SystemC

M. Barış Uyar

What Is SystemC?

- A subset of C++ that models/specifies synchronous digital hardware
  - Processes (for concurrency)
  - Clocks (for time)
  - Hardware (finite) data types [bit vectors, 4-valued logic, fixed-point types, arbitrary precision integers] and infinite data types
  - Waiting and watching (for reactivity)
  - Modules, ports, signals (for hierarchy)
- SystemC provides
  - Modelling in a higher level of abstraction
  - Faster simulation times
  - Testing the behaviour of the entire chip before production
  - Allows to make an *"Executable Specification"*
  - Synthesis support??!!

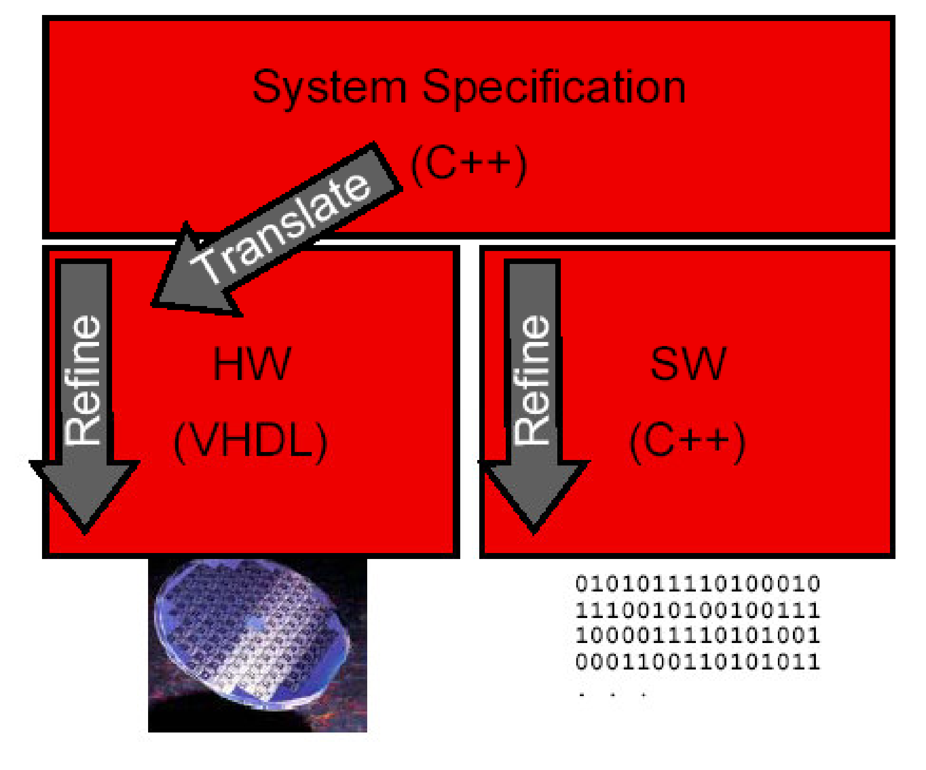
Where to use?

1. Modelling in a higher level of abstraction

Algorithmic model	✓ Functional verification ✓ Algorithm validation
UnTimed Functional (UTF) model	
Timed Functional (TF) model	✓ Coarse benchmarking ✓ Application SW development ✓ Architectural analysis
Bus Cycle Accurate (BCA) model	
Cycle Accurate (CA) model	✓ Detailed benchmarking ✓ Driver development ✓ Microarchitectural analysis
Register Transfer Level (RTL) model	

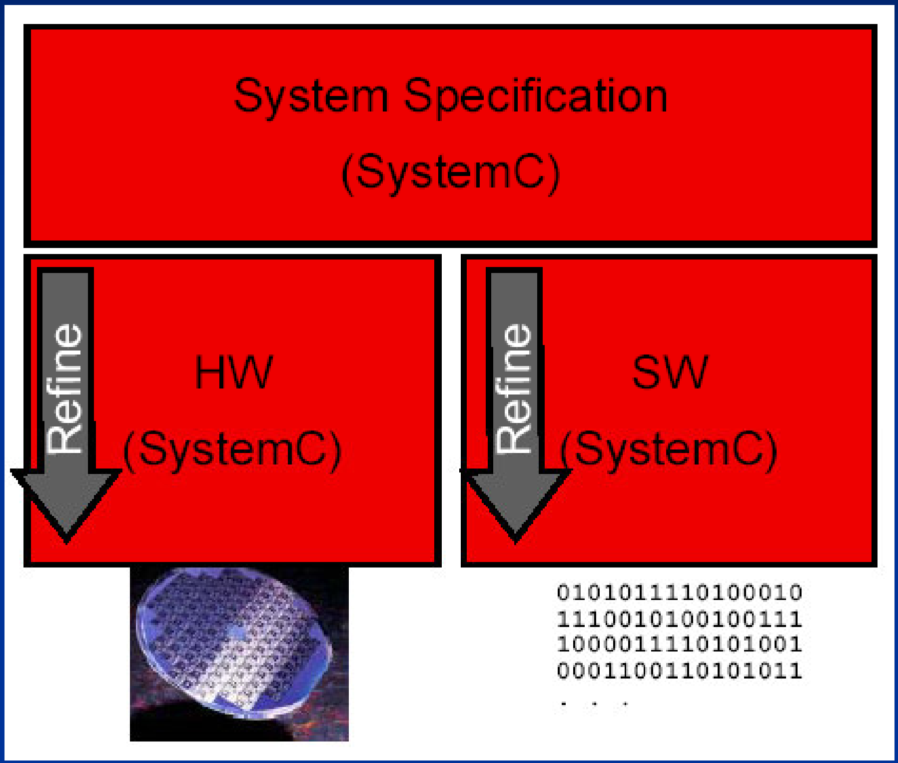
2. Synthesis

Why not just C++

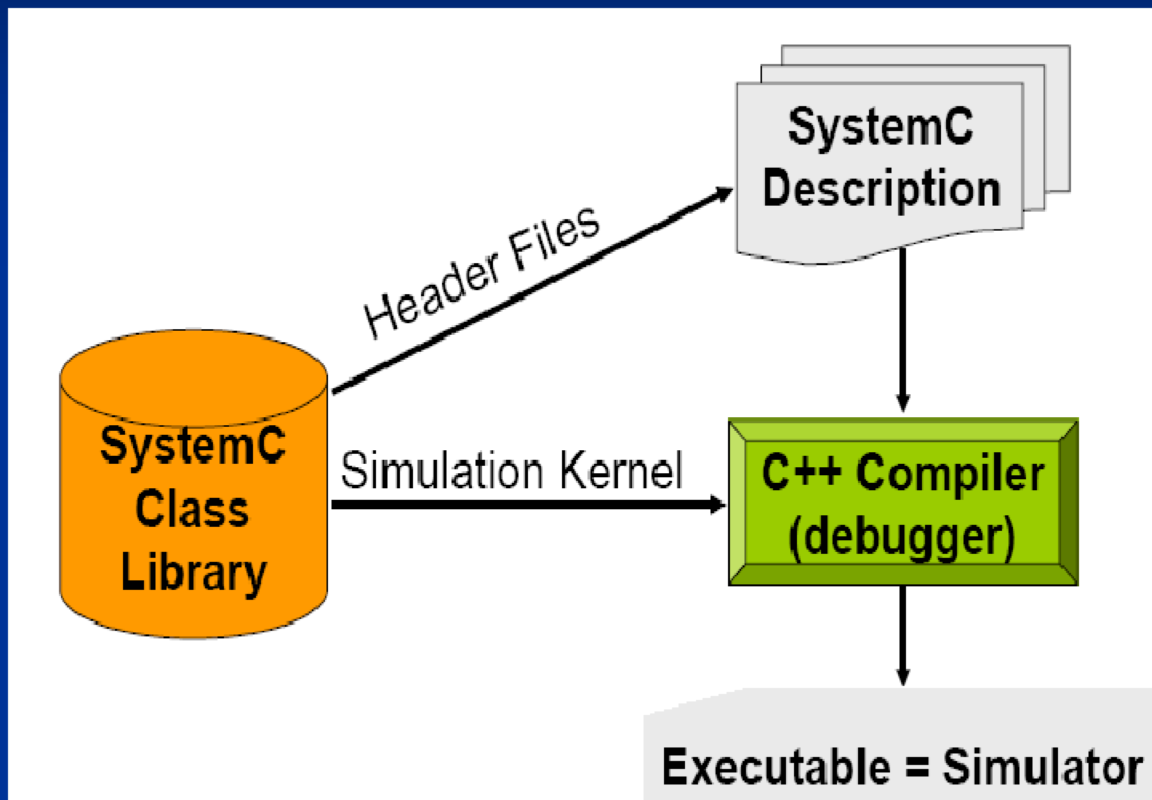


• Concurrency support is missing (HW is inherently parallel)
No notion of time (clock, delays)
Communication model is very different from actual HW model (signals)
Weak/complex reactivity to events
Missing data types (logic values, bit vectors, fixed point math)

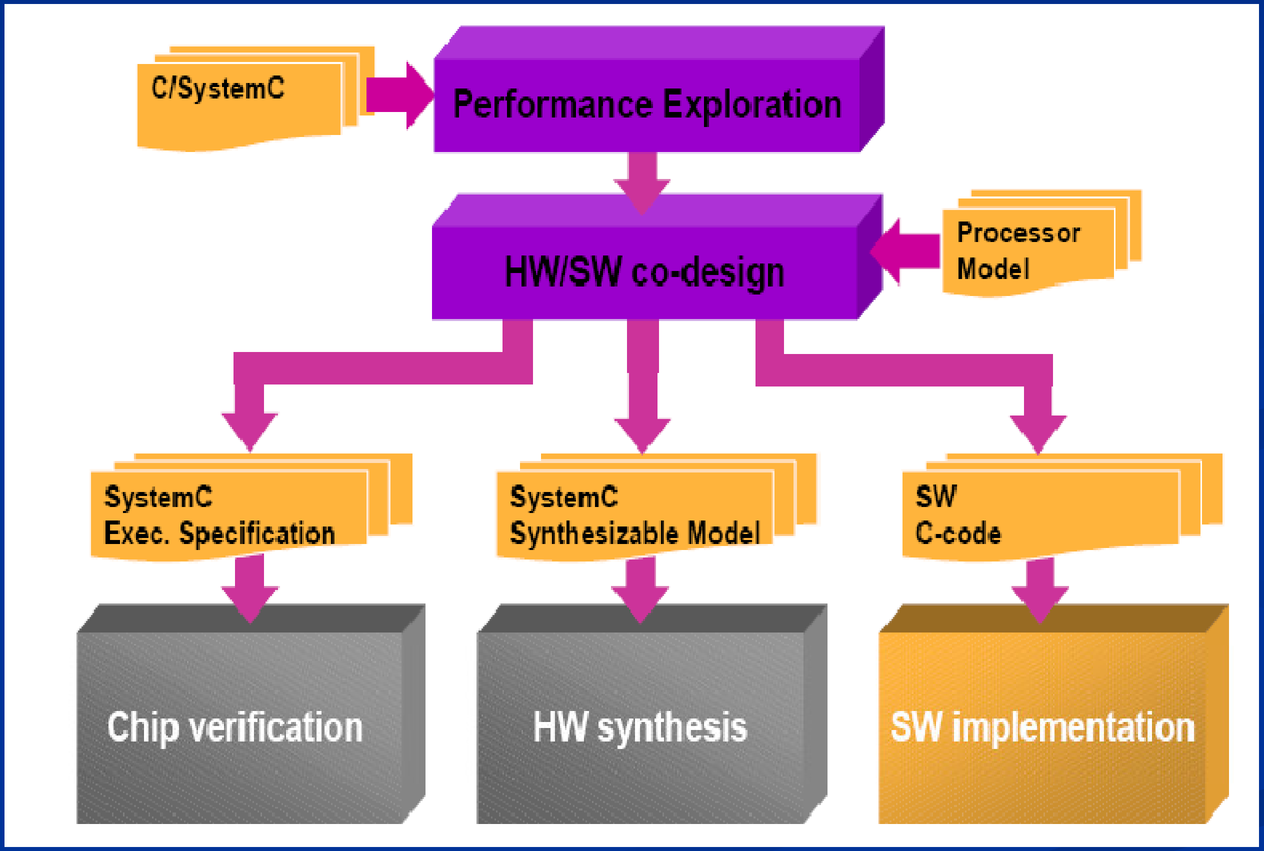
SystemC



- **… C++ *extensions*!**
- New library (**libsystemc.a**) providing additional functionality
- Building upon C++ features (inheritance!) and data types to better express HW behavior
- "SystemC" HW-modeling code is actually C++ code and can be freely mixed with plain C++


SystemC
Description
Header Files
SystemC
Class
Library
Simulation Kernel
C++ Compiler
(debugger)
Executable = Simulator

SystemC usage models



Standard Methodology for ICs

- System-level designers write a C or C++ model
  - Written in a stylized, hardware-like form
  - Sometimes refined to be more hardware-like
- C/C++ model simulated to verify functionality
- Model given to Verilog/VHDL coders
- Verilog or VHDL specification written
- Models simulated together to test equivalence
- Verilog/VHDL model synthesized

Designing Big Digital Systems

- Every system company was doing this differently
- Every system company used its own simulation library

- "Throw the model over the wall" approach makes it easy to introduce errors

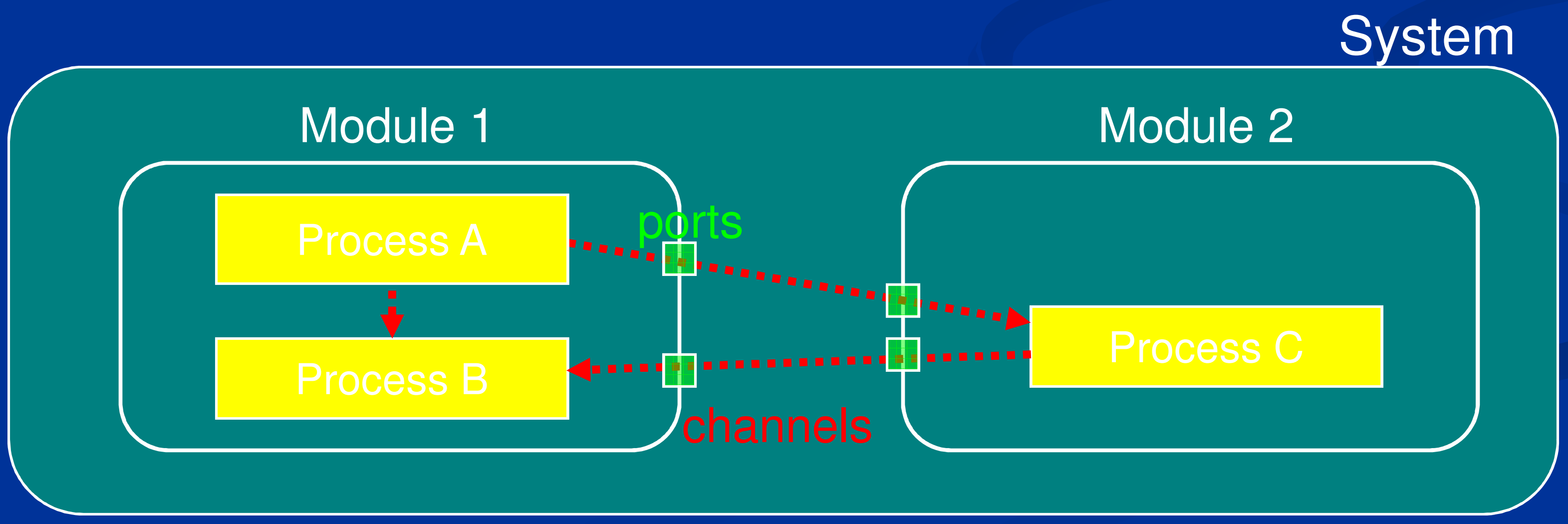
- Problems:
  - System designers don't know Verilog or VHDL
  - Verilog or VHDL coders don't understand system design

Idea of SystemC

- C and C++ are being used as ad-hoc modeling languages
- Why not formalize their use?
- Why not interpret them as hardware specification languages just as Verilog and VHDL were?

Quick Overview

- A SystemC program consists of module definitions plus a top-level function that starts the simulation
- Modules contain processes (C++ methods) and instances of other modules
- Ports on modules define their interface
  - Rich set of port data types (hardware modeling, etc.)
- Signals in modules convey information between instances
- Clocks are special signals that run periodically and can trigger clocked processes
- Rich set of numeric types (fixed and arbitrary precision numbers)

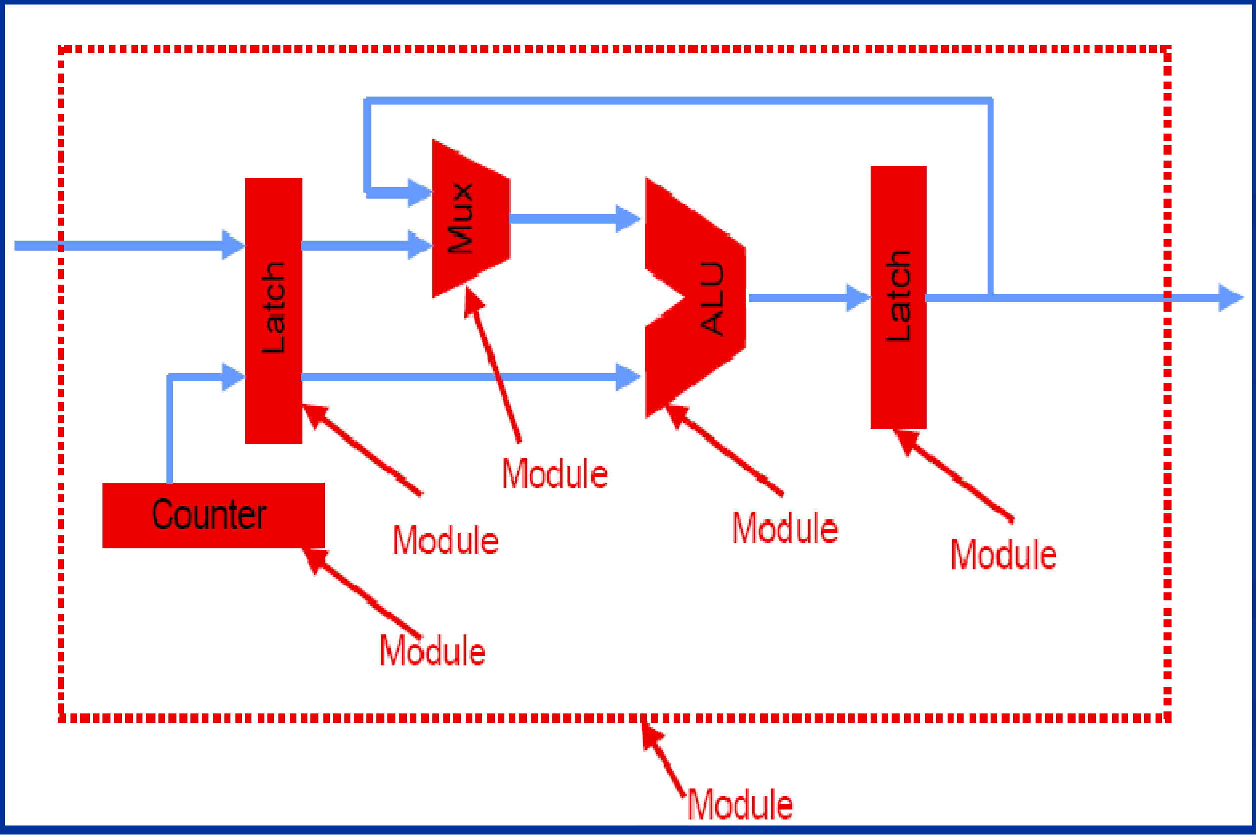

System
Module 1
Module 2
Process A
Process B
Process C
ports
channels

Modules

- Hierarchical entity
- Similar to Verilog's module
- Actually a C++ class definition
- Simulation involves
  - Creating objects of this class
  - They connect themselves together
  - Processes in these objects (methods) are called by the scheduler to perform the simulation

Modules

- map functionality of HW/SW blocks
- derived from SystemC class **sc_module**
- represent the basic building block of every system
- modules can contain a whole hierarchy of sub-modules
- and provide private variables/signals



- **Communication:** Modules can interface to each other via **ports/interfaces/channels**
- **Functionality:** achieved by means of **processes**

Modules

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

SC_MODULE(my_module)
{
	//port declarations
	//internal data
	//process declarations

	SC_CTOR(my_module)
	{
		//map processes to member functions
		//sensitivity lists
		//initialization code
	}
};
```

Basically a C++ class with member variables, member functions and constructor!

Ports

- Define the interface to each module
- Channels through which data is communicated
- Port consists of a direction
  - input sc_in
  - output sc_out
  - bidirectional sc_inout
- and any C++ or SystemC type

Ports

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

SC_MODULE(my_module)
{
      sc_in<bool> id;
      sc_in<sc_uint<3> > in_a;
      sc_in<sc_uint<3> > in_b;
      sc_out<sc_uint<3> > out_c;
      //process declarations
      SC_CTOR(my_module)
      {
            //process configuration
            //initialization code
      }
};
```

Ports can have any of the SystemC data types! sc_uint<3> is a 3-bit unsigned int

Signals

- Convey information between modules within a module
- Directionless: module ports define direction of data transfer
- Type may be any C++ or built-in type

Signals

```
SC_MODULE(mymod) {
 /* port definitions */
 sc_signal<sc_uint<32> > s1, s2;
 sc_signal<bool> reset;

  /* … */
  SC_CTOR(mymod) {
    /* Instances of modules that connect to the signals */
  }
};
```

Instances of Modules

- Each instance is a pointer to an object in the module

```
SC_MODULE(mod1) { … };
SC_MODULE(mod2) { … };
SC_MODULE(foo) {
  mod1* m1;
  mod2* m2;
  sc_signal<int> a, b, c;
  SC_CTOR(foo) {
    m1 = new mod1(“i1”);  (*m1)(a, b, c);
    m2 = new mod2(“i2”);  (*m2)(c, b);
  }
};
```

Connect instance’s ports to signals

Signal instanciation and binding

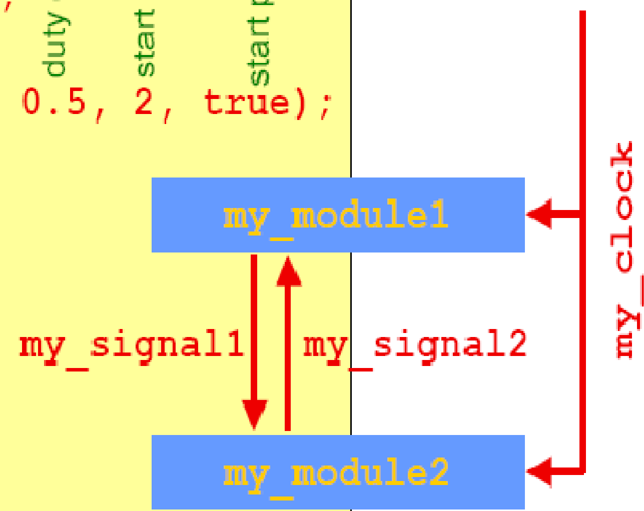
```
my_module_type_1 my_module_1;
my_module_type_2 my_module_2;

sc_signal<sc_uint<8> > my_signal1;
sc_signal<bool> my_signal2;
sc_clock my_clock("my_clock", 20, 0.5, 2, true);

my_module_1.clock_port(my_clock);
my_module_1.out_port(my_signal1);
my_module_1.in_port(my_signal2);

my_module_2.clock_port(my_clock);
my_module_2.out_port(my_signal2);
my_module_2.in_port(my_signal1);
```

name period duty cycle start time start phase



Processes

- They provide module functionality
- Implemented as member functions
- Three kinds of processes available:
  - **SC_METHOD**
  - **SC_THREAD**
  - **SC_CTHREAD**

- All of the processes in the design run concurrently
- Code inside of every process is sequential

METHOD Processes

- Sensitive to any change on input ports
- Usually used to model purely combinational logic (*i.e.* NORs, NANDs, muxes, …)
- Cannot be suspended. All of the function code is executed every time the **SC_METHOD** is invoked
- Does not remember internal state among invocations (unless explicitly kept in member variables)

Method

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

SC_MODULE(my_module)
{
  sc_in<bool> id;
  sc_in<sc_uint<3> > in_a;
  sc_in<sc_uint<3> > in_b;
  sc_out<sc_uint<3> > out_c;
  void my_method();
  SC_CTOR(my_module) {
    SC_METHOD(my_method);
    sensitive << in_a
              << in_b;
  }
};
```

```
//my_module.cpp
#include "my_module.h"


void my_module::my_method()
{
  if (id.read())
    out_c.write(in_a.read());
  else
    out_c.write(in_b.read());
};
```

A mux?...

...ALMOST!!

Thread Processes

- Adds the ability to be suspended to **SC_METHOD** processes by means of **wait()** calls (and derivatives)
- Still has a sensitivity list. **wait()** returns when a change is detected on a port in the sensitivity list
- Remembers its internal state among invocations (*i.e.* execution resumes from where it was left)
- Very useful for clocked systems, memory elements, multi-cycle behavior
- Imposes a heavier load onto the SystemC scheduler (slower simulations) due to context switches and state tracking

Thread

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

SC_MODULE(my_module)
{
  sc_in<bool> id;
  sc_in<bool> clock;
  sc_in<sc_uint<3> > in_a;
  sc_in<sc_uint<3> > in_b;
  sc_out<sc_uint<3> > out_c;
  void my_thread();
  SC_CTOR(my_module)
  {
    SC_THREAD(my_thread);
    sensitive << clock.pos();
  }
};
```

```
//my_module.cpp
#include "my_module.h"

void my_module::my_thread()
{
  while(true)
  {
    if (id.read())
      out_c.write(in_a.read());
    else
      out_c.write(in_b.read());
    wait();
  }
};
```

Again almost a mux…

SystemC Types

- SystemC programs may use any C++ type along with any of the built-in ones for modeling systems

- SystemC custom types
  - Scalar: `sc_bit` (*i.e.* `bool`), `sc_logic` (*i.e.* 01XZ)
  - Integer: `sc_int`, `sc_uint`, `sc_bigint`, `sc_biguint`
  - Bit and logic vector: `sc_bv`, `sc_lv`
  - Fixed point: `sc_fixed`, `sc_ufixed`, `sc_fix`, `sc_ufix`
- Special operators
  - bit select: `x[i]`
  - part select: `x.range(4, 2)`
  - concatenation: `(x.range(2, 1), y)`
  - or reduction: `x.or_reduce()`

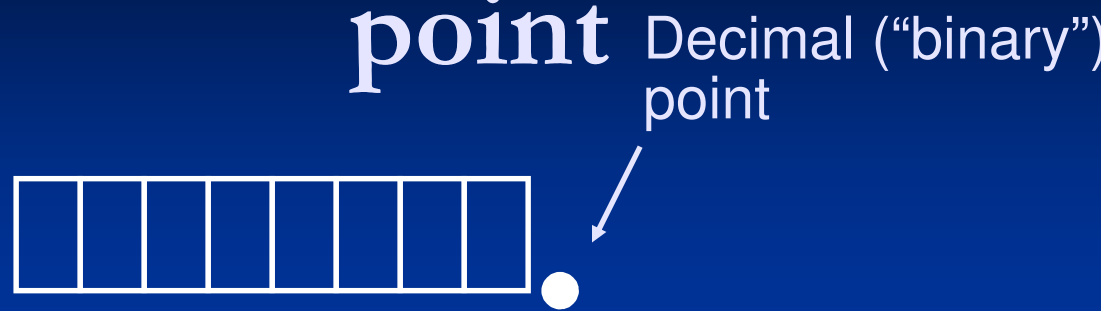
SystemC Built-in Types

- sc_bit, sc_logic
  - Two- and four-valued single bit
- sc_int, sc_unint
  - 1 to 64-bit signed and unsigned integers
- sc_bigint, sc_biguint
  - arbitrary (fixed) width signed and unsigned integers
- sc_bv, sc_lv
  - arbitrary width two- and four-valued vectors
- sc_fixed, sc_ufixed
  - signed and unsigned fixed point numbers

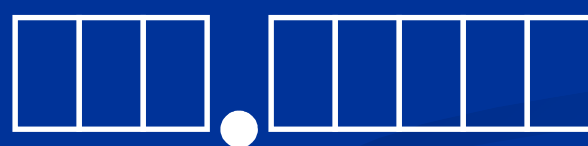
Fixed and Floating Point Types

- Integers
  - Precise
  - Manipulation is fast and cheap
  - Poor for modeling continuous real-world behavior
- Floating-point numbers
  - Less precise
  - Better approximation to real numbers
  - Good for modeling continuous behavior
  - Manipulation is slow and expensive
- Fixed-point numbers
  - Worst of both worlds
  - Used in many signal processing applications

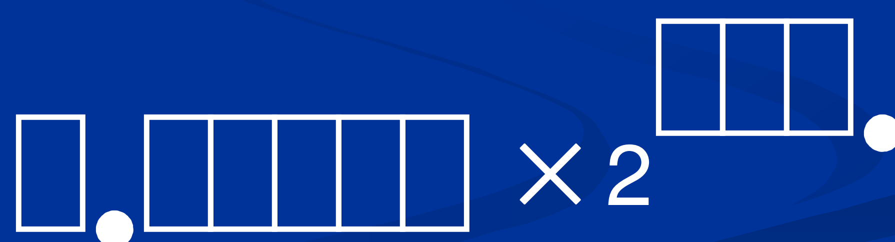
Integers, Floating-point, Fixed-point



- Integer



- Fixed-point



- Floating-point

Using Fixed-Point Numbers

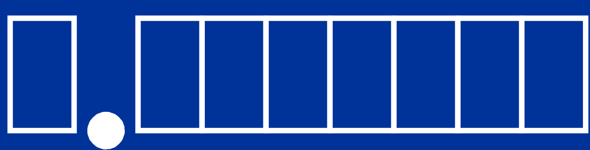
- High-level models usually use floating-point for convenience
- Fixed-point usually used in hardware implementation because they're much cheaper
- Problem: the behavior of the two are different
  - How do you make sure your algorithm still works after it's been converted from floating-point to fixed-point?

- SystemC's fixed-point number classes facilitate simulating algorithms with fixed-point numbers

SystemC’s Fixed-Point Types

- sc_fixed<8, 1, SC_RND, SC_SAT> fpn;

- 8 is the total number of bits in the type
- 1 is the number of bits to the left of the decimal point
- SC_RND defines rounding behavior
- SC_SAT defines saturation behavior

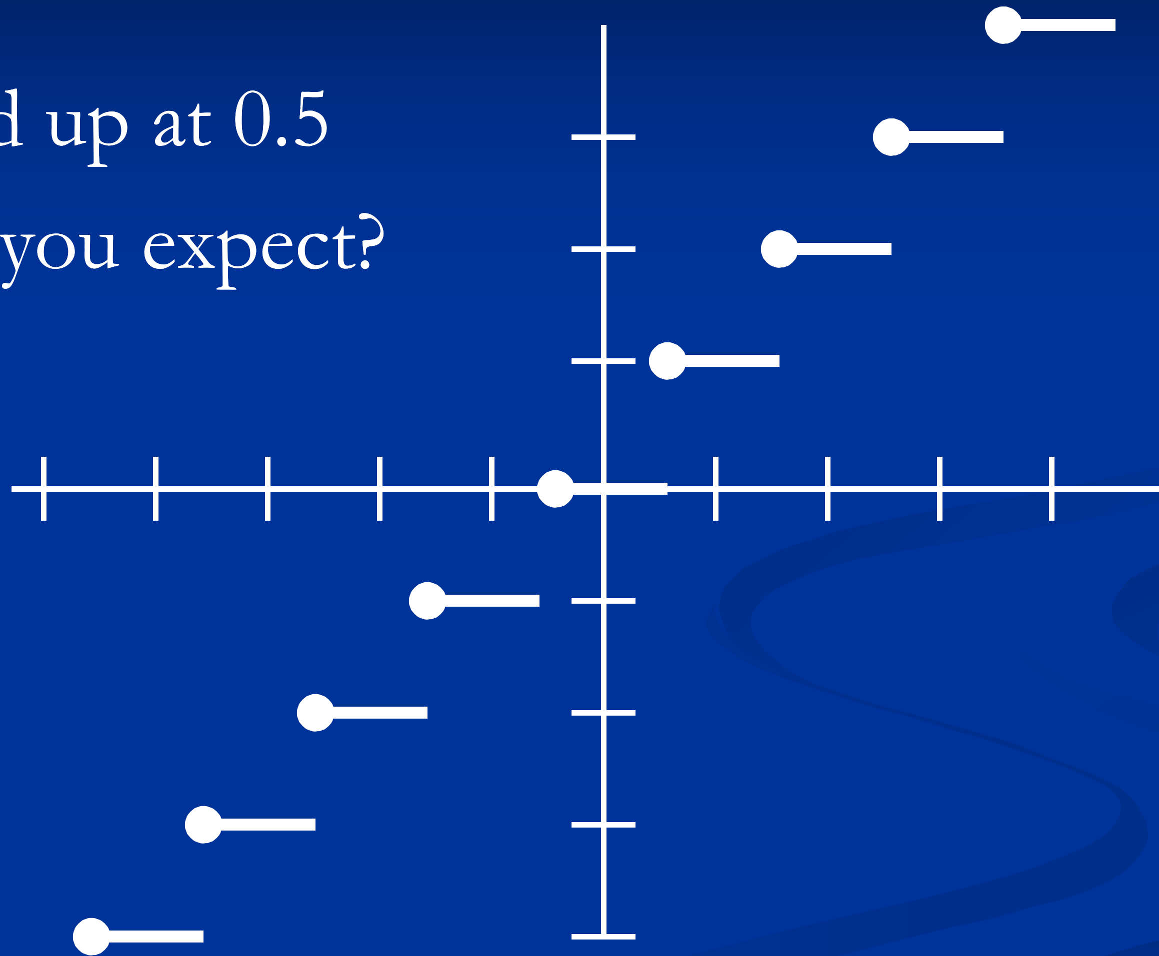


Rounding

- What happens when your result doesn't land exactly on a representable number?
- Rounding mode makes the choice

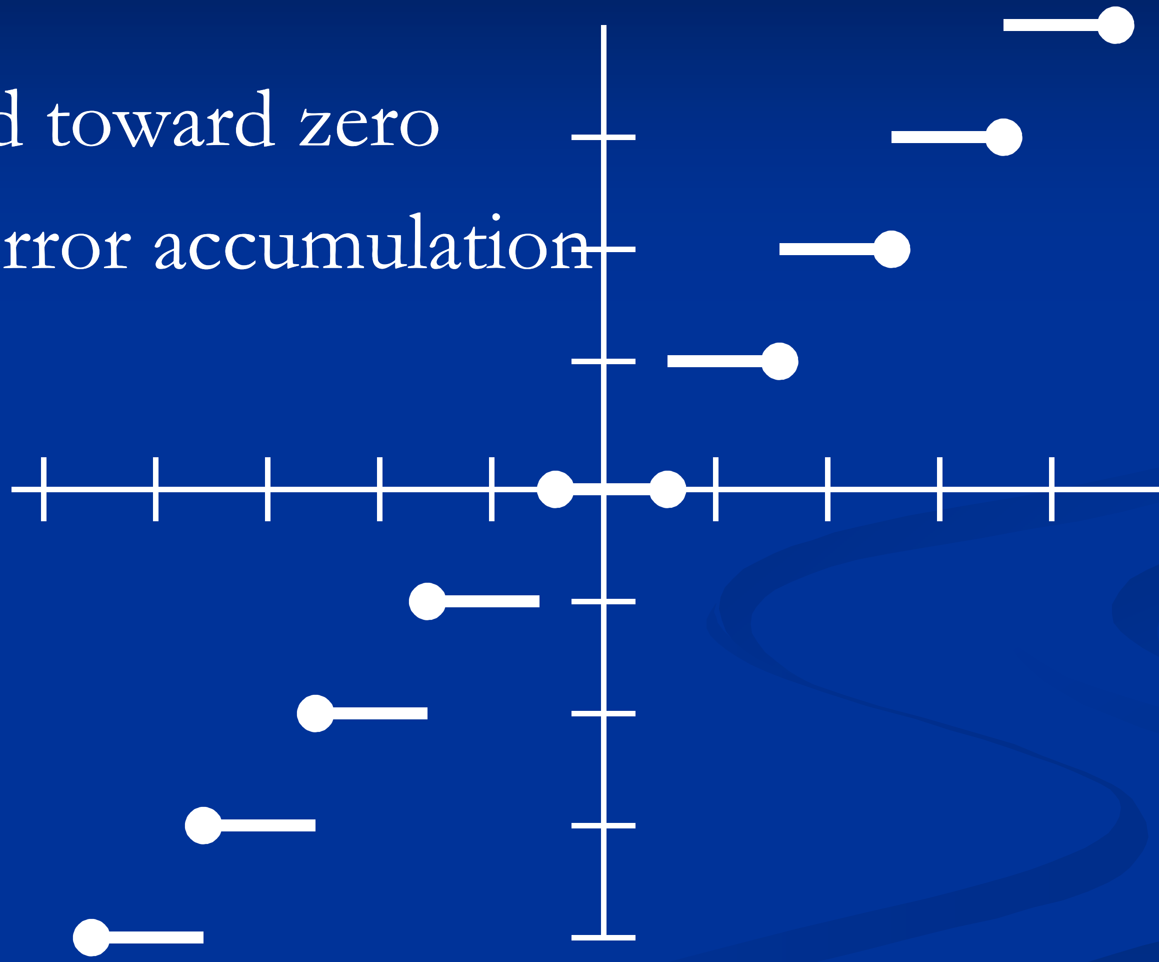
SC_RND

- Round up at 0.5
- What you expect?



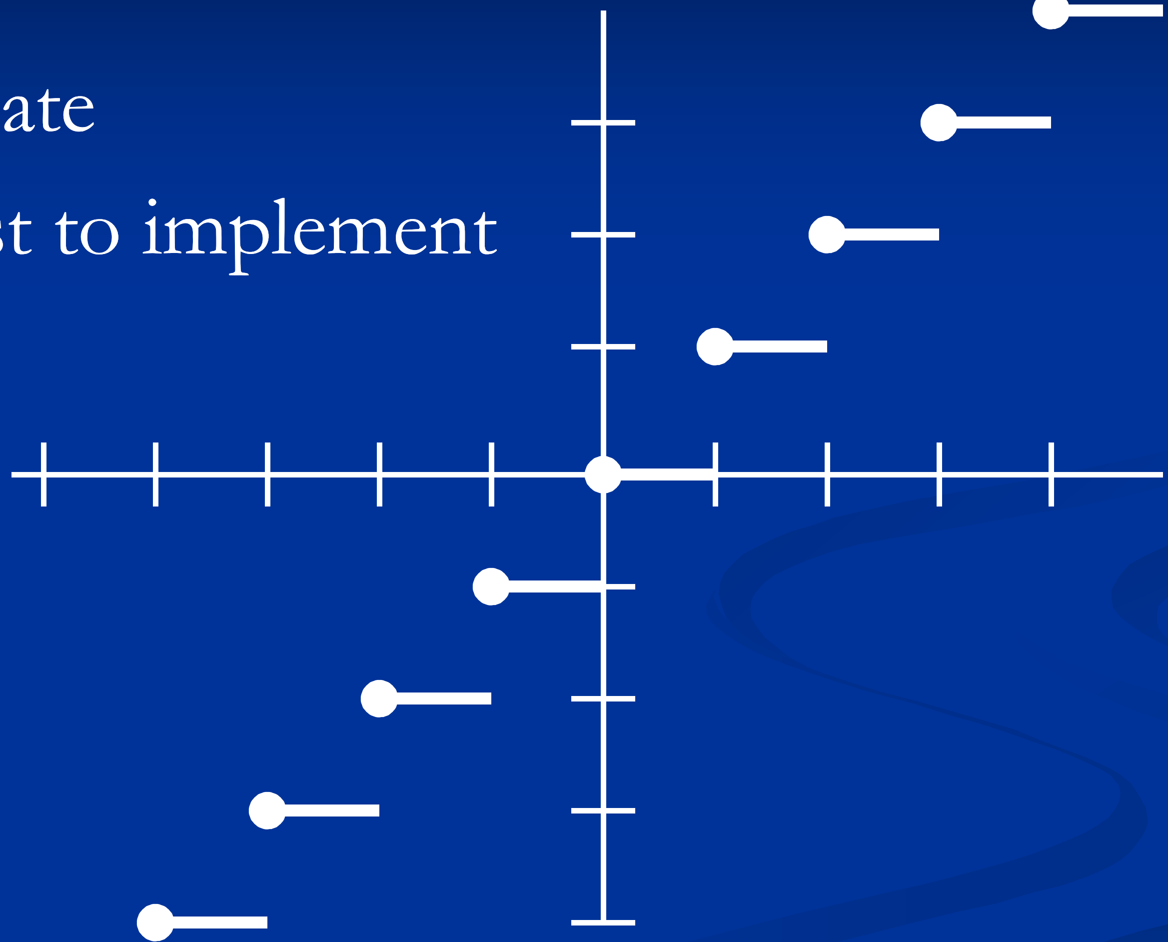
SC_RND_ZERO

- Round toward zero
- Less error accumulation



SC_TRN

- Truncate
- Easiest to implement

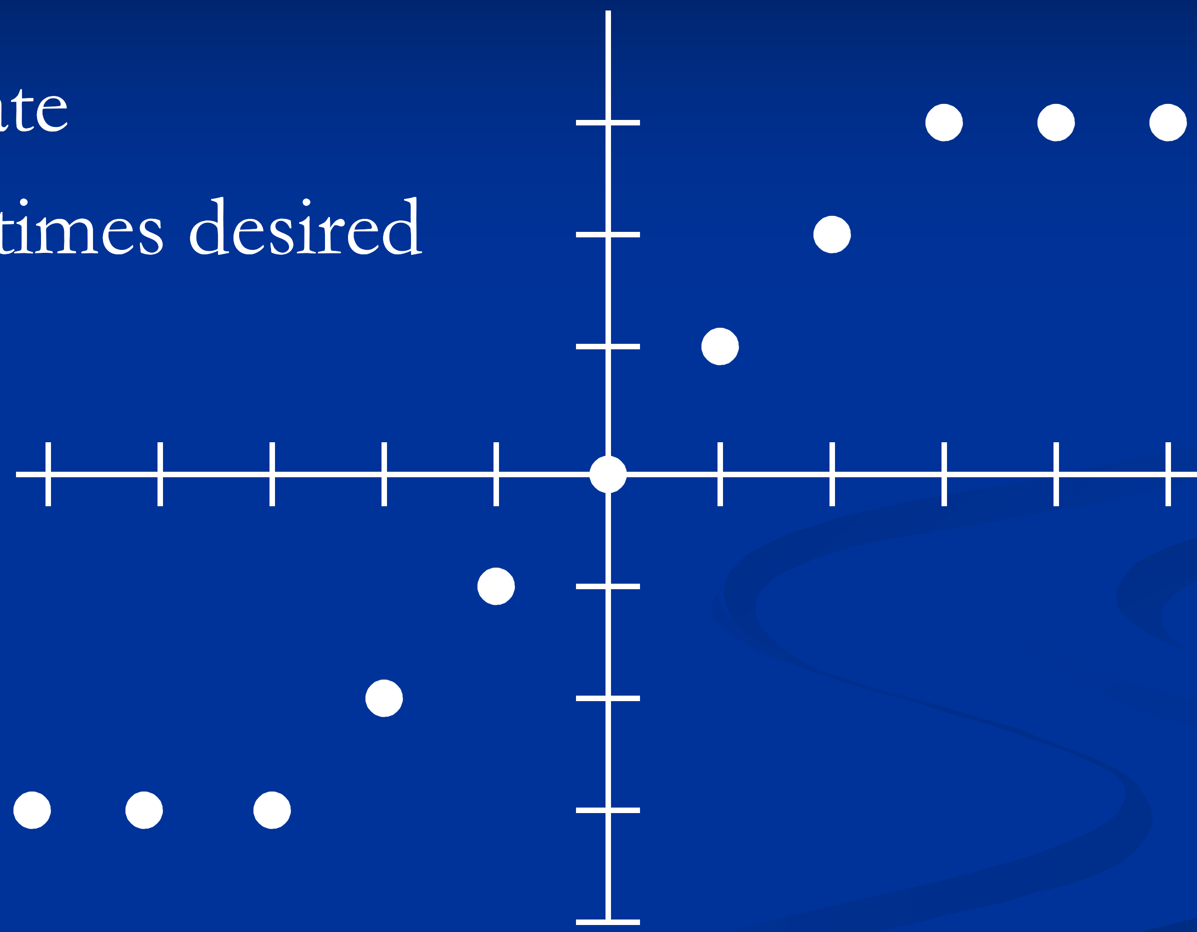


Overflow

- What happens if the result is too positive or too negative to fit in the result?
- Saturation? Wrap-around?
- Different behavior appropriate for different applications

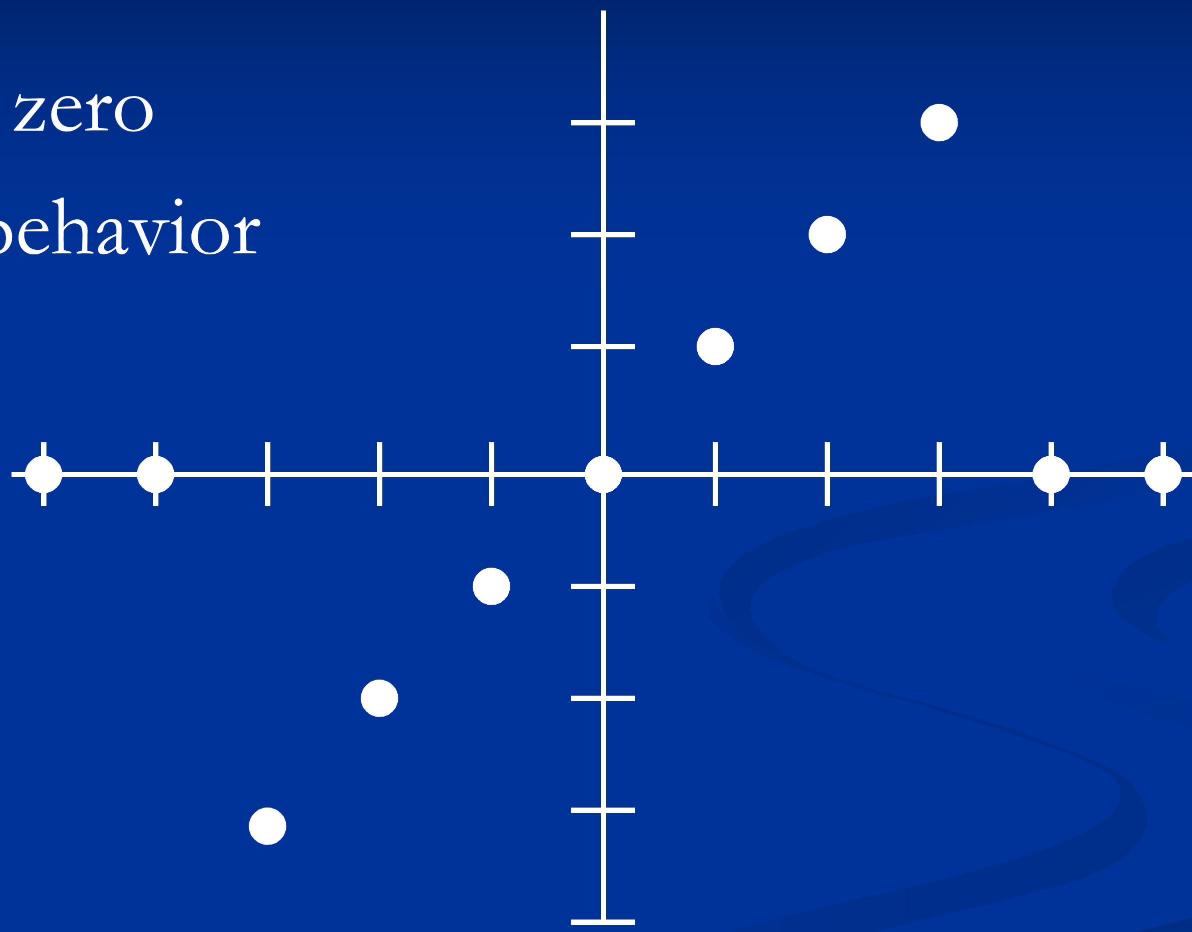
SC_SAT

- Saturate
- Sometimes desired



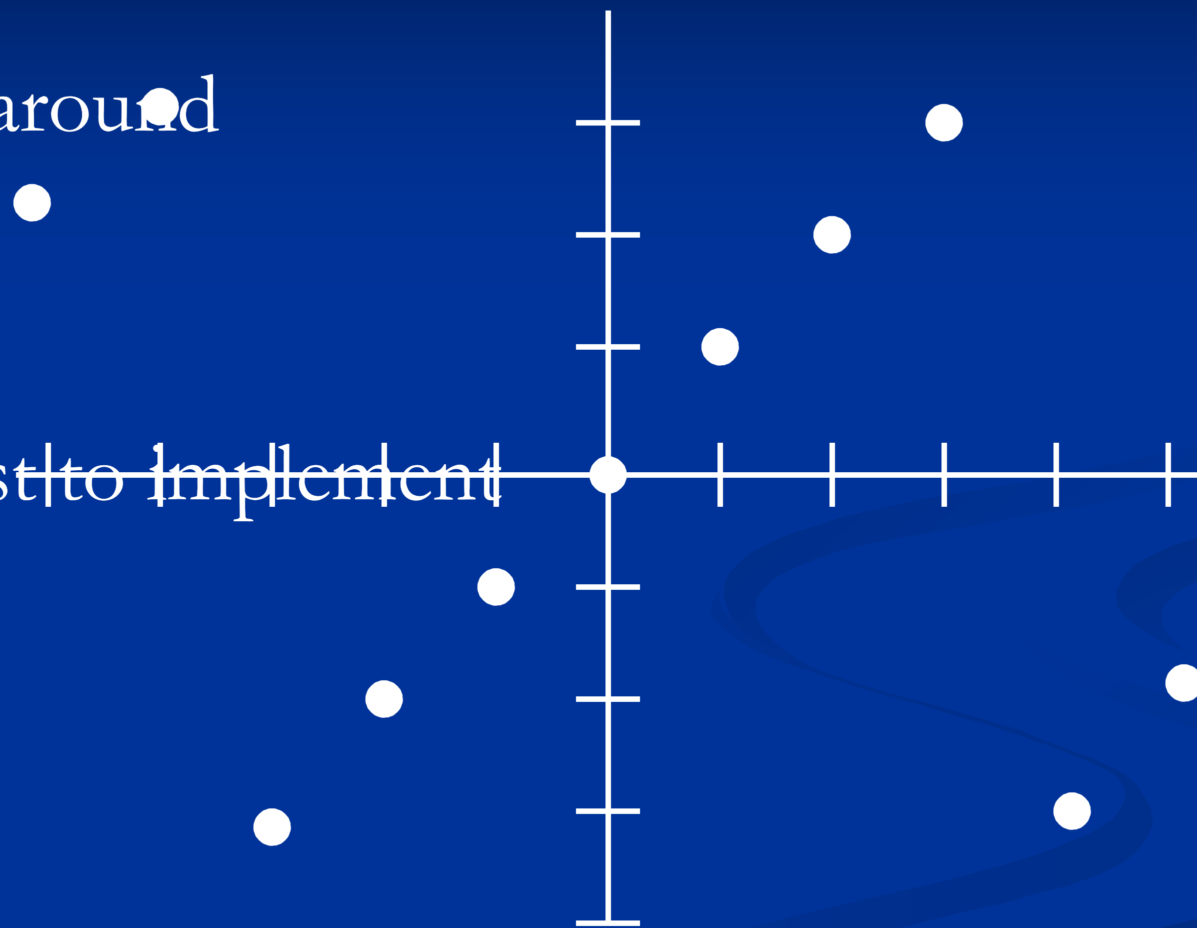
SC_SAT_ZERO

- Set to zero
- Odd behavior

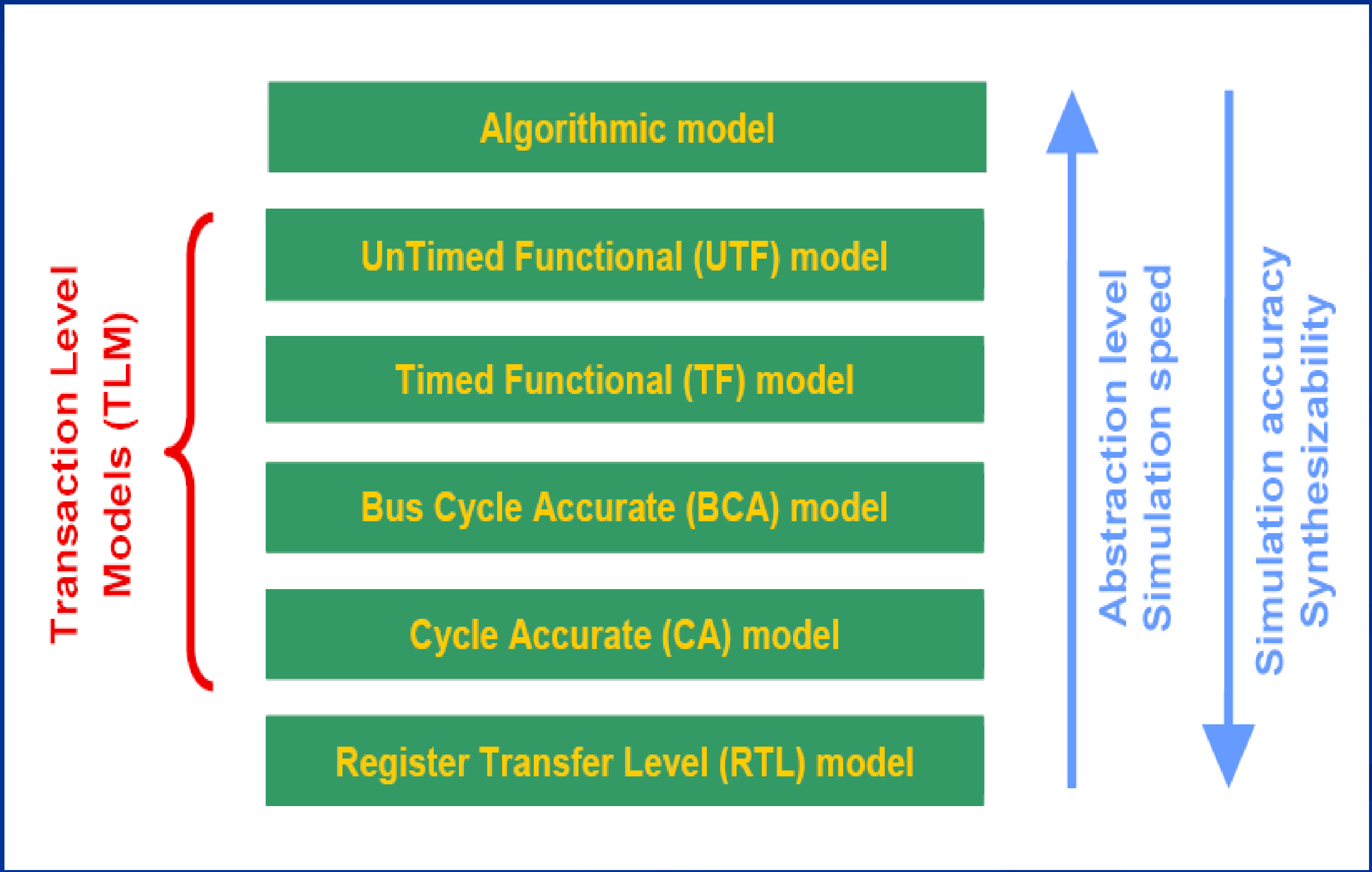


SC_WRAP

- Wraparound
- Easiest to Implement



Layers of HW design

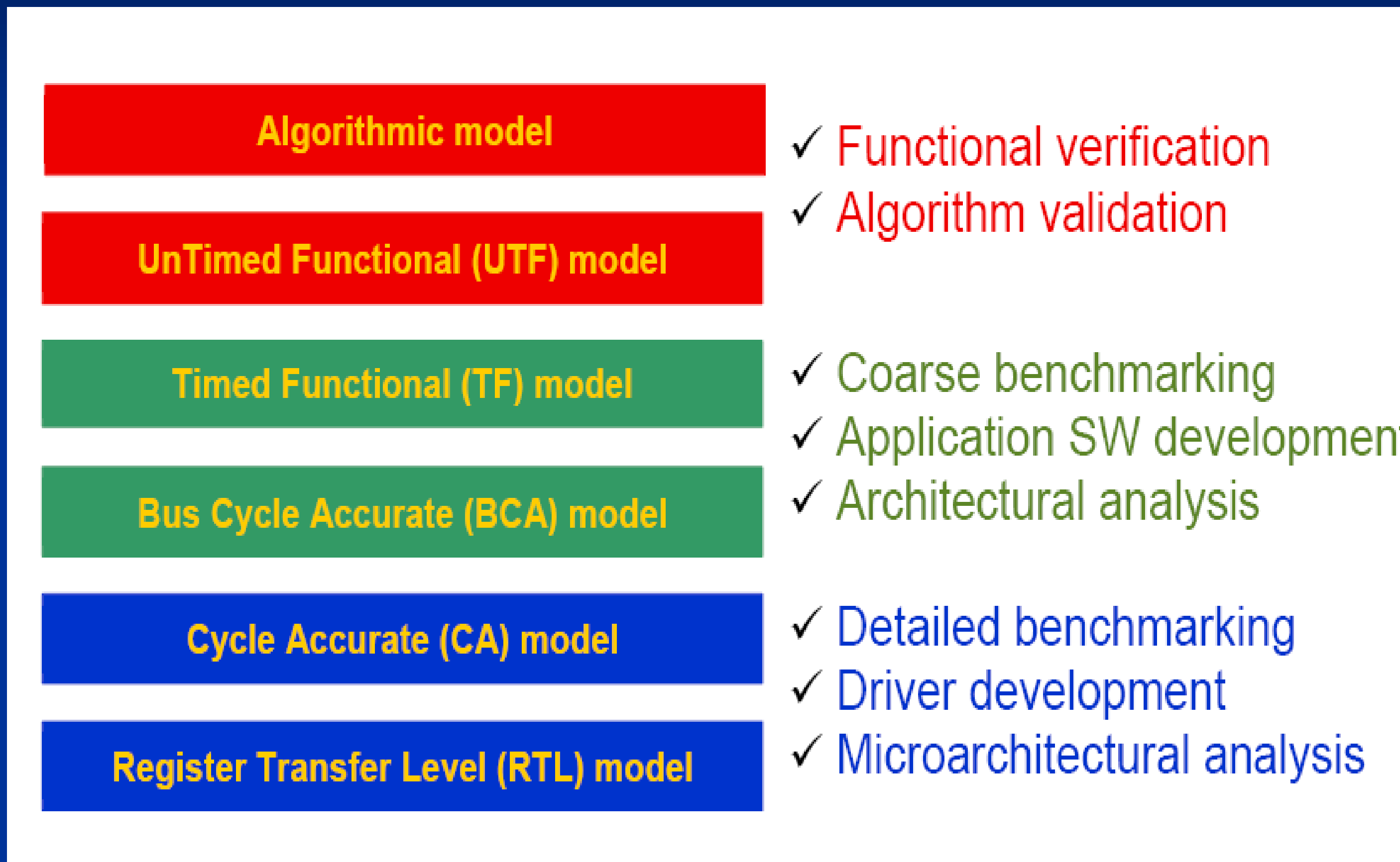


Scope of Layers

- Algorithmic model
- UnTimed Functional (UTF) model
- Timed Functional (TF) model
- Bus Cycle Accurate (BCA) model
- Cycle Accurate (CA) model
- Register Transfer Level (RTL) model

- No notion of time (processes and data transfers)
- Notion of time (processes and data transfers)
- Cycle accuracy, signal accuracy

Purpose of layers



- Low-level layers: Clock management (**sc_clock**), signal support (**sc_signal**),01XZ values (**sc_lv**), flexible synchronization of modules(**SC_METHOD**, **SC_THREAD**), VHDL-like scheduling
- High-level layers: Powerful object-oriented features (C++ roots), easy synchronization of modules (**SC_THREAD, SC_CTHREAD**), reconfigurable sensitivity according to circumstances (**sc_event**), high-level abstractions of HW resources (FIFOs, mutexes,semaphores…)

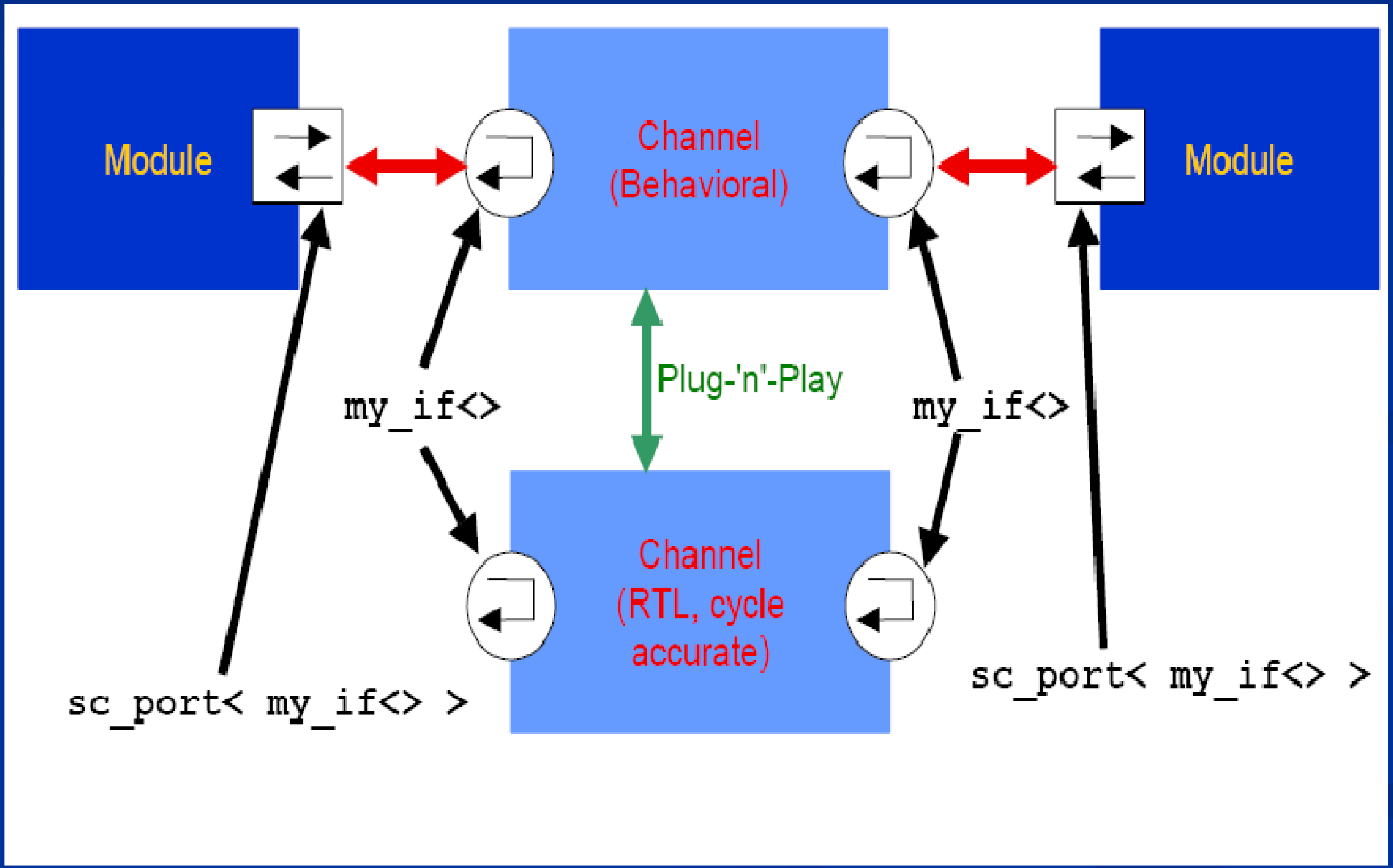
- SystemC "channels" do not just translate into `sc_signal`
- Other channels are available, *e.g.*:
  - `sc_fifo`
  - `sc_mutex`
  - `sc_semaphore`
- These channels can be bound to ports like `sc_signal` channels do, but...
  - Very useful high-level functionality
  - Not cycle accurate!
- Custom channels can be built (whole interconnects!)

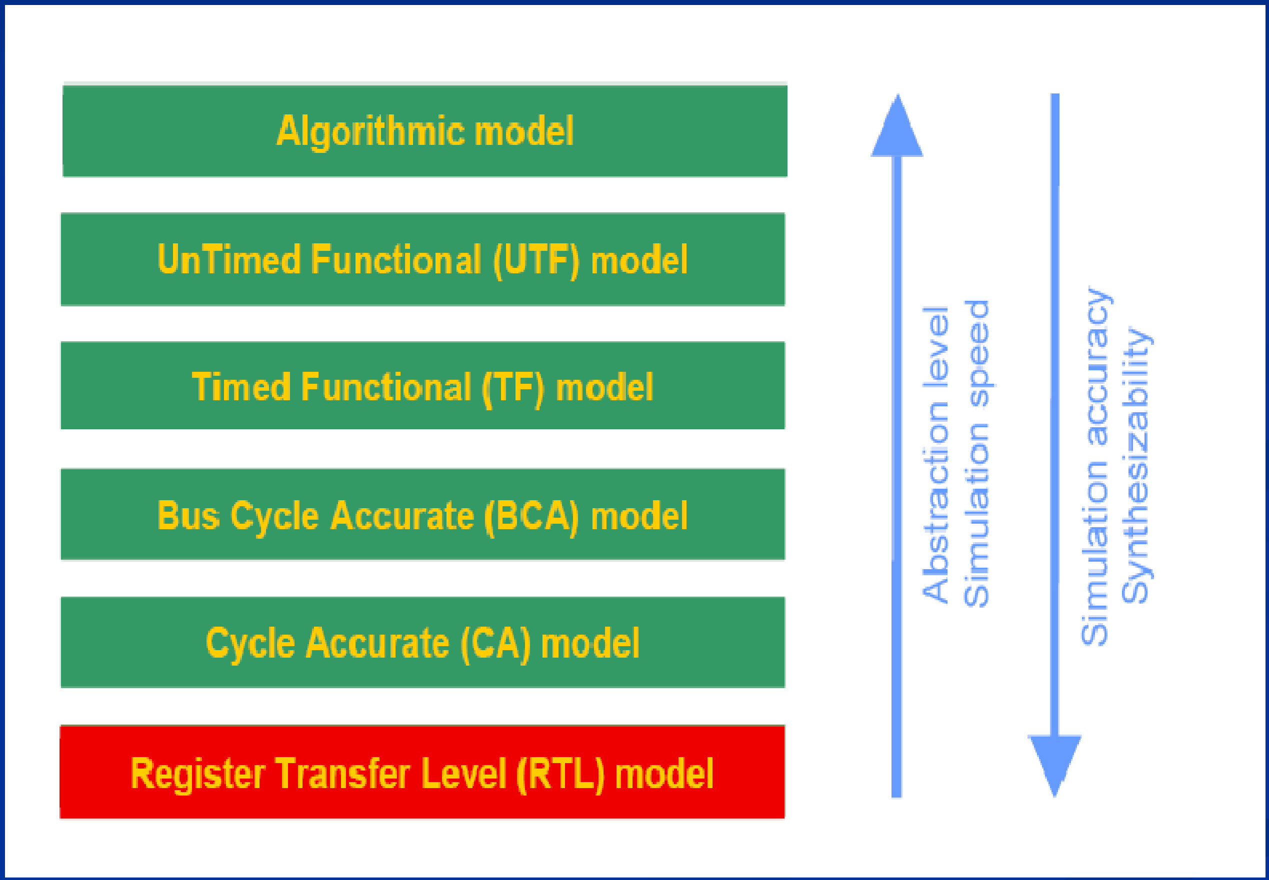
SystemC Philosophy

- Language is very rich
- While features can be intermixed, SystemC extensions to C++ are mostly aimed at different design domains:
  - RTL designers can write VHDL-like code
  - System designers (HW/SW designers) can write C++ code while taking advantage of some “bonus” features like concurrency, powerful synchronization mechanisms, and abstractions of actual hardware
- Design refinement can be done while staying within the SystemC framework, without learning new languages/tools

- SystemC provides a very powerful mechanism to refine communication protocols:
  - Modules only have ports
  - Communication happens through channels
  - Ports are connected to channels via interfaces
  - Interfaces just *declare* channel functionality, actual implementation is inside of channel itself
- If two channels expose the same interface, they can be replaced with full plug-'n'-play
- Cycle-accurate and purely functional channels could be interchanged!

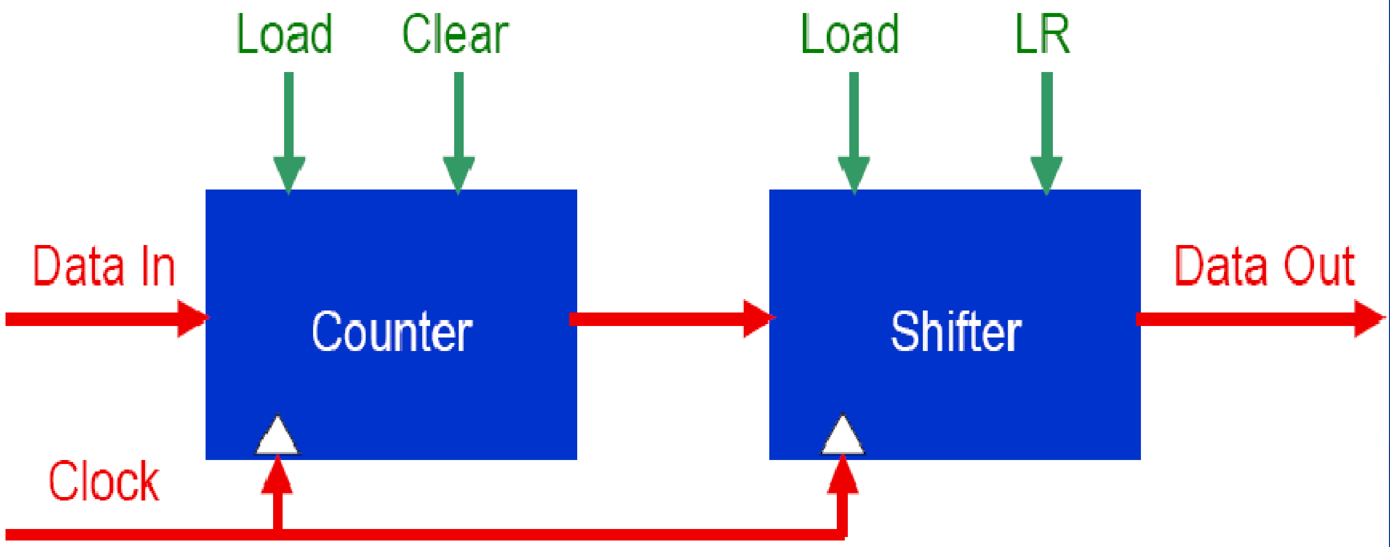
Channel binding




Algorithmic model
UnTimed Functional (UTF) model
Timed Functional (TF) model
Bus Cycle Accurate (BCA) model
Cycle Accurate (CA) model
Register Transfer Level (RTL) model
Abstraction level
Simulation speed
Simulation accuracy
Synthesizability

RTL Model Example

- RTL level: signal accurate, cycle accurate, resource accurate
- Can not use abstractions (functional units, communication infrastructures, …)



- An 8 bit counter. This counter can be loaded on a **clk** rising edge by setting the input **load** to 1 and placing a value on input **din**. The counter can be cleared by setting input **clear** high.
- A very basic 8 bit shifter. The shifter can be loaded on a **clk** rising edge by placing a value on input **din** and setting input **load** to 1. The shifter will shift the data left or right depending on the value of input **LR**. If **LR** is low the shifter will shift right by one bit, otherwise left by one bit.
- Local temporary values are needed because the value of output ports cannot be read.

```
// counter.h
SC_MODULE(counter) {
  sc_in<bool> clk;
  sc_in<bool> load;
  sc_in<bool> clear;
  sc_in<sc_uint<8> > din;
  sc_out<sc_uint<8> > dout;
  sc_uint<8> countval;
  void counting();
  SC_CTOR(counter) {
    SC_METHOD(counting);
    sensitive << clk.pos();
  }
};
```

Only SC_METHODs for synthesis. Tools don’t like the “wait()” concept ☹

```
// counter.cpp
#include "counter.h"

void counter::counting()
{
  if (clear.read())
    countval = 0;
  else if (load.read())
        countval = (unsigned int)din.read();
      else
        countval++;
  dout.write(countval);
}
```

Which control priorities did we choose?

```
// shifter.h
SC_MODULE(shifter) {
  sc_in<sc_uint<8> > din;
  sc_in<bool> clk;
  sc_in<bool> load;
  sc_in<bool> LR;              // shift left if true
  sc_out<sc_uint<8> > dout;
  sc_uint<8> shiftval;
  void shifting();
  SC_CTOR(shifter) {
    SC_METHOD(shifting);
    sensitive << clk.pos();
  }
};
```

```
// shifter.cpp
#include "shifter.h"
void shifter::shifting() {
  if (load.read())
    shiftval = din.read();
  else if (!LR.read()) {            // shift right
       shiftval.range(6, 0) = shiftval.range(7, 1);
       shiftval[7] = '0'; }
     else if (LR.read()) {          // shift left
          shiftval.range(7,1)=shiftval.range(6,0);
          shiftval[0] = '0'; }
  dout.write(shiftval);
}
```

Bus Cycle Accurate model

- Pin-accurate like RTL, but not cycle-accurate
- Does not imply mapping of computation onto HW resources

- Euclid's algorithm to find the Greatest Common Divisor (GCD) of two numbers:
  - Given a, b, with $a \geq 0$, $b > 0$,
  - If b divides a, then GCD(a, b) = b;
  - Else, GCD(a, b) = GCD(b, a mod b).

```
// euclid.h
SC_MODULE (euclid) {
  sc_in_clk clock;
  sc_in<bool> reset;
  sc_in<unsigned int> a, b;
  sc_out<unsigned int> c;
  sc_out<bool> ready;
  void compute();

  SC_CTOR(euclid) {
      SC_CTHREAD(compute, clock.pos());
      watching(reset.delayed() == true);
  }
};
```

```
// euclid.cpp
void euclid::compute()
{
   unsigned int tmp_a = 0, tmp_b;              // reset section
   while (true) {
        c.write(tmp_a);                        // signaling output
        ready.write(true);
        wait();                                // moving to next cycle
        tmp_a = a.read();                      // sampling input
        tmp_b = b.read();
        ready.write(false);
        wait();                                // moving to next cycle
        while (tmp_b != 0) {                   // computing
                unsigned int r = tmp_a;
                tmp_a  = tmp_b;
                r = r % tmp_b;
                tmp_b = r;
        }
   }
}
```

% operator: how to do in HW?

Recursive: how many cycles will it take?

UnTimed Functional

```
// constgen.h
SC_MODULE(constgen) {
{
   sc_fifo_out<float> output;

   SC_CTOR(constgen) {
        SC_THREAD(generating());
   }

   void generating() {
        while (true) {
                output.write(0.7);
        }
   }
}
```

```
// adder.h
SC_MODULE(adder) {
{
   sc_fifo_in<float> input1, input2;
   sc_fifo_out<float> output;

   SC_CTOR(adder) {
        SC_THREAD(adding());
   }

   void adding() {
        while (true) {
                output.write(input1.read() + input2.read());
        }
   }
}
```

```
// forker.h
SC_MODULE(forker) {
{
   sc_fifo_in<float> input;
   sc_fifo_out<float> output1, output2;
   SC_CTOR(forker) {
        SC_THREAD(forking());
   }
   void forking() {
        while (true) {
                float value = input.read();
                output1.write(value);
                output2.write(value);
        }
   }
}
```

```
// printer.h
SC_MODULE(printer) {
{
   sc_fifo_in<float> input;
   SC_CTOR(printer) {
        SC_THREAD(printing());
   }
   void printing() {
        for (unsigned int i = 0; i < 100; i++) {
                float value = input.read();
                printf("%f\n", value);
        }
        return;                 // this indirectly stops the simulation
                                // (no data will be flowing any more)
   }
}
```

```
// main.cpp
int sc_main(int, char**) {
   constgen my_constgen("my_constgen_name");                  // module
   adder my_adder("my_adder_name");                           // instantiation
   forker my_forker("my_forker_name");
   printer my_printer("my_printer_name");
   sc_fifo<float> constgen_adder("constgen_adder", 5);        // FIFO
   sc_fifo<float> adder_fork("adder_fork", 1);                // instantiation
   sc_fifo<float> fork_adder("fork_adder", 1);
   sc_fifo<float> fork_printer("fork_printer", 1);
   fork_adder.write(2.3);                                     // initial setup
   my_constgen.output(constgen_adder); my_adder.input1(constgen_adder);
   my_adder.input2(fork_adder); my_adder.output(adder_fork);
   my_fork.input(adder_fork); my_fork.output1(fork_adder);   // binding
   my_fork.output2(fork_printer); my_printer.input(fork_printer);
   sc_start(-1);                      // simulate "forever". Will exit
   return 0;                          // when no events are queued
}                                     // (printer exits, fifos saturate)
```

Timed Functional

```
// constgen.h
SC_MODULE(constgen) {
{
   sc_fifo_out<float> output;

   SC_CTOR(constgen) {
        SC_THREAD(generating());
   }

   void generating() {
        while (true) {
                wait(200, SC_NS);
                output.write(0.7);
        }
   }
}
```

Synthesis Subset of SystemC

- At least two

- "Behavioral" Subset
  - Resource sharing, binding, and allocation done automatically
  - System determines how many adders you have

- Register-transfer-level Subset
  - More like Verilog
  - You write a "+", you get an adder

Do People Use SystemC?

- Not as many as use Verilog or VHDL
- Growing in popularity
- People recognize advantage of being able to share models
- Most companies were doing something like it already
- Use someone else's free libraries? Why not?

Conclusions

- C++ dialect for modeling digital systems
- Provides a simple form of concurrency
  - Cooperative multitasking

- Modules
  - Instances of other modules
  - Processes

Conclusions

- Perhaps even more flawed than Verilog
- Verilog was a hardware modeling language forced into specifying hardware
- SystemC forces C++, a software specification language, into modeling and specifying hardware

- Will it work? Time will tell.

References

- Course slides : Prof. Stephen A. Edwards
- Course slides : Federico Angiolini