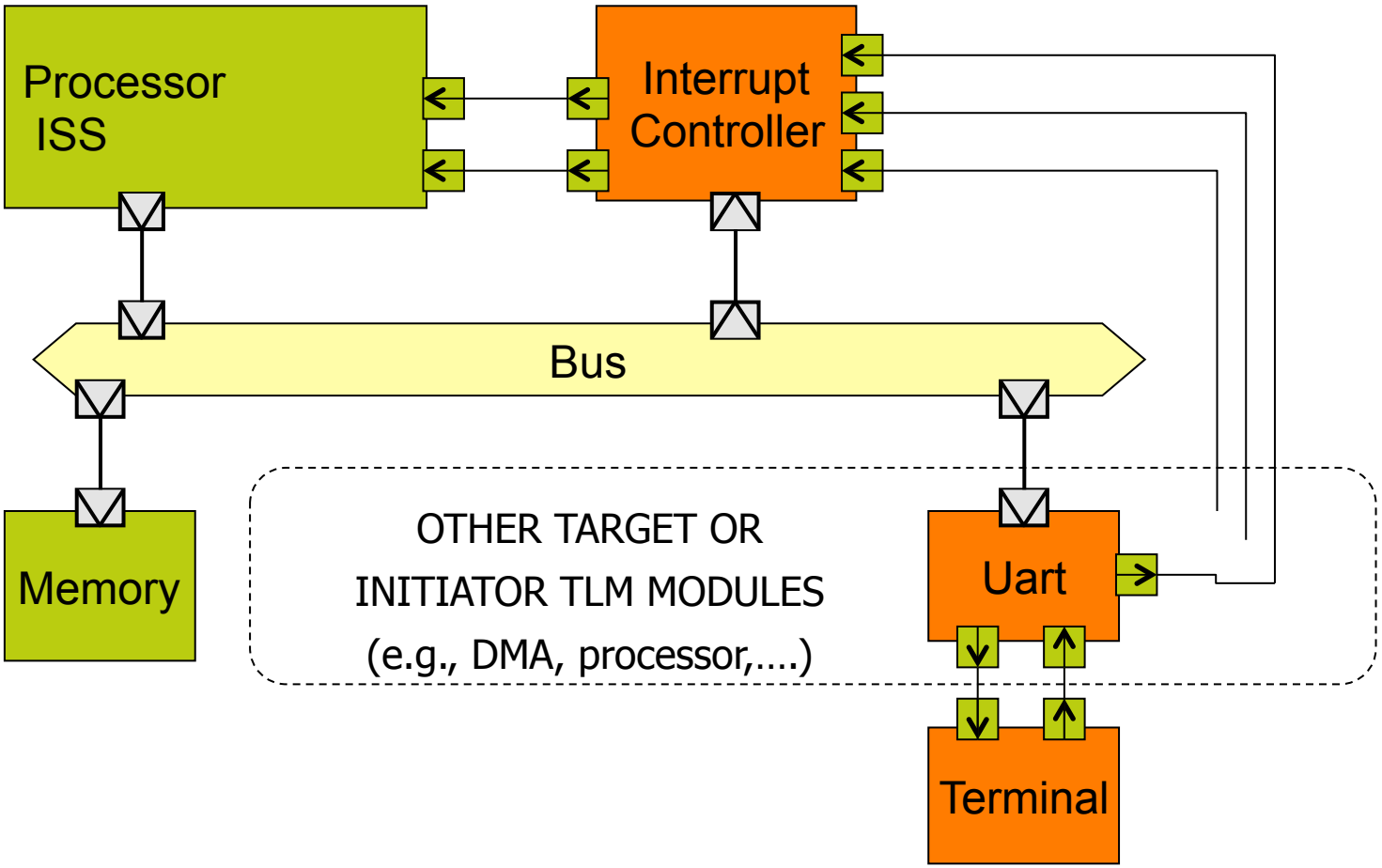
- SystemC is an **open source** and the latest versions can be downloaded free from **systemc.org (accellera.org)**
- SystemC can be installed in various platforms, including **ModelSim**, VC++, Linux G++, Cygwin etc. SystemC community at systemc.org discusses various issues with different tools and operating systems.
- SystemC designs can be **synthesized to gate level HDLs** using various synthesis tools, examples **Xilinx Vivado_HLS,** Mentor Catapult C++, Cadence CtoS, Forte Cynthesizer 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

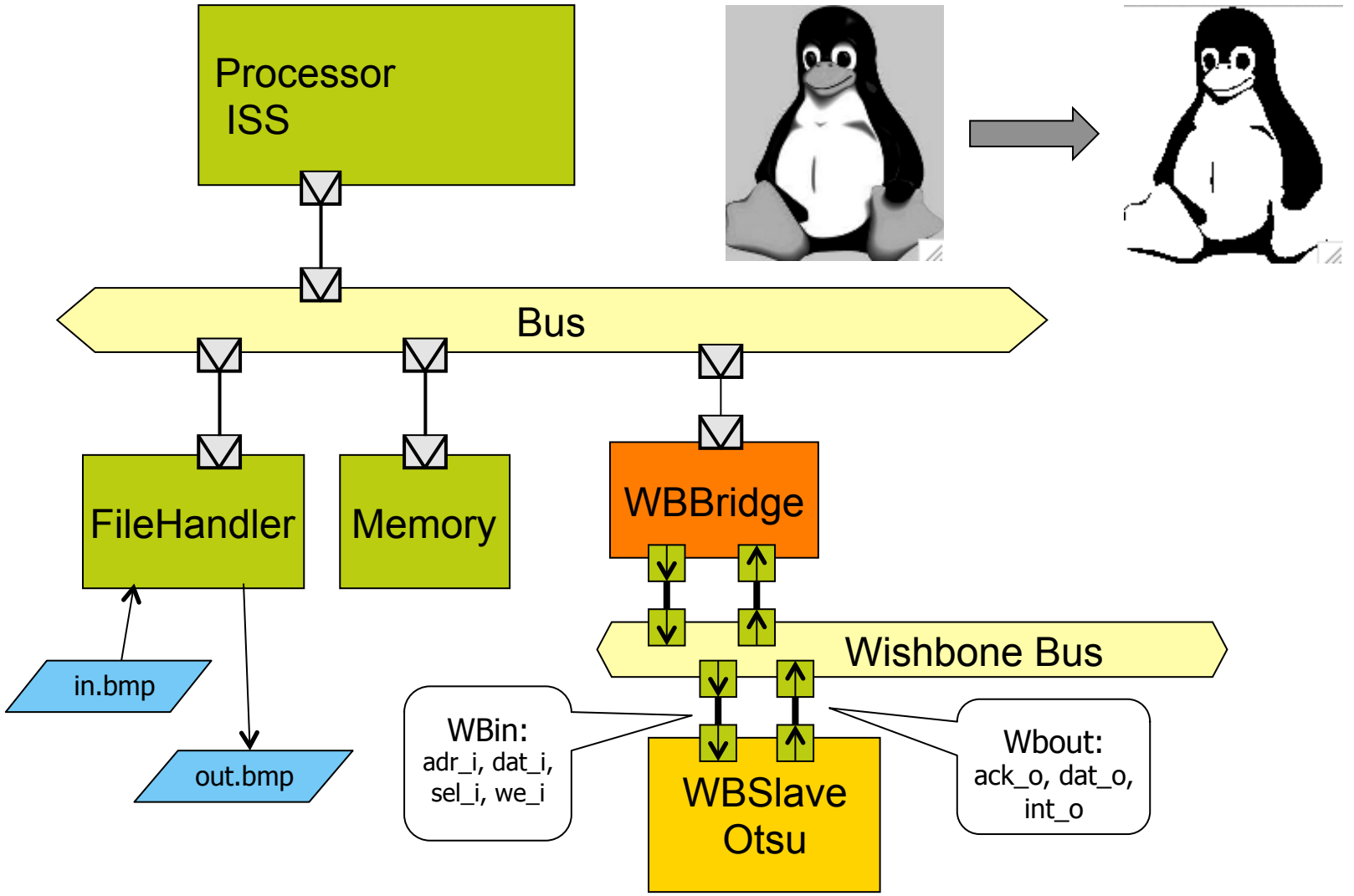
- Black, D.C., Donovan, J., “SystemC : from the Ground Up", Kluwer Academics, 2004.
- J. Bhasker, “A SytemC Primer”
- Thortsten Grötker e.a., “SystemC Design with SystemC”

Website, www.systemc.org
- Website, www.doulos.com
- Website, www.forteds.com
- C++ tutorial: www.cplusplus.com/doc/tutorial

SimSoc Platform



Otsu Platform



RAM_32s Platform