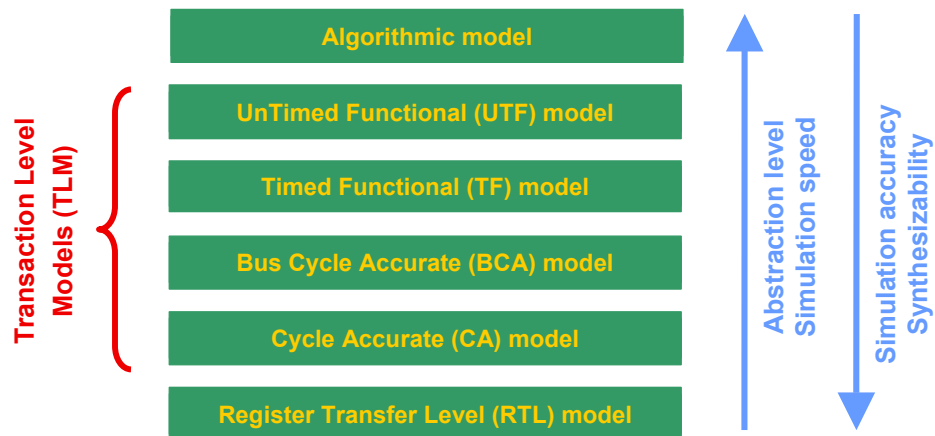


SystemC Tutorial

Federico Angiolini
DEIS Universita' di Bologna, Italy
fangiolini@deis.unibo.it

Layers of Hardware Design



Scope of Layers

Algorithmic model	No notion of time (processes and data transfers)
UnTimed Functional (UTF) model	
Timed Functional (TF) model	Notion of time (processes and data transfers)
Bus Cycle Accurate (BCA) model	
Cycle Accurate (CA) model	Cycle accuracy, signal accuracy
Register Transfer Level (RTL) model	

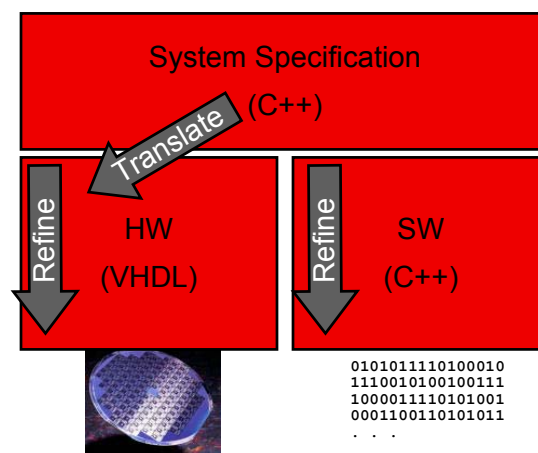
Purpose of Layers

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
Register Transfer Level (RTL) model	✓ Microarchitectural analysis

What language?

Problem: very different levels of abstraction
Modeling language to use?
High-level: Java, Visual Basic, C++
Low-level: HDLs (VHDL, Verilog)
Huge **modeling gap**
Need to translate models...

Design Overhead

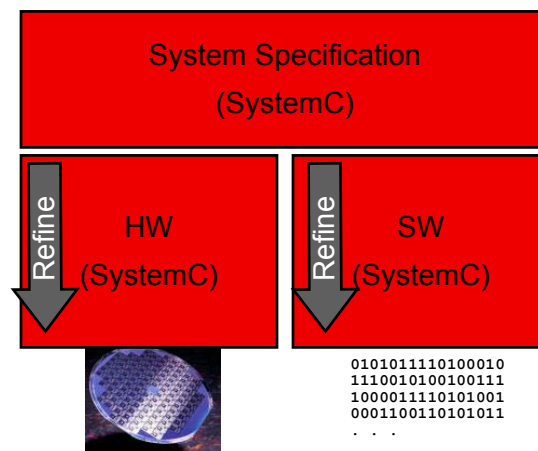


Why not just C++?

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)

Plain vanilla C++ not viable!

SystemC paradigm



How can this be achieved?

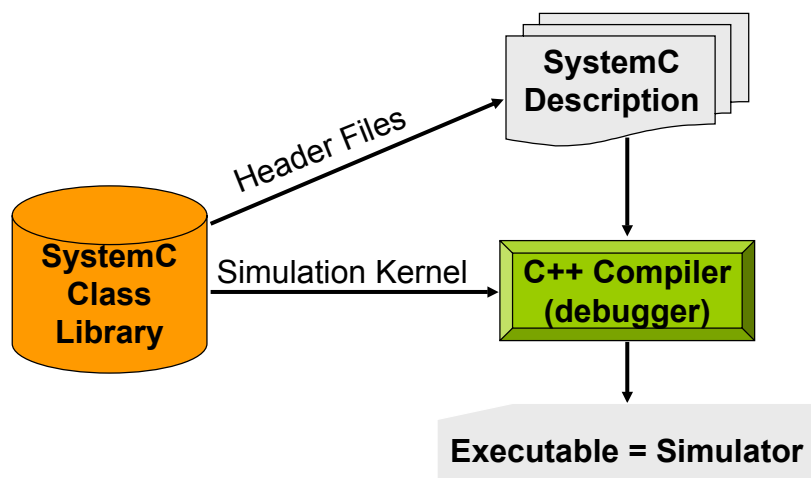
... C++ extensions!

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 Infrastructure



SystemC Advantages

Unified language across all stages of platform design
(easier tool interoperability, designer training)

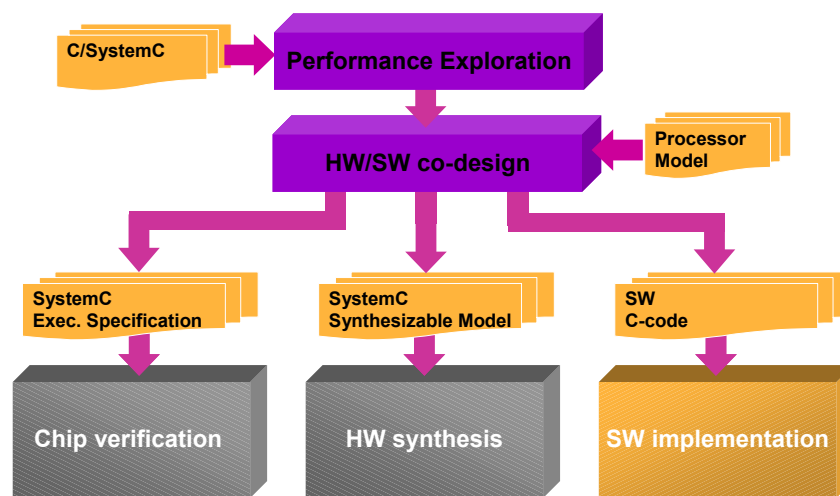
Unified language across HW and SW development
(promoting co-design)

Allows faster simulation/refinement/reworking of modules

Builds upon one of the most widespread programming
languages (many tools, programmers)

Lightweight

SystemC Toolchain



SystemC Features

- ✓ Concurrency support: modules
- ✓ Notion of time: clocks, custom `wait()` calls
- ✓ Communication model: signals, protocols, handshakes
- ✓ Reactivity to events: support for events, sensitivity list, `watching()` construct
- ✓ Data types: logic values, bit vectors, fixed point

C ++/SystemC results in a suitable platform!

SystemC Core Language

High-Level Channels

Kahn Process Networks, Master/Slave libraries, ...

Elementary channels

Signals, Timers, Mutexes, Semaphores, FIFOs, ...

Core language

Modules/Processes
Ports/Interfaces
Events
Channels
Event-driven simulation kernel

Data types

4-valued logic types (01XZ)
Bit/logic vectors
Arbitrary precision integers
Fixed point
C++ user-defined

C++

Modules

They map functionality of HW/SW blocks

Derived from SystemC class `sc_module`

They represent the basic building block of every system

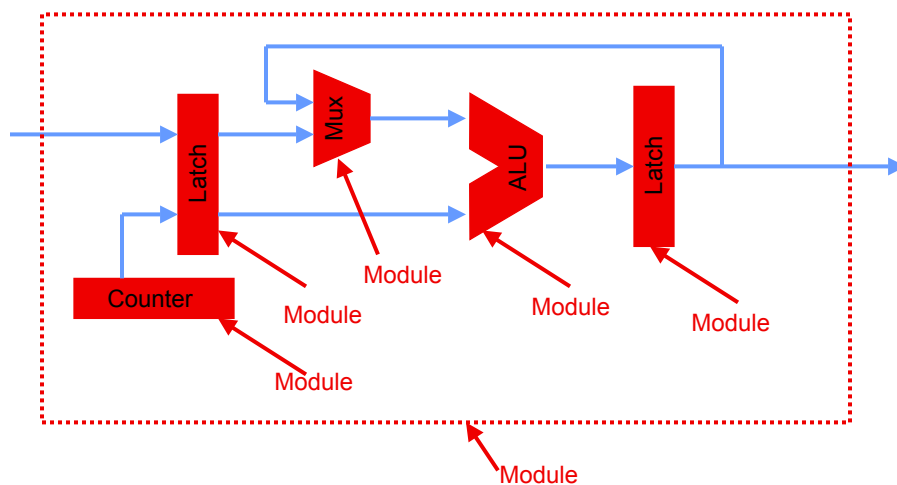
Modules can contain a whole hierarchy of sub-modules and provide private variables/signals

Modules need:

Communication: Modules can interface to each other via **ports/interfaces/channels**

Functionality: achieved by means of **processes**

Modules



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
    }
};
```

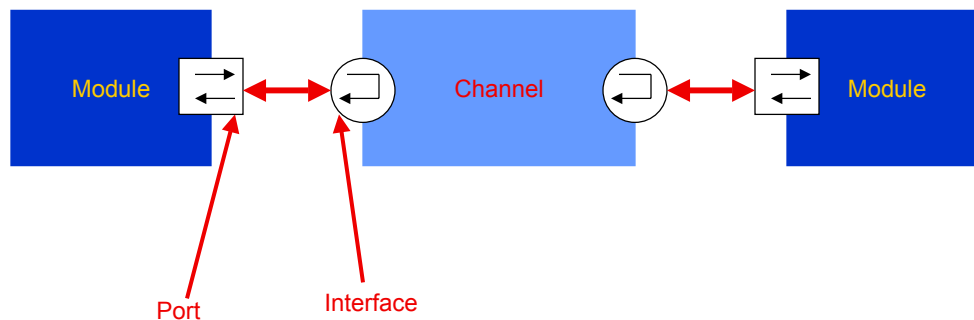
Basically a C++ class
with member variables,
member functions and
constructor!

Interfaces, Ports, Channels

Need to exchange data among modules

Give **ports** to modules

At initialization time, ports are bound to **channels** via **interfaces**



Interfaces

They provide just the prototype of a set of operations (typically, at least `read` and `write`)

Implementation is a burden of channels

Derived from the `sc_interface` class

Two are widely used:

```
sc_signal_in_if<T>
```

provides a `read()` returning data of type `T`

```
sc_signal_inout_if<T>
```

derives from the first, thus providing `read()`;

additionally declares a `write()` function taking a reference to `T` as its input

(The “out” version is just the same as “inout”)

Ports

They provide communication functions to modules

Derived from SystemC class `sc_port<class IF, int N=1>` (type of interface, number of connected interfaces)

On the outside, they connect to channels by means of interfaces

Typical channel (in RTL models): `sc_signal`. In this case, shortcuts exist for specialized classes: `sc_in<class T>`, `sc_out<class T>`, `sc_inout<class T>` (ports connected to `N=1` interfaces of type `sc_signal_in_if<class T>` etc.)

Simple instantiation:

```
sc_inout<bool> my_port;
```

Methods are made available by the underlying interface:

```
my_port.read(), my_port.write(), ...
```

Channels

They implement the prototypes described by interfaces
Actual data shuffling...

Between modules, or between processes in a module

Typical example: the hardware signal `sc_signal<T>`,
derived from the interface `sc_signal_inout_if<T>`

Arbitrary complexity, up to whole interconnects

Other examples:

```
sc_fifo<T> implementing the interfaces  
sc_fifo_in_if<T> and sc_fifo_out_if<T> (blocking  
and non-blocking versions of access)  
sc_mutex
```

Port instantiation

```
//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

Processes

They provide module functionality
Implemented as member functions
Three kinds of processes available:

`SC_METHOD`
`SC_THREAD`
`SC_CTHREAD`

These macros tell the SystemC scheduler what to do with such processes. Needed because a member function is not necessarily a process. The macros operate specific registrations with the scheduler

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

SC_METHOD

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)

SC_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!!

SC_THREAD

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

SC_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...

SC_CTHREAD

Almost identical to `sc_THREAD`, but implements "clocked threads"

Will be deprecated in future releases: not really needed

Only allows a single edge of a single signal on the sensitivity list (*i.e.* clock rising or falling edges)

Useful for high-level simulations, where the clock is used as the only synchronization device

Adds `wait_until()` and `watching()` semantics for easy deployment

wait_until()

```
do
    wait();
while (in_a.read() != true ||
       in_b.read() != true);
```



```
wait_until(in_a.delayed() == true &&
           in_b.delayed() == true);
```

watching()

```
//my_module.h
SC_CTOR(my_module) {
    SC_THREAD(my_thread);
    sensitive << clock.pos();
}
```

```
//my_module.cpp
void my_module::my_thread() {
    while(true) {
        if (reset.read())
            [reset code]
        [...]
        if (reset.read())
            [reset code]
        [...]
        wait();
    }
}
```

```
//my_module.h
SC_CTOR(my_module)
{
    SC_CTHREAD(my_ctypead, clock.pos());
    watching(reset.delayed() == true);
}
```

```
//my_module.cpp
void my_module::my_ctypead()
{
    [reset code]
    while(true)
    {
        [...]
        wait();
    }
}
```

Process Summary

SC_METHOD: method process

sensitive to a set of signals
executed until it returns

SC_THREAD: thread process

sensitive to a set of signals
executed until a `wait()`

SC_CTHREAD: clocked thread process

sensitive only to one edge of clock
execute until a `wait()` or a `wait_until()`

`watching()` restarts from top of process body (reset evaluated on active edge)

Signals

The most common type of channels (in RTL designs)

Derived from `sc_prim_channel`

Are bound to ports by means of interfaces

Used to connect modules through ports

May be local inside of a specific module

Special signal: clock (`sc_clock`). Multiple clocks can be instantiated at once, with arbitrary phase relationship

Signal Instantiation 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);
           name      period
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);
```

The diagram illustrates the signal instantiation and binding between two modules, `my_module_1` and `my_module_2`, and a clock signal `my_clock`. The code on the left shows the instantiation and port connections. The diagram on the right shows the resulting connections: `my_clock` is connected to the `clock_port` of both `my_module_1` and `my_module_2`. `my_module_1` outputs `my_signal1` to `my_module_2`, and `my_module_2` outputs `my_signal2` to `my_module_1`. The parameters for `sc_clock` are: name ("my_clock"), period (20), duty cycle (0.5), start time (2), and start phase (true).

Data Types

Can be applied to signals and ports

All of what C++ supports, including user-defined types
(`typedef ...`)

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)`

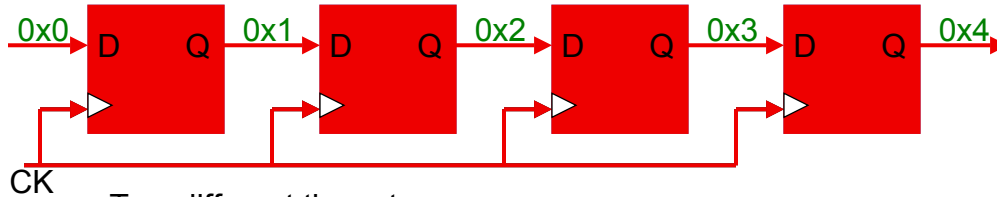
or reduction: `x.or_reduce()`

SystemC Scheduler

Similar to HDL schedulers

Problem of concurrency (execution order??)

Actual execution must be **sequential!!**



Two different time steps:

Discrete simulation cycle (*i.e.* clock period)

“Delta cycle” (*i.e.* virtual delay, without actual length)

“**Evaluate then update**” semantics

Scheduler Timings

1. Major timing step (clock cycle, specified delay, ...)
2. Resume processes waiting for this event and execute their body (evaluate stage) until they all suspend again. Output signal updates are only queued
3. Minor timing step (“Delta cycle”: simulation time does not actually go on)
4. Update all of the output signals (update stage)
5. Check whether this resumes other processes. If yes, back to 2
6. Back to 1

Scheduler Timings

Order of process resumption, while actually deterministic, is unspecified

Designer must not rely on peculiar behavior (like in hardware!)

SystemC semantics allow for designs independent of specific order of process execution (concurrency support)

Events and Dynamic Sensitivity

Events are pervasive in SystemC

Event objects: derived from class `sc_event`

Event objects are **not** event occurrences: they represent the “what-if” scenario

Event objects keep a list of processes waiting for them

Event generators (processes, channels) “notify” event objects of event occurrences

Event objects, checking their list, tell the scheduler which processes to resume

This is how sensitivity lists are made (“**static sensitivity**”)

Dynamic sensitivity: at runtime, making processes sensitive to events other than those in their sensitivity list (this makes no sense at RTL level...)

Events and Dynamic Sensitivity

```
sc_event e1, e2, e3;
```

```
// Process P1 (SC_THREAD, sensitive to clock.pos())
wait(); // wait for static sensitivity list
wait(e1); // wait for event
wait(e1 | e2 | e3); // wait for first event
wait(e1 & e2 & e3); // wait for all events
wait(200, SC_NS); // wait for 200 ns
wait(200, SC_NS, e1 | e2); // wait, 200 ns timeout
```

Dynamic
sensitivity

```
// Process P2
e1.notify();
```

Events and Dynamic Sensitivity

Notification can be either:

Immediate (DANGEROUS!!!): `e1.notify()` (This may trigger an unpredictable amount of executions of the sensitive process, depending on scheduling)

After a Delta cycle: `e1.notify(SC_ZERO_TIME)`

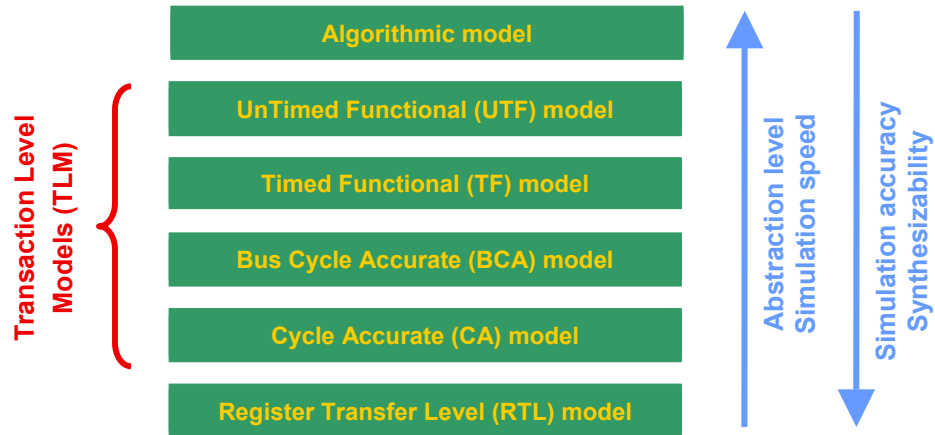
After a specified time interval: `e1.notify(sc_time(20, SC_NS))`

Especially useful for:

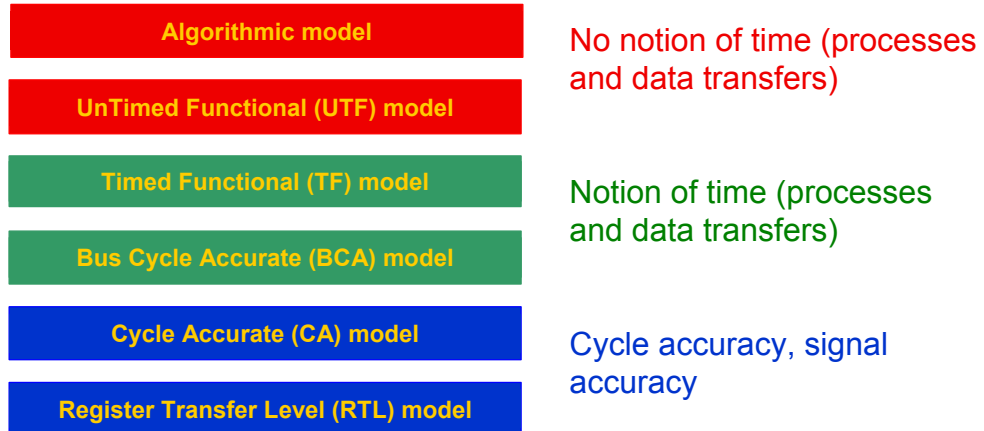
RTL level design (back-annotation of delays)

Behavioral level design ("asynchronous"/conditional event management without much effort)

Layers of Hardware Design



Scope of Layers



Purpose of Layers

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
Register Transfer Level (RTL) model	✓ Microarchitectural analysis

What Tools Do We Need?

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...)

High-Level Abstractions of HW Resources

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

Communication Refinement

SystemC provides a very powerful mechanism to refine communication protocols:

Modules only have **ports**

Communication happens through **channels**

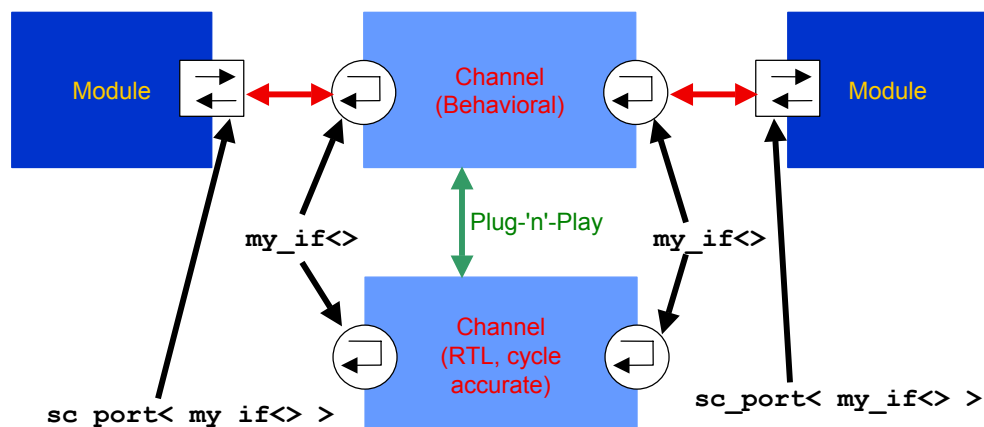
Ports are connected to channels via **interfaces**

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



Creating New Channels

Useful to encapsulate low-level behavior in reusable code modules

Useful to abstract from full implementations of complex interconnects to less accurate ones, which are fast to deploy and simulate (prototyping)

Channels can be

primitive (supporting “Evaluate then update” semantics)

hierarchical (may include modules, processes, ports, subchannels)

Code to be implemented:

The interface

The channel itself

Implementing an Interface

An interface does not actually do anything

It just has to declare prototypes for the channel methods

```
class my_read_if : virtual public sc_interface
{
    public:
        virtual char read() = 0;
};
```

Written this way, a port will only be able to connect to one of the interfaces – either for read or for write!

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

Implementing an Interface

To be able to implement a read/write connection to the channel, we need a derived interface:

```
class my_readwrite_if :  
    public my_read_if,  
    public my_write_if  
{  
};
```

Now both `read()` and `write()` will be available to a module port connected to `my_readwrite_if`

If a port uses directly `my_read_if` and `my_write_if`, it will be a unidirectional port

The channel can choose whether to inherit from `my_readwrite_if` or from both `my_read_if` **AND** `my_write_if`

Implementing a Channel

The channel must provide communication functionality

The channel must provide the implementation of the methods declared in its interfaces

The channel inherits from `sc_prim_channel` and thus can access its API, especially:

```
void request_update()
```

```
virtual void update() = 0 (mandatorily requires an  
implementation inside of the channel!)
```

The channel also inherits from `sc_interface` (through its interfaces) and thus can access its API, especially:

```
virtual const sc_event& default_event() (tells the  
simulation kernel which is the event on which to enqueue  
processes waiting on their sensitivity list)
```

Request-Update Semantics

The channel must comply with the “**Evaluate then update**” (VHDL-like) paradigm of SystemC! To do so:

When some channel method is invoked (typically writes), the channel calls the simulation kernel via `sc_prim_channel`'s `request_update()`

The simulation kernel schedules the channel for update in the next Delta cycle. When such time arrives, the kernel calls the channel's `update()` method

`update()` is still inherited by `sc_prim_channel`, but its implementation is channel-specific

Implementing a Channel

```
class my_channel :
  public sc_prim_channel, public my_read_if, public my_write_if
{
public:
  Inherits both interfaces
  [...]
  virtual char read() { return curr_val };
  virtual void write(char val) { next_val = val;
    if (next_val != curr_val) request_update(); }
  virtual const sc_event& default_event() const
    { return val_chg_event; }
protected:
  virtual void update() { curr_val = next_val;
    val_chg_event.notify(SC_ZERO_TIME); }
  char curr_val, next_val;
  sc_event val_chg_event;
};
```

“Evaluate then update” semantics

Sensitivity support (called by the simulation kernel at initialization)

One Delta cycle

Instantiating a Channel

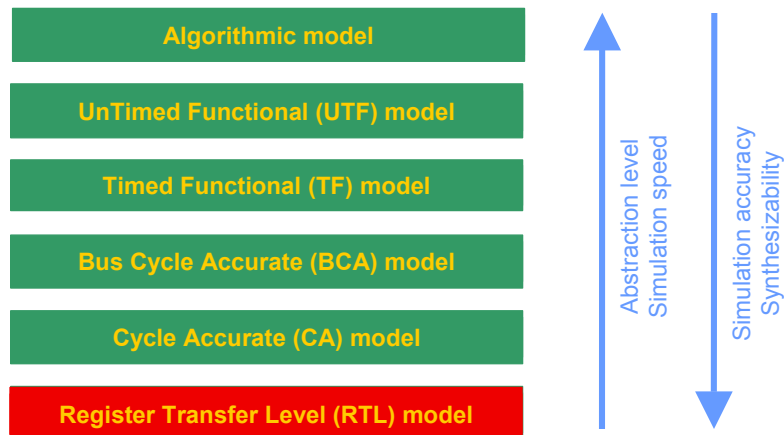
```
// my_module.h
#include "systemc.h"
SC_MODULE (my_module)
{
    sc_port <my_read_if> in;
    void my_function();

    SC_CTOR(my_module)
    {
        SC_METHOD(my_function);
        sensitive << in;
    }
};
```

```
// my_module.cpp
#include "my_module.h"
void my_module::my_function()
{
    printf("%c\n", in.read());
};
```

```
// main.cpp
#include "my_module.h"
#include "my_channel.h"
[...]
my_module my_mod;
my_channel my_ch;
my_mod.in(my_ch);
```

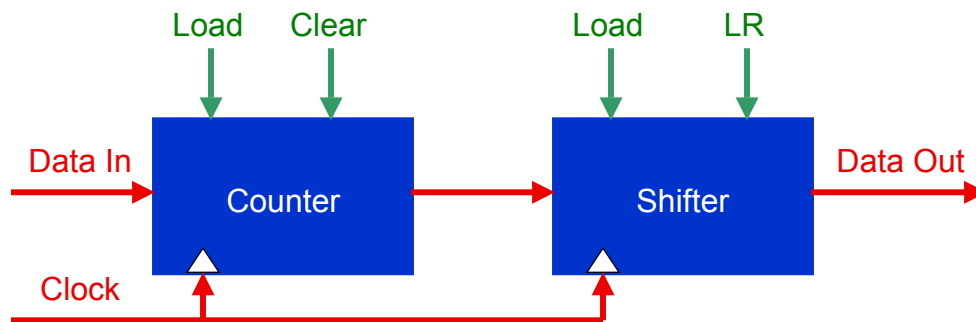
Layers of Hardware Design



An RTL Model Example

RTL level: signal accurate, cycle accurate, resource accurate

Can not use abstractions (functional units, communication infrastructures, ...)



An RTL Model Example

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.

An RTL Model Example

```
// 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 ☹

An RTL Model Example

```
// 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?

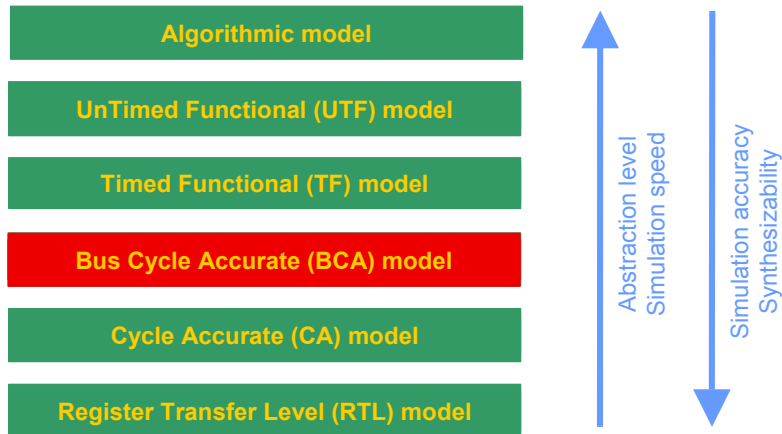
An RTL Model Example

```
// 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();
    }
};
```

An RTL Model Example

```
// 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);
}
```

Layers of Hardware Design



A Bus Cycle Accurate Model Example

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 $\text{GCD}(a, b) = b$;
Else, $\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$.

A Bus Cycle Accurate Model Example

```
// 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);
    }
};
```

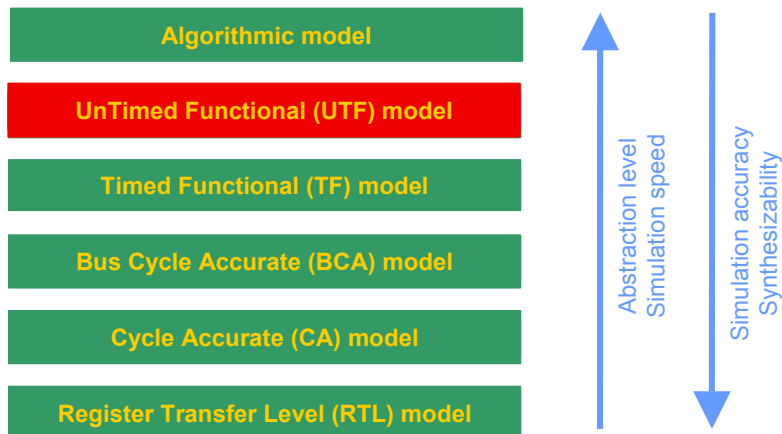
A Bus Cycle Accurate Model Example

```
// euclid.cpp
void euclid::compute()
{
    unsigned int tmp_a = 0, tmp_b;           // reset section
    while (true) {                          // signaling output
        c.write(tmp_a);
        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?

Layers of Hardware Design



L. Benini Metodologie di Progettazione Hardware-Software

AA 2004-2005

Slide -69 -

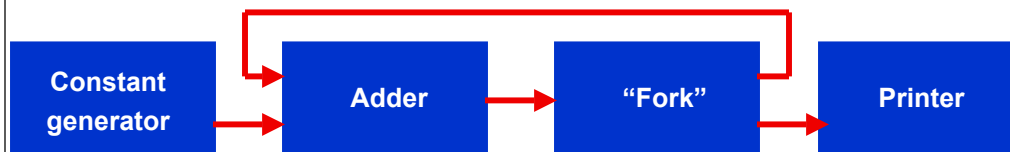
An UnTimed Functional (UTF) Model Example

Very widespread modeling level

Describes functionality, but not timing

Most general form of UTF model: "Kahn Process Networks" (KPN)

But often implemented as "Dataflow model": modules communicating with each other via blocking FIFOs



L. Benini Metodologie di Progettazione Hardware-Software

AA 2004-2005

Slide -70 -

An UnTimed Functional (UTF) Model Example

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

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

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

An UnTimed Functional (UTF) Model Example

```
// 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());
        }
    }
}
}
```

An UnTimed Functional (UTF) Model Example

```
// 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);
        }
    }
}
}
```

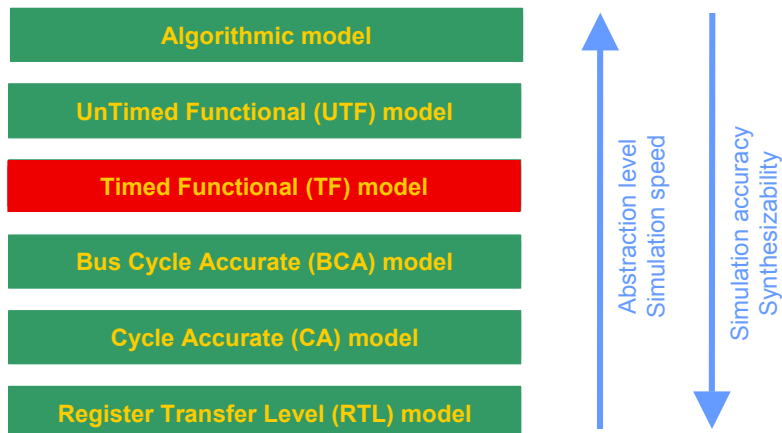
An UnTimed Functional (UTF) Model Example

```
// 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)
    }
}
}
```

An UnTimed Functional (UTF) Model Example

```
// 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_forker.input(adder_fork); my_forker.output1(fork_adder); // binding
    my_forker.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)
```

Layers of Hardware Design



Refining to Timed Functional (TF) Model

```
// 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);
        }
    }
}
}
```

Of course a bit
simplistic, but...