

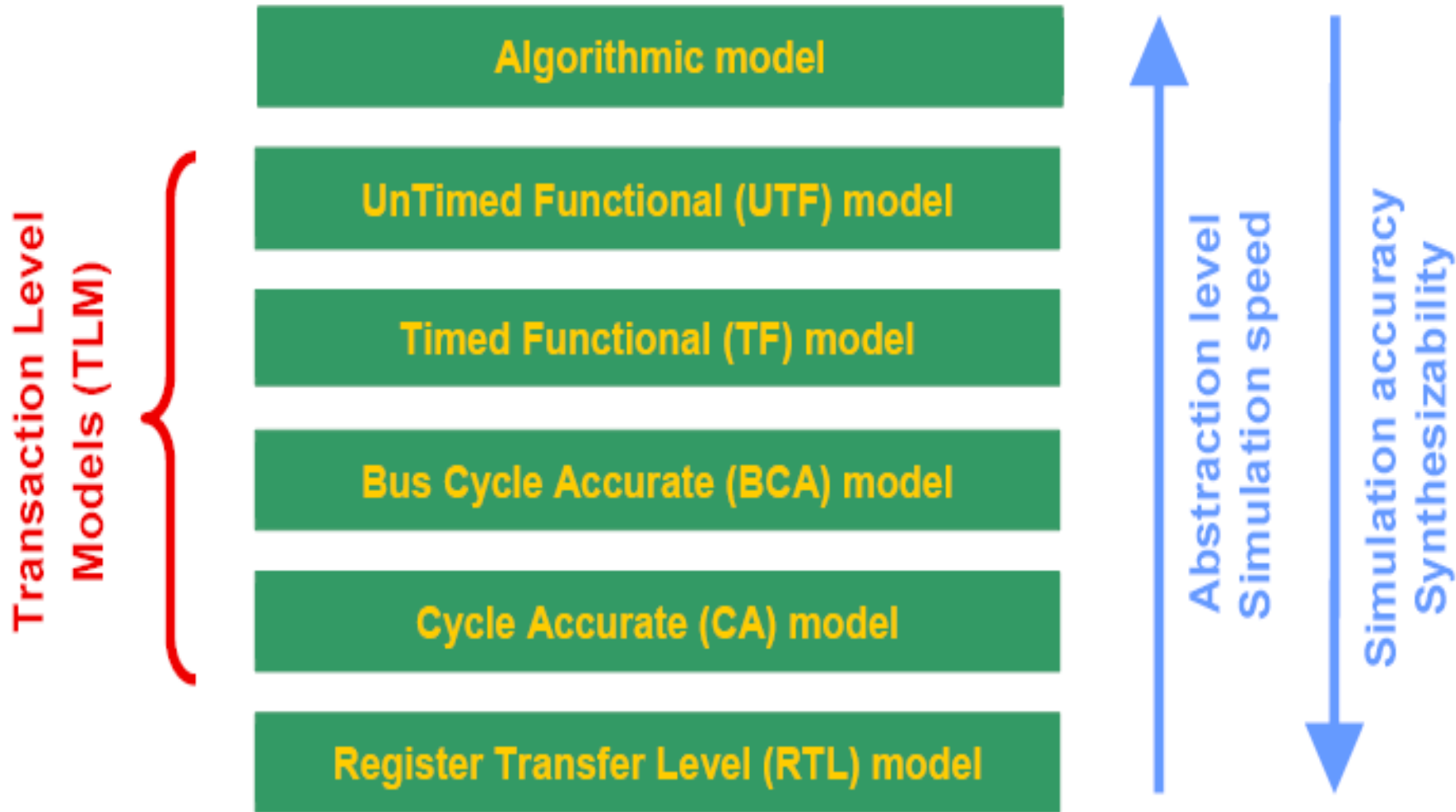


Macchine per L'Elaborazione dell'Informazione -
9. SystemC

Your Name
Your Title

Your Organization (Line #1)
Your Organization (Line #2)

Livelli di Astrazione nella Progettazione dell'HW



Livelli di Astrazione nella Progettazione dell'HW

Algorithmic model

Processi e Data transfer
NO TIME

UnTimed Functional (UTF) model

Timed Functional (TF) model

Processi e Data transfer
TIME: SI!

Bus Cycle Accurate (BCA) model

Cycle Accurate (CA) model

Definizione Accurata (Cicli, porti,
segnali ...)

Register Transfer Level (RTL) model

TIME: SI!

Scopi dei vari livelli:

Algorithmic model

Verifica Funzionale
Validazione degli Algoritmi

UnTimed Functional (UTF) model

Timed Functional (TF) model

Analisi Architetture
Benchmarking di alto livello
Sviluppo applicazioni SW

Bus Cycle Accurate (BCA) model

Cycle Accurate (CA) model

Benchmarking dettagliato
Sviluppo di Drivers

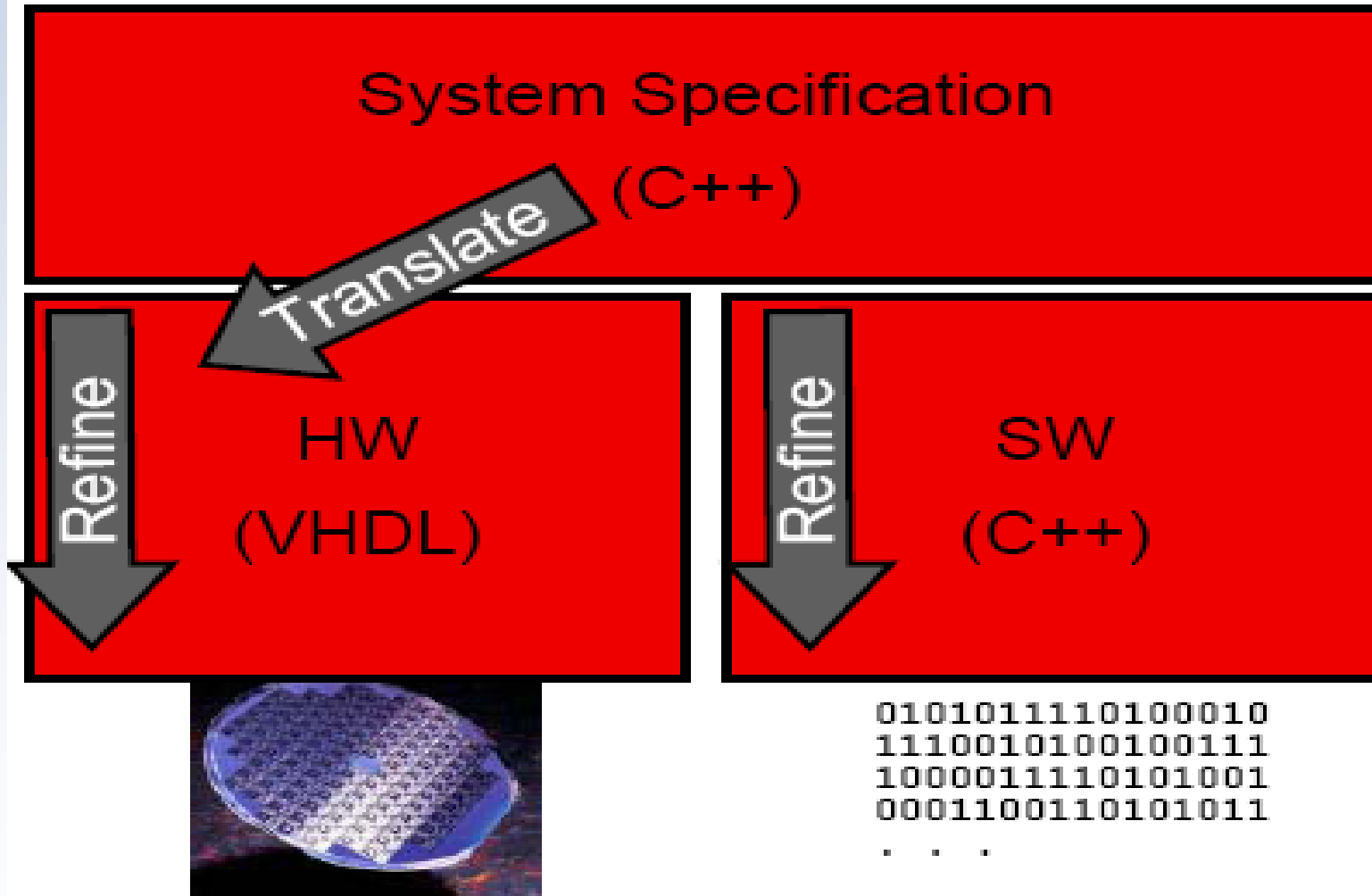
Register Transfer Level (RTL) model

Analisi della Micro-architettura

Che Linguaggi utilizzare ?

- Problema: Differenti Livelli di Astrazione
- Che linguaggio di Modellazione Utilizzare ?
 - Linguaggi ad alto Livello ?
 - Java, C++, C, LISP, ...
 - Linguaggi a Basso livello?
 - HDL (VHDL, Verilog), ...
- Rimane un gap di modellazione **enorme**
- **Bisognerebbe tradurre I modelli**
 - **Il traduttore coprirebbe il gap semantico...**

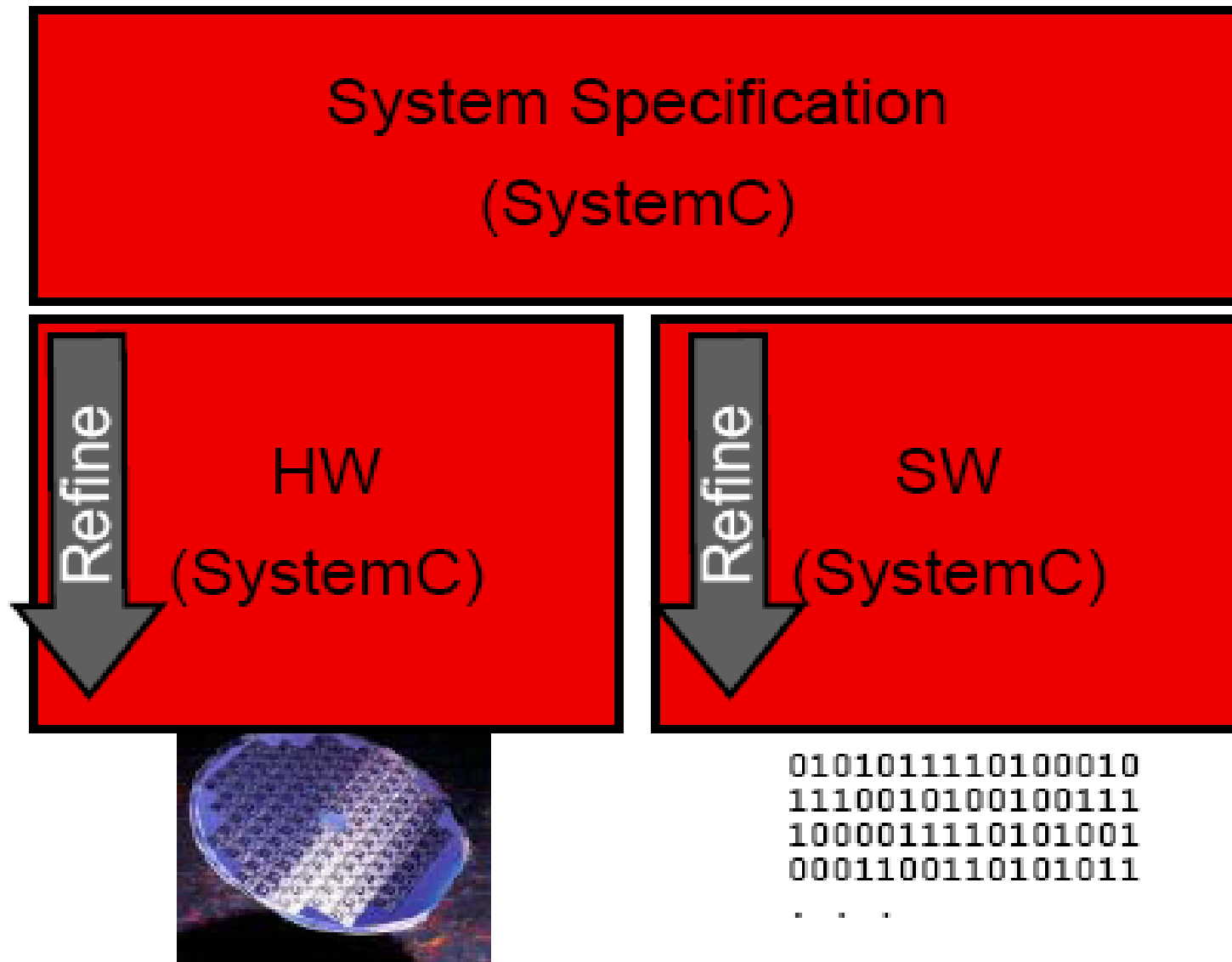
Che Linguaggi utilizzare ?



Perchè non utilizzare direttamente C++?

- L'HW è intrinsecamente parallelo
 - Non c'è supporto per la concorrenza in C++
- Il comportamento dell'HW è definito da opportune tempificazioni
 - In C++ manca un supporto esplicito per modellare il tempo
 - Clock, delays
- L'HW è reattivo agli eventi ...
 - La gestione degli Eventi è molto complessa e non sufficiente per modellare componenti HW
- Mancano in C++ i data-type necessari per la definizione dell'HW (valori logici, bit vectors, fixed/floating point maths etc..)
- **NON ESISTE UN C++ vanilla (e veramente standard)**

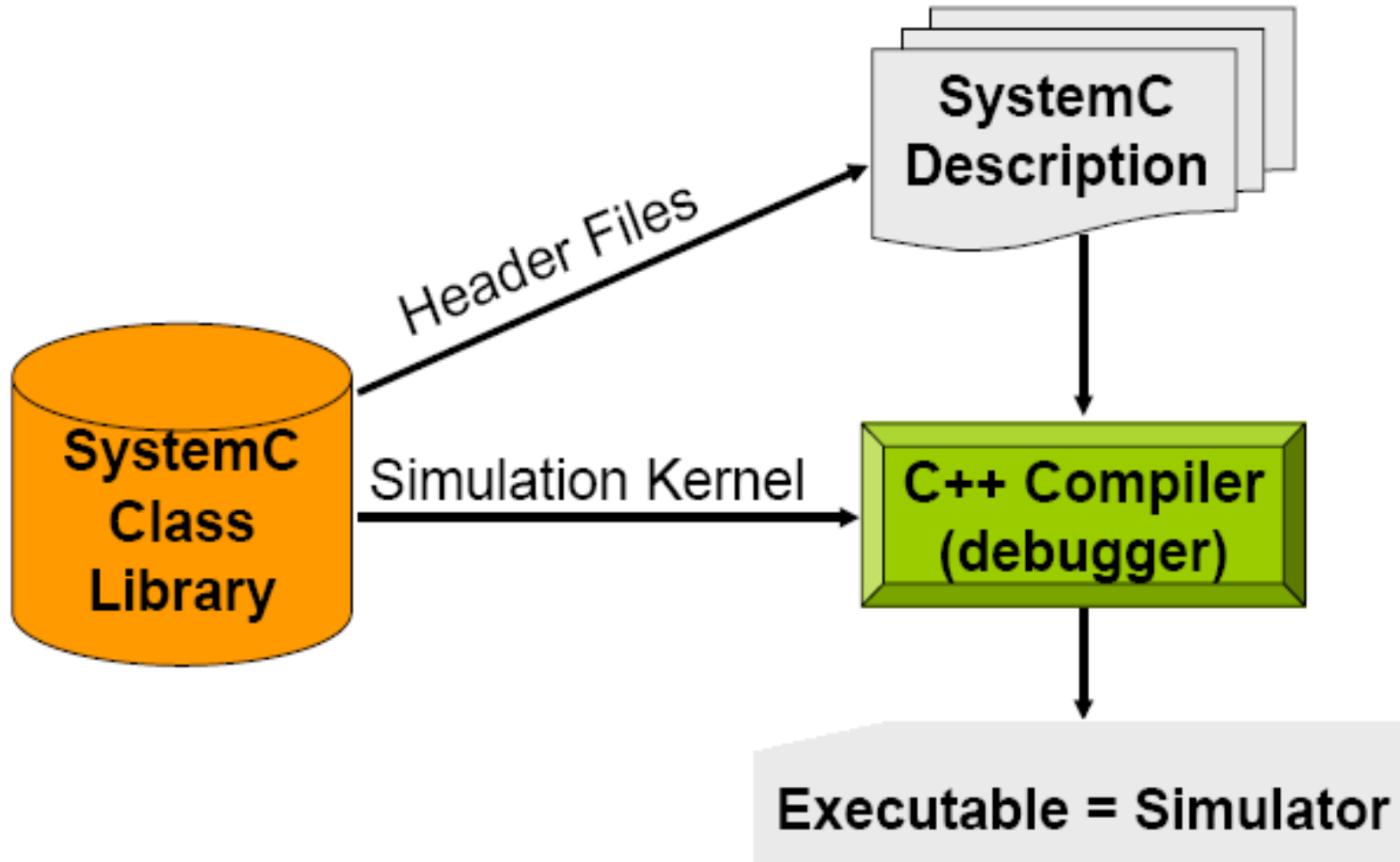
SystemC:



Realizzazione

- ... Un'estensione di C++
- Librerie per fornire funzionalità aggiuntive
- Sfrutta i meccanismi di ereditarietà del C++
- Ereditando nuovi Data-Type, si può esprimere il behavior
- SystemC è praticamente C++ e può essere tranquillamente affiancato da codice C++ puro.

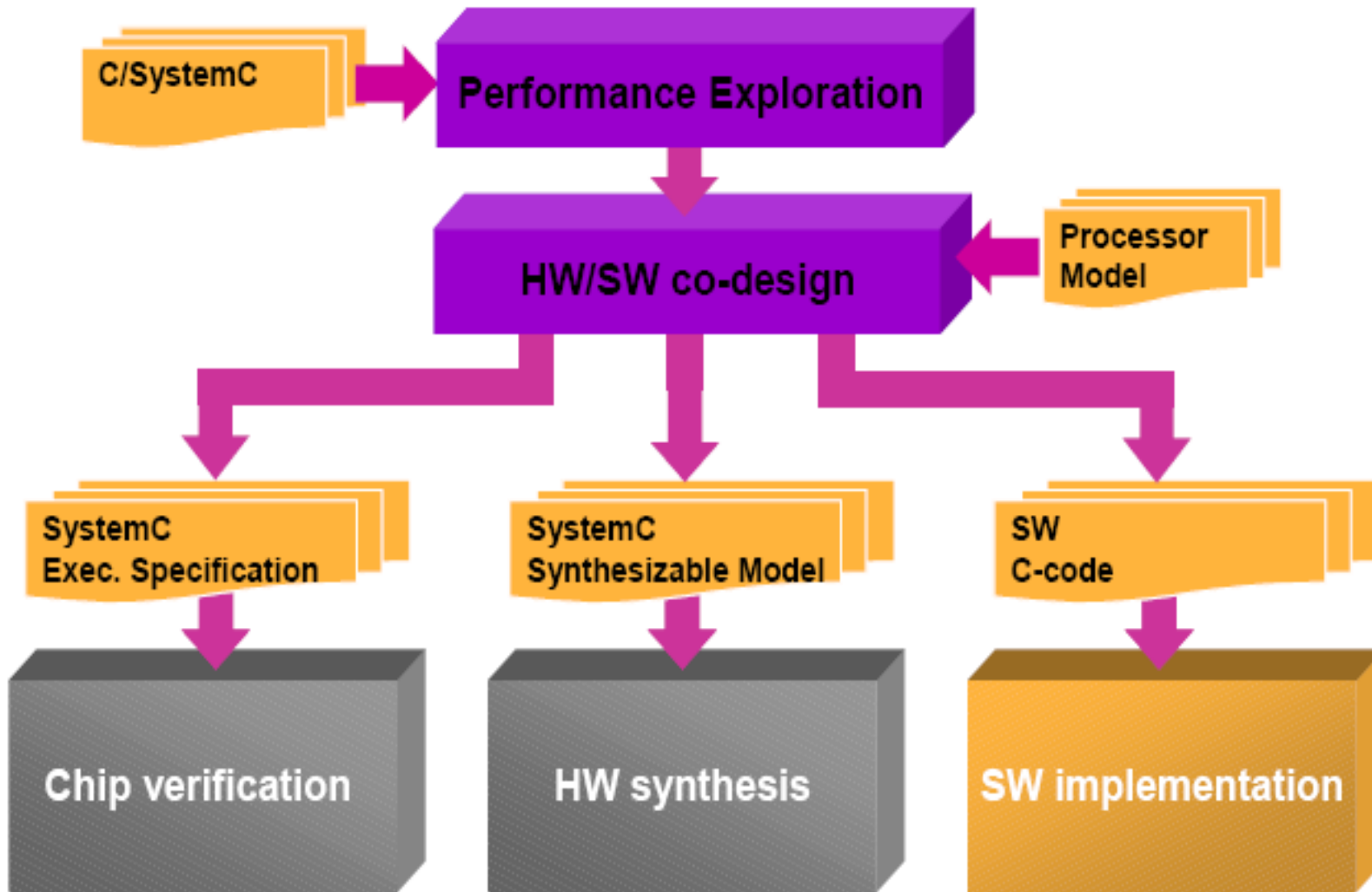
Architettura SystemC



Vantaggi del SystemC

- Unico linguaggio per tutte le fasi di progettazione
 - Facilita l'interoperabilità e l'apprendimento
- “Unico” linguaggio per sviluppo HW e SW
 - Facilita il co-design
- Permette una più veloce fase di simulazione/rifinimento
- Utilizza uno dei più diffusi linguaggi di programmazione
- Leggerezza

SystemC ToolChain



SystemC Features:

- Suporto alla concorrenza
 - Module
- Nozione di tempo
 - Clocks, wait()
- Modello di Comunicazione
 - Segnali, protocolli, handshaking
- Reattività agli eventi
 - Eventi, sensitivity list, watching()
- Data Types
 - Valori logici, bit vectors,..

SystemC: Core

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

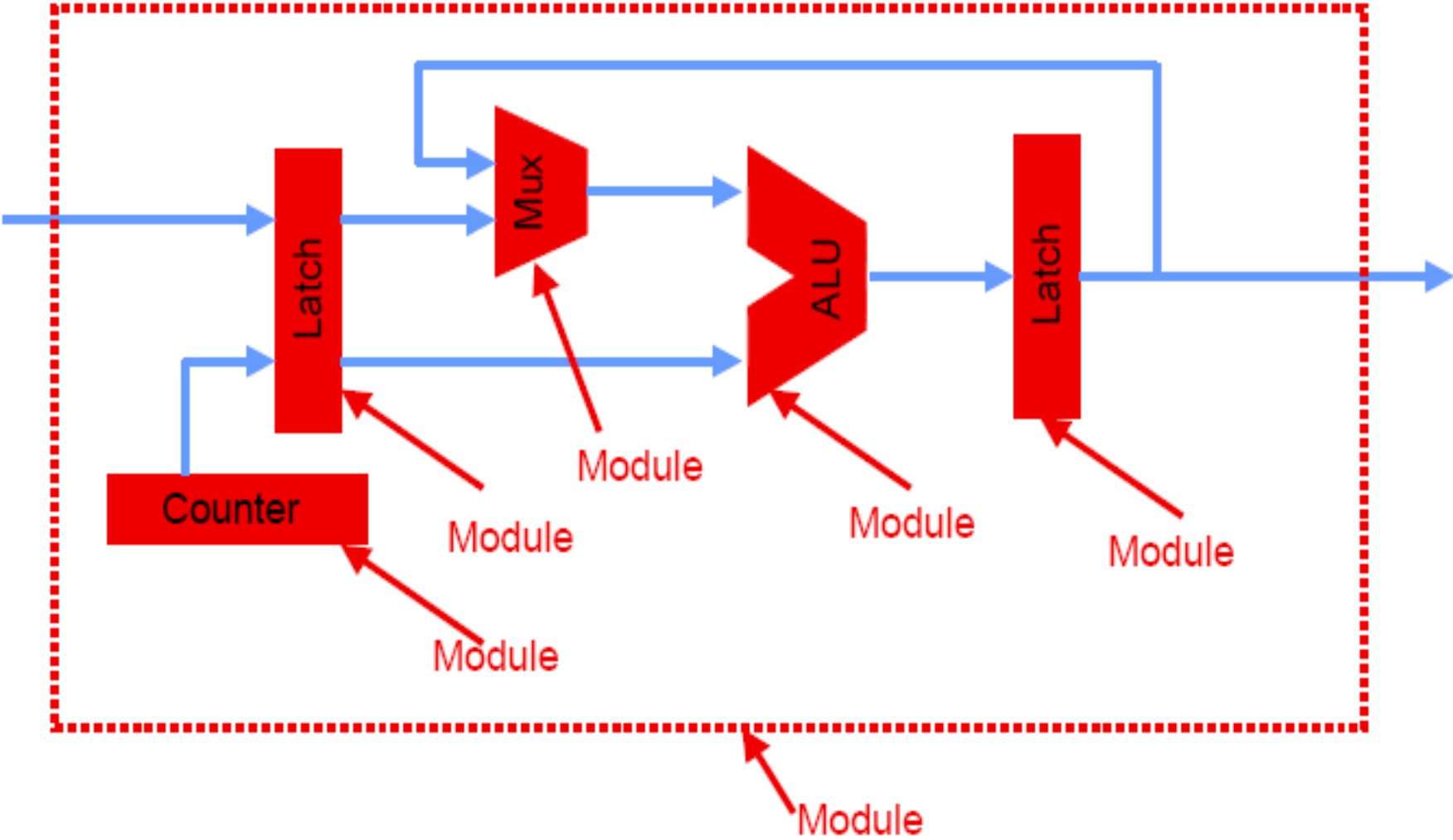
C++ user-defined

C++

Modules

- Mappano le funzionalità dei blocchi HW/SW
- Derivano dalla classe SystemC : `sc_module`
- Rappresentano I mattoncini di base di ogni sistema
- Possono contenere gerarchie di sub.modules
 - Con variabili / segnali privati
- Richiedono:
 - Comunicazione: Si interfacciano tramite
 - Ports
 - Interfaces
 - Channels
 - Funzionalità:
 - processes

Modules



Modules

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

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

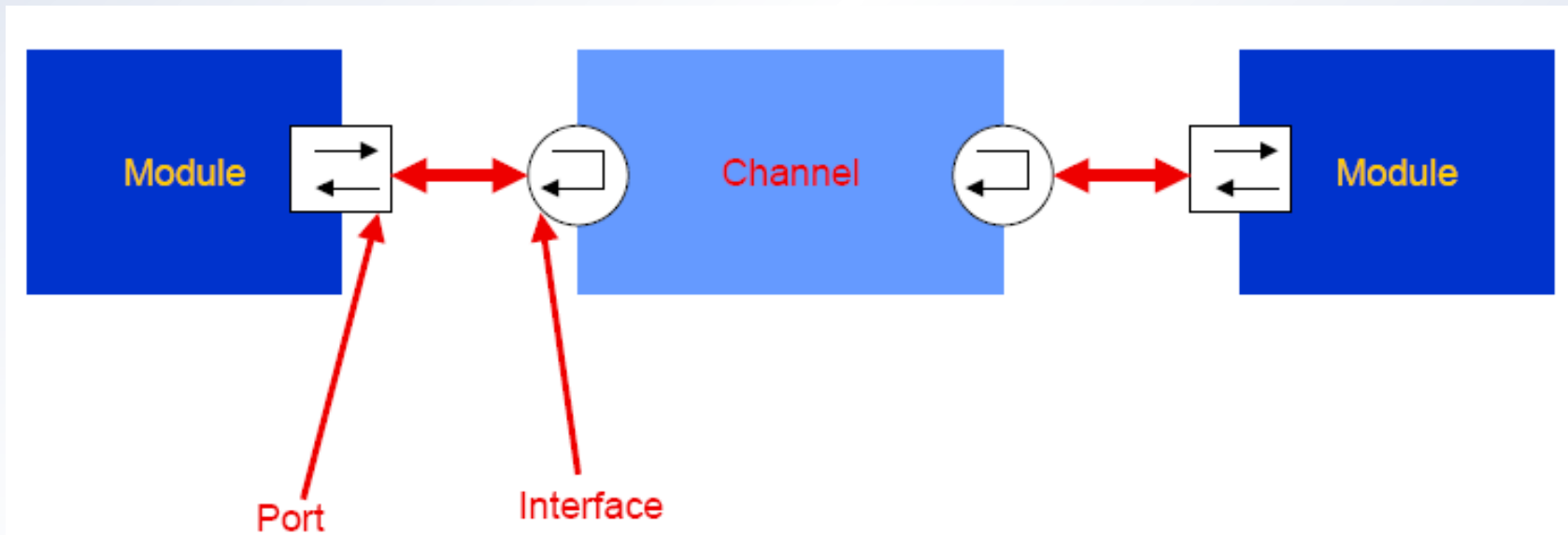
    SC_CTOR(my_module)
    {
        //map processes to member functions
        //sensitivity lists
        //initialization code
    }
};
```

Una Classe C++
con:

- variabili membro
- funzioni membro
- costruttore

Interfaces, Ports, Channels

- Elementi per la comunicazione
- Nei moduli sono definite le Porte
- A tempo di inizializzazione le porte sono associate ai channels tramite delle interfaces



interfaces

- Hanno prototipo e un paio di operazioni
 - Es: read, write
- L'implementazione è delegata nei channels
- Derivano dalla classe `sc_interface`
- Le più usate:
 - `sc_signal_in_if<T>`
 - Ha il metodo `read()` e torna un oggetto di tipo T
 - `sc_signal_inout_if<T>`
 - Deriva dal primo (e quindi ha la `read()`)
 - Dichiarata un metodo `write()` che ha come input un oggetto di tipo T passato per riferimento
 - `sc_signal_inout_if<T>`
 - E' lo stesso di `inout`

Ports

- Forniscono le funzioni di comunicazione ai moduli
- Derivate dalla classe SystemC `sc_port<class IF, int N=1>`
 - IF: tipo di interfaccia, N: numero di interfacce connesse
- Si connettono ai channel per mezzo delle interfacce
- Un channel usatissimo nei modelli RTL:
 - `sc_signal`
 - Esistono dellei shortcuts per le classi specializzate:
 - `sc_in<class T>`
 - `sc_out<class T>`
 - `sc_inout<class T>`
 - Es. Instanziiazione:
 - `sc_inout<bool>my_port;`
 - I metodi sulla porta sono disponibili grazie all'interfaccia:
 - `my_port.read()`, `my_port.write()`,

channels

- Implementano i prototipi descritti nelle interfacce
- Fanno trasferimento di dati
- Si possono dichiarare tra
 - Moduli
 - Processi nello stesso modulo
- Esempio tipico:
 - `sc_signal<T>`, derivato dall'interfaccia `sc_signal_inout_if<T>`
- Complessità arbitraria
 - Anche connessioni complete
- Altri esempi
 - `sc_fifo<T>`: implementa le interfacce `sc_fifo_in_if<T>` e `sc_fifo_out_if<T>` (nelle versioni bloccanti e non bloccanti)
 - `sc_mutex`

Istanziamento dei Porti

```
//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
    }
};
```

Processi

- Descrivono le funzionalità dei modules
- Implementati come funzioni memro
- 3 tipi:
 - SC_METHOD
 - SC_THREAD
 - SC_CTHREAD
- Implementati come Macro C++
 - Non necessariamente implementate come processo
 - Operano una registrazione allo scheduler systemC in modo trasparente
- Tutti I Process in SystemC vengono eseguiti IN CONCORRENZA!!!
- Il codice all'interno di ogni processo viene eseguito in maniera SEQUENZIALE!!!

SC_METHOD

- Sensitivo ad ogni cambiamento delle porte di input
- In genere usato per modellare reti logiche combinatorie
 - NOR, NAND, mux,
- Non può essere sospeso
 - Tutto il codice viene eseguito interamente ogni volta che il metodo viene invocato
- Non memorizza lo stato interno tra diverse invocazioni
 - A meno che non si tenga traccia di questo in variabili membro esterne

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());
};
```

SC_THREAD

- Permette al processo di essere sospeso
 - Wait() e derivate
- Ha una sensitivity list
 - La wait() termina quando viene individuato un cambiamento nelle porte della sensitivity list
- Mantiene lo stato tra invocazioni differenti
 - Ovvero l'esecuzione ricomincia da dove si era bloccato
- Utilissimo per i sistemi clockati, per le reti sequenziali e per i comportamenti multi-ciclo.
- Carico più pesante per lo schedulatore
 - Ci sono i context switches e la memorizzazione dello stato

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();
    }
};
```

SC_CTHREAD

- Simile a SC_THREAD ma implementa il “clocked thread”
- Non ce ne è bisogno e verrà deprecato
- Permette di avere un solo edge di un solo segnale sulla sensitivity list
- Utile per le simulazioni ad alto livello
- Introduce:
 - wait_until()
 - watching()

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();  
    }  
}
```

Signals

- Il tipo di canale più comune (in RTL)
- Derivato da `sc_prim_channel`
- Sono legati alle porte per mezzo delle interfacce
- Usati per connettere i moduli attraverso le porte
- Possono essere locali ai moduli
- Un segnale speciale: clock
 - `sc_clock`
 - Possono esserne istanziati più di uno anche con fasi arbitrarie

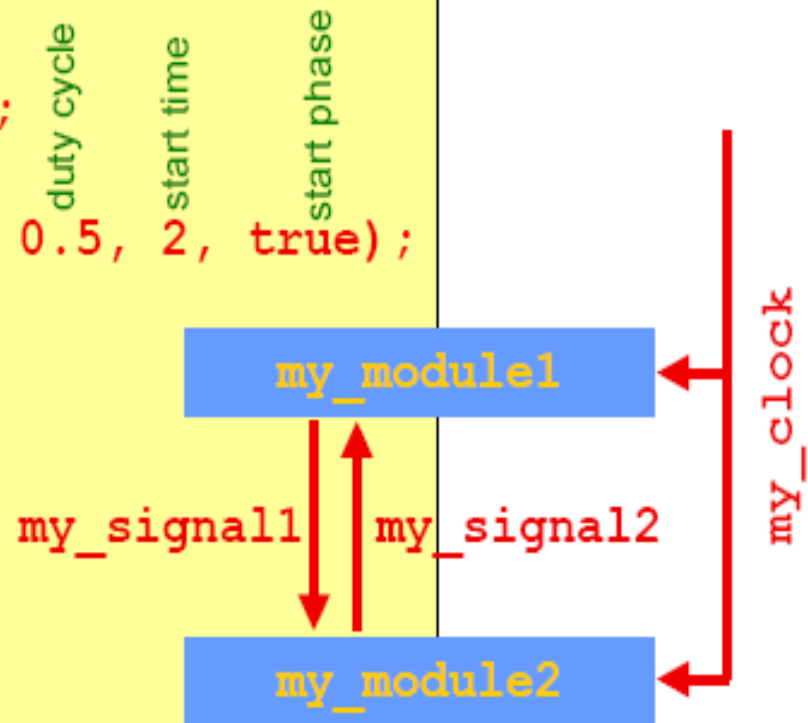
Signals: istanziazione e binding

```
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
                duty cycle  start time  start phase

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);
```



Data types

- Associati a signals e ports
- Tutti quelli del C++ (anche quelli definiti con typedef) e:
 - Scalari: `sc_bit`, `sc_logic`
 - Integer: `sc_int`, `sc_uint`, `sc_bigint`, `sc_biguint`
 - Bit e logic vector: `sc_bv`, `sc_lv`
 - Fixed point: `sc_fixed`, `sc_ufixed`. `sc_fix`, `sc_ufix`
- Operatori speciali
 - Bit selector: `x[i]`
 - Part selector: `x.range(4,2)`
 - Concatenation: `(x.range(2,1), y)`
 - ...

Tracing

- Utile per debug
- Waveform dumping
- Produce .VCD da aprire con un qualsiasi waveform analyzer

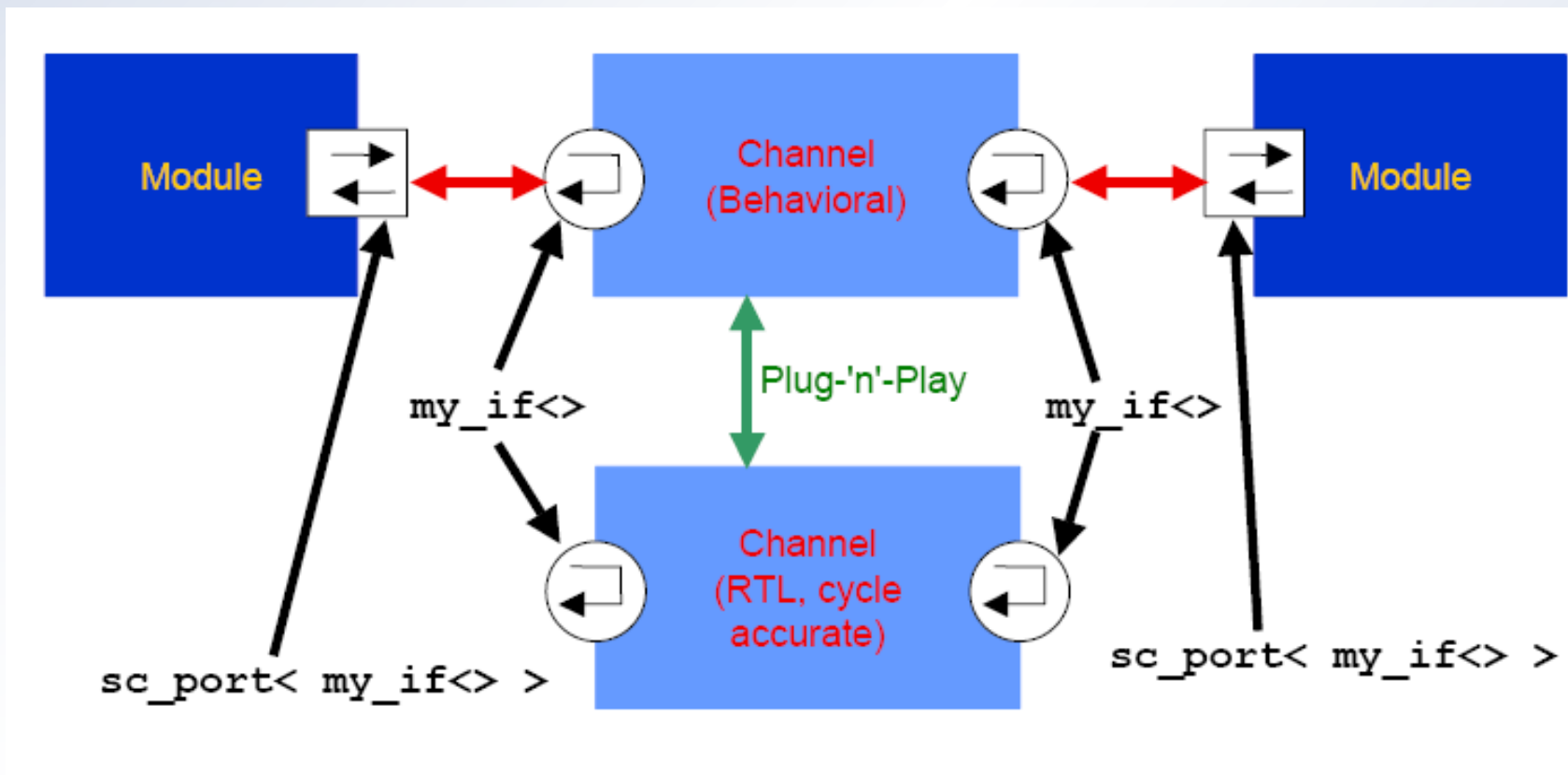
```
// main.cpp
sc_signal<bool> my_signal;
sc_trace_file *tf;
tf = sc_create_vcd_trace_file("my_waveform_filename");
sc_trace(tf, my_signal, "my_signal_name");
sc_start(-1);           // run until error/spontaneous exit
sc_close_vcd_trace_file(tf);
return(0);
```


Scheduler

- Simile a VHDL
 - Delta time e discrete time
- Timing step
 - Clock cycle, delay, ...
- Resume di processi waiting per eventi
 - Ne esegue ovviamente il corpo fino a che non vengono sospesi ancora
- Delta cycle:
 - Il tempo di simulazione non avanza
 - Update di tutti i segnali di output
 - Controlla se ci sono processi da invocare o ripristinare
- L'ordine di resume dei processi non e' deterministico
- La semantica di SystemC permette di mantenersi indipendenti dall'ordine di esecuzione dei processi

Rifinire le comunicazioni: Channel Binding

- Se 2 channels hanno la stessa interfaccia possono essere sostituite a run-time
 - E' possibile sostituire quindi channels funzionali con altri cycle-accurate etc.



Implementare le interfaces

```
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;
};
```

Implementare le interfaces

- Per implementare connessioni read/write bisogna derivare un'interfaccia
- Nell'esempio read() e write() saranno disponibili per la nuova interfaccia
- Chi implementa il channel può scegliere
 - Di implementare my_readwrite_if
 - Di implementare my_read_if E my_write_if

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

Implementare un channel

- Deve offrire funzionalità di comunicazione
- Deve offrire l'implementazione dei metodi dichiarati nelle sue interfaces
- Deve ereditare (direttamente o indirettamente) `sc_prim_channel` e avere accesso a:
 - `Void request_update()`
 - `Virtual void update() = 0`
- Deve ereditare anche `sc_interface` e accedere a:
 - `Virtual const sc_event & default_event()`
 - Dice al kernel di simulazione qual e' l'evento da accodare per le qaiting queue.

Implementare un channel

- Un channel deve essere compatibile col paradigma **Evaluate then update** del SystemC
 - Quando un metodo viene invocato il canale deve incoccare il kernel di simulazione (`request_update()`)
 - Il kernel di simulazione schedula il canale per l'update nel prossimo delta cycle
 - Il kernel invoca il metodo `update()` del channel
 - `Update()` è ereditato da `sc_prim_channel` ma l'implementazione è specifica per il channel.

Implementare un 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

Istanziare un 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);
```

RTL Models

Algorithmic model

UnTimed Functional (UTF) model

Timed Functional (TF) model

Bus Cycle Accurate (BCA) model

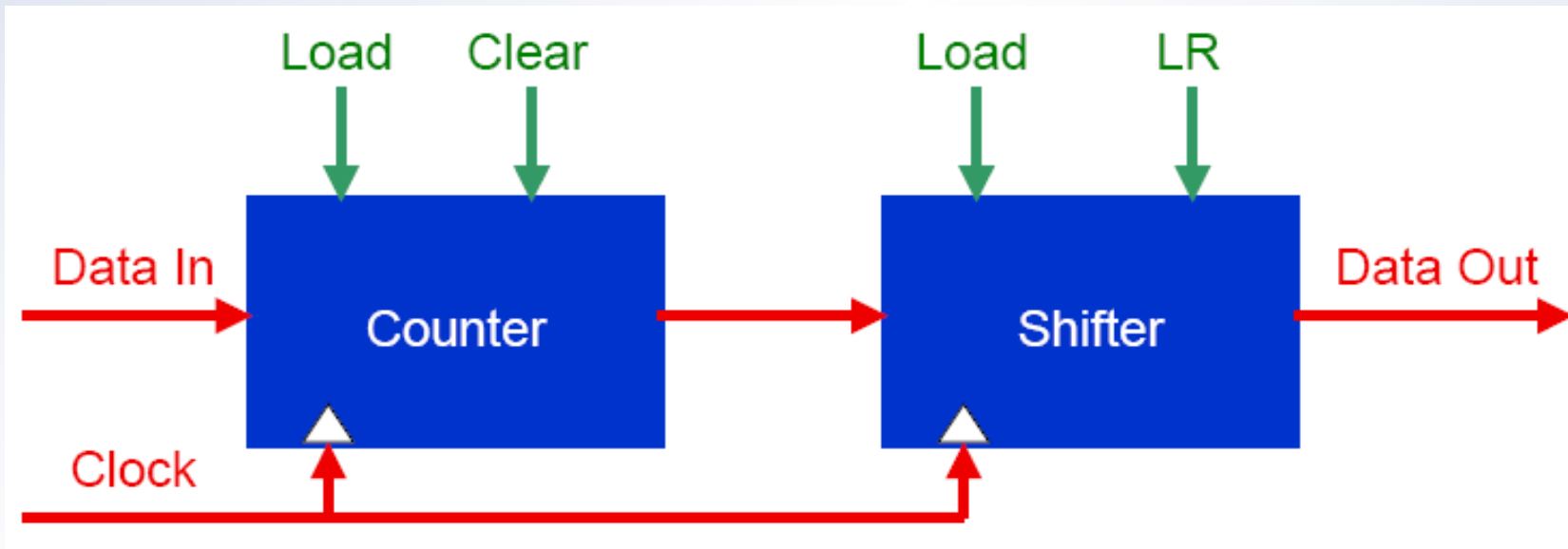
Cycle Accurate (CA) model

Register Transfer Level (RTL) model



8 bit Counter e shifter: RTL

- Non si possono usare astrazioni
- Signal, cycle, resource...



8 bit counter e shifter: RTL

```
// 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 ☹

8 bit counter e shifter: RTL

```
// 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?

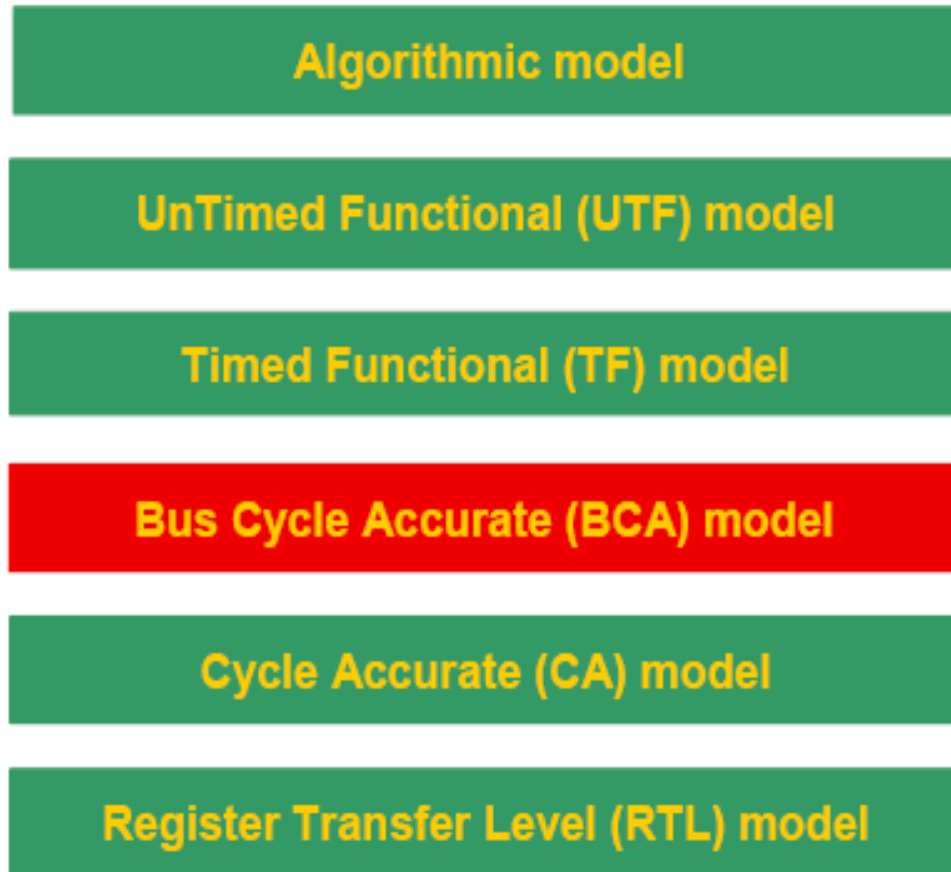
8 bit counter e shifter: RTL

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

8 bit counter e shifter: RTL

```
// 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 models



Bus Cycle Accurate models

- Porti e channel
- Non serve il cycle accurate
- Non deve essere mappato su risorse HW
- Es: MCD con algoritmo di euclide:
 - Dato $a \geq 0$ e $b > 0$
 - Se b divide a allora $\text{MCD}(a,b) = b$;
 - Altrimenti, $\text{MCD}(a,b) = \text{MCD}(b, a \bmod b)$

MCD: Bus cycle accurate

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

MCD: Bus cycle accurate

```
// 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?

Untimed Functional Model (dataflow)

Algorithmic model

UnTimed Functional (UTF) model

Timed Functional (TF) model

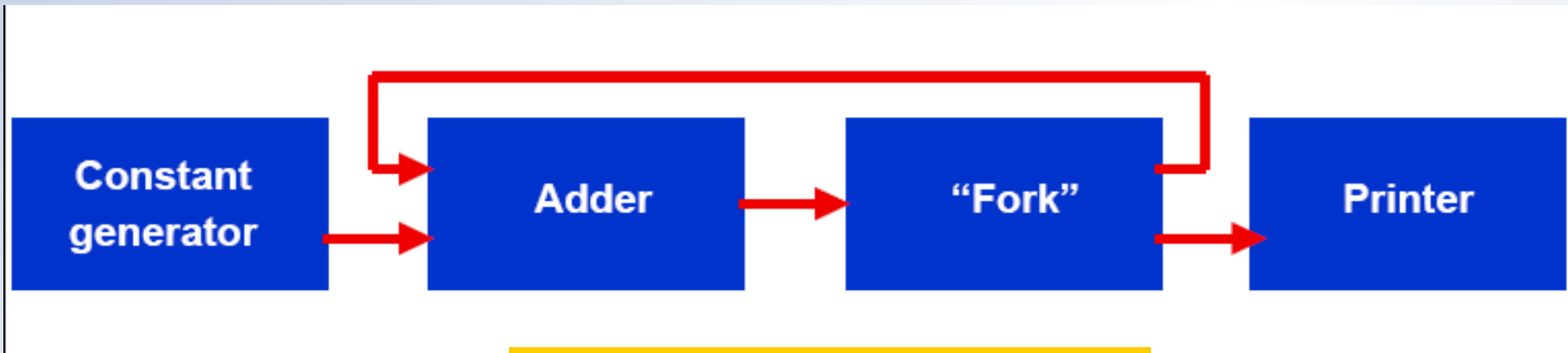
Bus Cycle Accurate (BCA) model

Cycle Accurate (CA) model

Register Transfer Level (RTL) model



UTF: dataflow



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

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

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

UTF: dataflow

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

UTF: dataflow

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

UTF: dataflow

```
// 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)
    }
}
```


Timed Functional Model (rifinisce l'UTF)

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



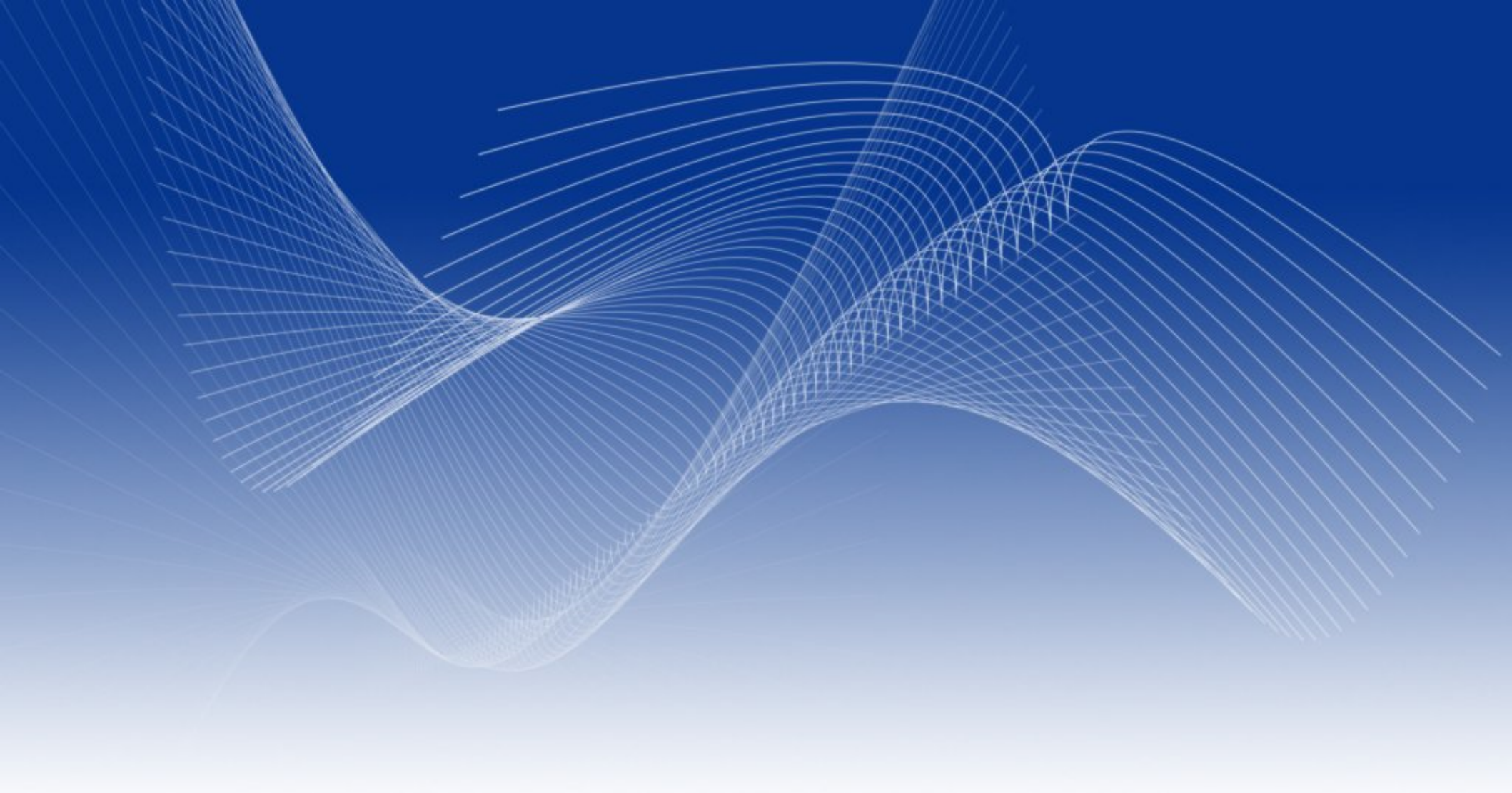
Simulation accuracy
Synthesizability

Timed Functional Model (rifinisce l'UTF)

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



Your Name
Your Title

Your Organization (Line #1)
Your Organization (Line #2)