Metodologie di Progettazione Hardware/Software:
SystemC

Motivazioni

- L'incremento della complessità dei sistemi rende inadeguato l'uso di HDLs.
- È necessario passare da una RT level a una system-level
- È necessario un linguaggio che supporti:
  - ✓ Specifica e design a diversi livelli di astrazione
  - ✓ L'inclusione di porzioni di software embedded di un sistema complesso
  - ✓ La creazione di specifiche eseguibili
  - ✓ Elevata velocità di simulazione per valutare le diverse alternative implementazioni architetturali
  - ✓ Costrutti che consentano la separazione tra funzioni del sistema e loro comunicazione
  - ✓ Basato su un consolidato linguaggio per sfruttarne conoscenze e tool

Accuratezza dei modelli

- Structural Accuracy
- Timining accuracy
- Functional accuracy
- Data organization accuracy
- Communication protocol accuracy

Livelli di astrazione

- Specifica eseguibile
  - ✓ Diretta traduzione delle specifiche
  - ✓ Indipendente dalla implementazione
- Untimed Functional Model
  - ✓ Nessuna nozione del tempo
  - ✓ Comunicazione punto-punto (usualmente mediante FIFO)
- Timed Funtional Model
  - ✓ I ritardi sono aggiunti ai process e sono annotati nelle FIFO
  - ✓ Spesso usati per un'analisi anticipata hardware-software

Livelli di astrazione

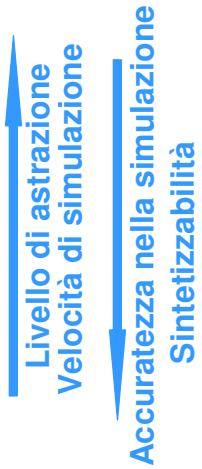
- Transaction level model
  - ✓ Comunicazione mediante function calls
  - ✓ Comunicazione accurata per la funzionalità e il timing, non strutturalmente
- Behavioral hardware model
  - ✓ Pin accurate, functional accurate.
  - ✓ Non è structural accurate e cycle accurate

Livelli di astrazione

- Pin accurate, cycle accurate hardware model
  - ✓ Functional accurate, ma non necessariamente structural accurate
- Register-Transfer Level model
  - ✓ Il modello riflette accuratamente l'organizzazione di una certa implementazione

Livelli di astrazione nella progettazione Hardware

- Algorithmic Model
- UnTimed Functional (UTF) model
- Timed Functional (TF) model
- Transaction level (TL) model
- Cycle Accurate (CA) model
- Register Transfer Level (RTL) model



Livelli di astrazione

- Algorithmic Model
- UnTimed Functional (UTF) model
- Timed Functional (TF) model
- Transaction Level (TL) model
- Cycle Accurate (CA) model
- Register Transfer Level (RTL) model

Nessuna nozione del tempo

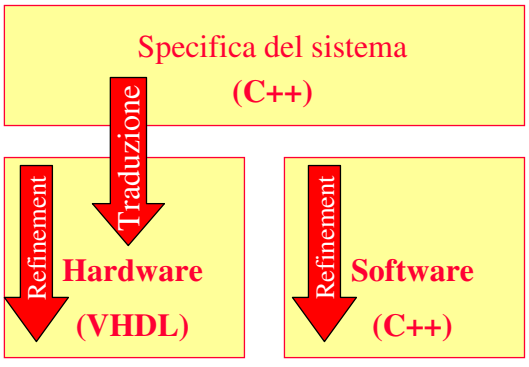
Nozione del tempo

Accuratezza a livello di ciclo

Quale linguaggio utilizzare?

- Il livelli di astrazione sono assai diversi.
- Quale linguaggio di modeling è più adatto?
- Alto livello di astrazione: Java, Visual Basic, C++
- Basso livello di astrazione: HDLs (VHDL, Verilog)
- E' presente una notevole differenza tra queste due classi di linguaggio
  - ✓ Passando da un livello di astrazione a uno più basso è necessaria una traduzione da un linguaggio ad un altro.

Metodologia di progettazione



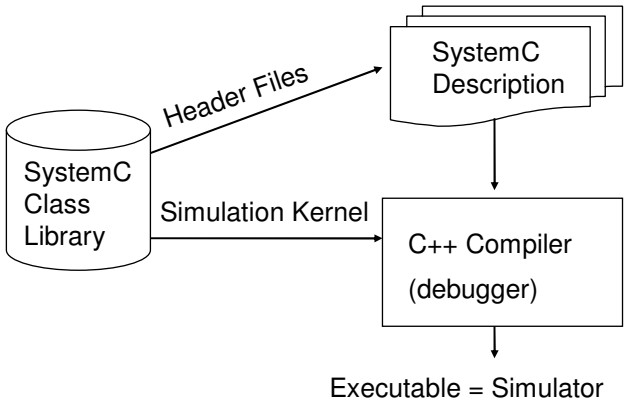
Il C++ non è sufficiente

- Non è supportata la concorrenza (HW è intrinsecamente parallelo)
- Nessuna nozione del time (clock, ritardi)
- Il modello di comunicazione è molto diverso dai modelli HW attuali (segnali)
- Manca la reattività agli eventi
- Mancano tipi di dati (logic values, bit vectors, fixed point math)

SystemC

- La libreria di classi SystemC fornisce un insieme di costrutti per modellare l'architettura del sistema
- Costruito sfruttando le caratteristiche e i tipi di dati del C++ per descrivere il comportamento HW
- Il modello HW "SystemC" è del codice C++ che può essere combinato con codice C++

Infrastruttura SystemC



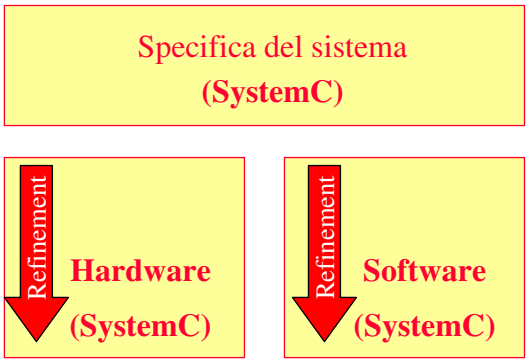
Vantaggi del SystemC

- Un unico linguaggio per tutti i passi del design
- Un unico linguaggio per lo sviluppo HW e SW
- Consente una più veloce simulazione /refinement/ modifica dei moduli
- Realizzato sopra uno dei più diffusi linguaggi di programmazione (molti tool, programmatori)
- Leggerezza

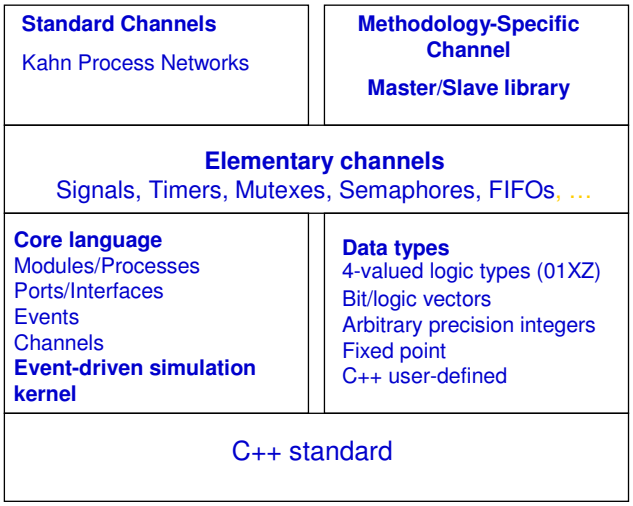
Caratteristiche del SystemC

- Supporto alla concorrenza: moduli
- Nozione del tempo: clock, wait()
- Modello di comunicazione: signali, protocolli, handshake
- Reattività agli eventi: supporto eventi, sensitivity list,..
- Tipi di dati: logic values, bit vectors, fixed point

SystemC Design Methodology



SystemC language architecture



Modello del tempo

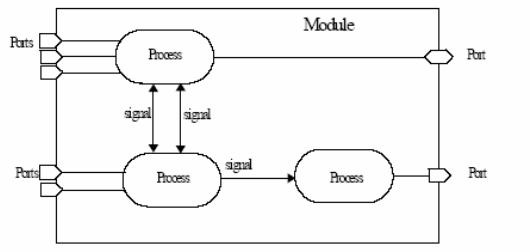
- Il SystemC usa un modello del tempo integer-valued
- Il tempo è rappresentato da un unsigned integer a 64 bit
- E' caratterizzato da una risoluzione (time resolution)
- La minima unità di tempo è il un picosecondo
- La risoluzione può essere impostata mediante la funzione **`sc_set_time_resolution()`**
  - ✓ **`Es: sc_set_time_resolution(10,SC_PS);`**
- **`sc_time`** è usata per definire un tempo
  - ✓ **`sc_time t1(42, SC_NS);`**

Moduli

- Un modulo rappresenta il blocco base per:
  - ✓ Partizionare un progetto
  - ✓ Dividere un sistema complesso in pezzi più piccoli
  - ✓ Dividere progetti complessi tra diversi progettisti
  - ✓ Consentire al progettista di nascondere la rappresentazione interna dei dati
  - ✓ Forzare il progettista ad usare interfacce pubbliche
  - ✓ Semplificare i cambiamenti e il mantenimento del sistema
- Un modulo è derivato dalla classe SystemC **`sc_module`**

Contenuto di un modulo

- Un tipico modulo contiene:
  - ✓ **Port**, **interface, channel** mediante i quali il modulo comunica con l'ambiente
  - ✓ **Process**, che descrivono la funzionalità del modulo
  - ✓ **Dati interni,** per il mantenimento dello stato interno
  - ✓ Gerarchicamente altri moduli



SC_MODULE

- Un modulo è descritto mediante la macro SC_MODULE, con il corpo del modulo tra parentesi graffe.

```
SC_MODULE (nome_module) {
  //Dichiarazione delle porte
  //Dichiarazione dei signal
  //Dichiarazione delle variable
  //Dichiarazione delle funzioni membro
  //Dichiarazione dei process
  //Costruttore del Module
  SC_CTOR (nome_module) {
      //Registrazione process
   //Dichiarazione sensitivity list
  }
};
```


SC_MODULE

```
#include <systemc.h>

SC_MODULE (adder) {
  //Dichiarazioni delle porte
  //Dichiarazione dei signal, dati interni, ecc
SC_CTOR (adder) {
      //corpo del costruttore
   //Dichiarazione dei process, sensitivity list
  }
};
```

sc_module

In alternativa

```
#include <systemc.h>

class adder : public sc_module {
  //Dichiarazioni delle porte
  //Dichiarazione dei signal, dati interni, ecc
   SC_CTOR (adder) {
       //corpo del costruttore
    //Dichiarazione dei process, sensitivity list
   }
};
```

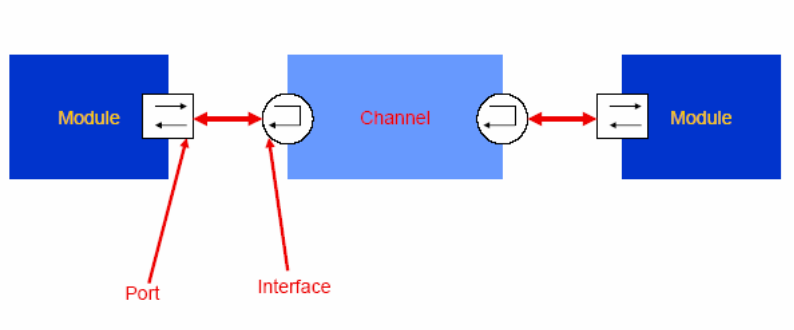

SC_CTOR e SC_HAS_PROCESS

- La macro SC_CTOR dichiara un costruttore che
  - ✓mappa le funzione membro sui process
  - ✓dichiara gli eventi a cui sono sensibili i process
- In alternativa si può usare la macro SC_HAS_PROCESS

SC_HAS_PROCESS

```
#include <systemc.h>

SC_MODULE (adder) {
  //Dichiarazioni delle porte
  //Dichiarazione dei signal, dati interni, ecc

SC_HAS_PROCESS(adder) ;
adder(sc_module_name name, int other par):
   sc_module(name) {
  // body of constructor
   // sensitivities
   }
};
```


Port, Interface e Channel



Interface

- Consiste di un insieme di operazioni
- Essa specifica solo la *signature* di ogni operazione ovvero il nome, i parametri e il valore restituito
- Non specifica come sono implementate
- Sono derivate tutte direttamente o indirettamente dalla classe astratta **`sc_interface`**

Interface: `sc_signal_in_if<T>` e `sc_signal_inout_if`

```
template <class T>
class sc_signal_in_if : virtual public sc_interface
{public:
    virtual const T& read() const = 0;
..
};

template <class T>
class sc_signal_inout_if : public sc_signal_in_if<T>
{public:
  virtual void write( const T& ) = 0;
..
};
```

Ports

- Permettono la comunicazione tra i moduli
- All'esterno sono connessi ai channel mediante le interface
- Derivate dalla classe SystemC **`sc_port`**

Ports

```
template <class IF, int N=1>
class sc_port:  ..// class derivation details
omitted
{ public:
IF* operator->();
//other member function
};
```

Ports

- Il template **`sc_port`** ha due parametri:
  - ✓l'interfaccia IF mediante la quale la porta può essere connessa a un channel
  - ✓un numero opzionale N che indica il numero massimo di interfacce che possono essere attaccate alla porta
- **`operator->()`** restituisce il puntatore all'interfaccia a cui la porta è associata

Ports

- **`sc_port< sc_signal_ inout_if<int> > p;`**
- La porta **`p`** accede a un channel mediante l'interfaccia **`sc_signal_ inout_if<int>`**
- La lettura e la scrittura dei valori del channel avviene mediante i metodi **`p->read()`** e **`p->write()`**

Ports

- Dalla classe sc_port sono derivate le seguenti porte:
  - ✓ ***sc_in<class T>***: porta di ingresso
  - ✓ ***sc_out<class T>***: porta di uscita
  - ✓ **sc_inout<class T>**: porta di ingresso/uscita
- **sc_inout** è identica a **sc_out**

```
template <class T>
class sc_in: public sc_port<sc_signal_in_if<T> > {
..
}
template <class T>
class sc_out: public sc_port<sc_signal_out_if<T> > {
..
}
```


Ports

```
SC_MODULE(Adder) {
 sc_in<int> a;
 sc_in<int> b;
 sc_out<int> c;
 // process declaration

 SC_CTOR(Adder) {
  //process
  // event sensitivity
 }
```

Channel

- Mentre le porte e le interfacce descrivono quali funzioni sono disponibili per la comunicazione, i channel descrivono come sono tali funzioni sono realizzate.
- I channel implementano le funzioni presenti nelle interfacce.
- Ad una interfaccia possono corrispondere più channel
- Ad un channel possono corrispondere più interface
  - ✓ sc_signal<T> implementa sia le interfacce di ingresso sia quelle di ingresso/uscita

Signals: sc_signal<T>

- Sono i più comuni tipi di channel (a livello RTL)
- Derivati dal channel primitivo **`sc_prim_channel`**
- Sono l'implementazione delle interfacce **`sc_signal_in_if<T>`** e **`sc_signal_inout_if<T>`**
- Usati per collegare moduli attraverso le porte
- Possono essere dentro uno specifico modulo
- Segnale speciale : clock (**`sc_clock`**). All'interno di un modulo il clock può essere instanziato una sola volta.

Signals: sc_signal<T>

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

}
```

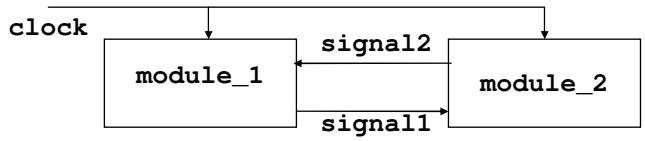

Variabili Dati membri

- In un modulo possono essere definite delle variabili membro secondo la seguente sintassi:

```
//Dichiarazione delle variable
int count_val; //Contatore interno
sc_int<8> mem[1024];//Array di sc_int
```

- Le variabili non devono essere utilizzate per la comunicazione tra moduli.

Signal: Esempio




```
sc_signal<int> signal1;
sc_signal<bool> signal2;
sc_clock clock("my_clock", 20, 0.5, 2, true);
module_1.clock_port(clock);
module_1.out_port(signal1);
module_1.in_port(signal2);
module_2.clock_port(clock);
module_2.out_port(signal2);
module_2.in_port(signal1);
```


Process

- I process descrivono il comportamento di un modulo.
- A differenza delle funzioni C/C++, la loro esecuzione avviene in modo concorrente.
- Il codice contenuto all'interno di un process viene eseguito in modo sequenziale.
- La definizione di un process è simile alla definizione di una funzione C/C++
- Un process viene dichiarato come funzione membro di una classe module e registrato come process nel costruttore del modulo.

Esecuzione di un process

- Ogni process ha associata una lista di segnali e porte di ingresso (sensitivity list) la cui variazione (evento) causa l'esecuzione del codice contenuto nel process.
- Si utilizzano ingressi sensibili:
  - ✓ ai livelli (level-sensitive) per specificare una rete combinatoria;
  - ✓ ai fronti di salita o discesa (edge-sensitive) per specificare una rete sequenziale.

Lettura/scrittura dei process

- Un process può leggere da e scrivere su porte, segnali interni e variabili.
- Per la comunicazione tra process è necessario utilizzare i segnali (channel).
- Un process causa l'esecuzione di un altro process se il primo assegna un nuovo valore al segnale che li mette in comunicazione.

Tipi di process

- SystemC prevede la dichiarazione di tre tipi di process all'interno del costruttore SC_CTOR():
  - ✓ Methods (SC_METHOD);
  - ✓ Threads (SC_THREAD);
  - ✓ Clocked Threads (SC_CTHREAD).
- Tali process vengono creati staticamente prima dell'avvio della simulazione

SC_METHOD

- E' un process che viene eseguito in seguito a un cambiamento del valore di un signal (channel) o di un ingresso (level-sensitive) o in presenza di un particolare fronte di un segnale (edge-sensitive).
- Il codice contenuto in un method process viene eseguito nella sua interezza. Un process SC_METHOD non può essere sospeso.
- Non ha memoria del proprio stato
  - ✓ Lo stato può essere conservato nelle variabili membro esplicitamente

Creazione di un process in un module

- I process sono dichiarati nel corpo di un modulo e registrati nel costruttore del modulo.

```
SC_MODULE(my_module){
   // Dichiarazione porte
   sc_in<int> a;
   …
   // Dichiarazione signal
   sc_signal<bool> c;
   // dichiarazione dei process
   void my_method_proc();
   // module constructor
   SC_CTOR(my_module){
    // registrazione process
    SC_METHOD(my_method_proc);
    //Definizione della sensitivity list
    }
};
```


Level-sensitive Process

- Questi process sono definiti usando la funzione sensitive() o sensitive e utilizzati per specificare logica combinatoria.

```
SC_MODULE(my_module) {
 sc_in<int> a;
 sc_in<bool> b;
 sc_out<int> y;
 sc_signal<bool> c;
 sc_signal<int>  d;

 void method_proc();

 SC_CTOR(my_module){
   SC_METHOD(method_proc);
      sensitive << a << b << c;
      sensitive(d); // Funzione
   }
};
```

```
void my_module::method_proc()
{
 //sequenza di istruzioni
}
```

Definizione di edge-sensitive process

- Questo tipo di process è utilizzato per specificare logica sequenziale.
- La sensitivity list deve contenere le porte di ingresso e i signal che determinano l'esecuzione del process.
- Le funzioni da utilizzare sono sensitive_pos, sensitive_neg o entrambi.
- Le porte e i signal sensibili ai fronti devono essere dichiarati come <bool>

Definizione di edge-sensitive process

```
SC_MODULE(my_module) {
 sc_in<int> a;
 sc_in<bool> b;
 sc_in<bool> clock;
 sc_out<int> y;
 // Signal
 sc_signal<bool>c;
 sc_signal<int> d;
 // Process
 void method_proc();
```

```
// Costruttore
SC_CTOR(my_module){
   // register process
   SC_METHOD(my_method_proc);
    // Sensitivity list
    sensitive_pos (clock);
    sensitive_neg << b;
   }
};
```

Lettura e scrittura di porte e signal

- Nella implementazione di un modulo è possibile leggere o scrivere utilizzando o i metodi read() e write() o mediante assegnamento.
- Nella pratica è raccomandato l'uso dei metodi di lettura/scrittura per le porte e i signal, l'operatore di assegnamento per le variabili interne.

```
// read method
 address = into.read();
// assignment
 temp1 = address;
 data_tmp = memory[address];
// write method
 outof.write(data_tmp);
// assignment
 temp2 = data_tmp;
```


Lettura e scrittura di bit delle porte e signal

- La lettura o scrittura coinvolge tutti i bit delle porte e dei signal. Non è possibile operare direttamente su singoli bit.
- Per accedere a singoli bit è necessario leggere il valore in una variabile temporanea e selezionare il bit su questa variabile.

Esempio

```
//...
sc_signal <sc_int<8> > a;
sc_int<8> b;
bool c;
b = a.read();
c = b[0];
//c=a[0]; Non è consentita in SystemC
```

SC_THREAD

- Un SC_THREAD è un process che può essere sospeso e riattivato.
- Per sospendere un process SC_THREAD in un certo punto si usa la funzione wait().
- Il process riprenderà la sua esecuzione appena ci sarà un evento nella sua sensitivity list.
- L'esecuzione riprende dall'istruzione successiva alla wait().
- Molto adatto per sistemi con clock, sistemi di memorie, per sistemi con comportamento multi-ciclo

SC_THREAD: esempio

```
//my_module.h

SC_MODULE(my_module) {
 sc_in<bool> id;
 sc_in<bool> clock;
 sc_in<sc_uint> a;
 sc_in<sc_uint> b;
 sc_out<sc_uint> c;
 void my_thread();
 SC_CTOR(my_module)  {
  SC_THREAD(my_thread);
    sensitive << clock.pos();
  }
};
```

```
//my_module.cpp

void my_module:: my_thread()
{
 while(true)
 {
  if (id.read())
    c.write(a.read());
  else
    c.write(b.read());
  wait();
 }
};
```

Un esempio: un semaforo (1/3)

```
// traff.h
#include "systemc.h"
SC_MODULE(traff) {

sc_in<bool> roadsensor;
sc_in<bool> clock;
sc_out<bool> NSred;
sc_out<bool> NSyellow;
sc_out<bool> NSgreen;
sc_out<bool> EWred;
sc_out<bool> EWyellow;
sc_out<bool> EWgreen;

void control_lights();
int i;
```

```
// Constructor
SC_CTOR(traff) {
 SC_THREAD(control_lights);
  // Thread Process
  sensitive << roadsensor;
  sensitive_pos << clock;
 }
};
```


Un esempio: un semaforo (2/3)

```
#include "traff.h"
void traff::control_lights()
{
NSred = false;
NSyellow = false;
NSgreen = true;
EWred = true;
EWyellow = false;
EWgreen = false;
while (true) {
  while (roadsensor == false)
    wait();
NSgreen = false;
NSyellow = true;
NSred = false;
```

```
for (i=0; i<5; i++)

NSgreen = false;
NSyellow = false;
NSred = true;
EWgreen = true;
EWyellow = false;
EWred = false;
for (i= 0;i<50; i++)
  wait();
```

Un esempio: un semaforo (3/3)

```
   EWgreen = false;
   EWyellow = true;
   EWred = false;
   for (i=0; i<5; i++) // times up for EW yellow
   wait();
   NSgreen = true; // set EW to red
   NSyellow = false; // set NS to green
   NSred = false;
   EWgreen = false;
   EWyellow = false;
   EWred = true;
   for (i=0; i<50; i++) // wait one more long
   wait();   // interval before allowing
             // a sensor again
   }
}
```

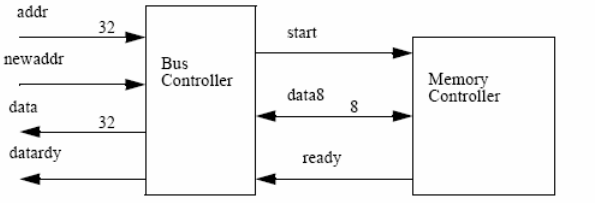

Clocked Thread Process

- Un SC_CTHREAD è un caso speciale di Thread process, che è reattivo solo ai fronti di un clock.
- E' consentito nella sensitivity list un solo fronte su un solo segnale.
- Questo tipo di process può essere utilizzato per creare delle implicite macchine a stati finiti, ovvero una macchina a stati dove gli stati del sistema non sono definiti in modo esplicito.
- Gli stati sono definiti mediante sequenze di istruzioni con interposte delle chiamate alla funzione wait().
- Contiene la funzione reset_signal_is().

SC_CTHREAD

```
//my_module.h


SC_MODULE(my_module) {
 sc_in<bool> id;
 sc_in<bool> clock;
 sc_in<sc_int> a;
 sc_in<sc_int> b;
 sc_out<sc_int> c;
 void my_thread();
 SC_CTOR(my_module)  {
  SC_CTHREAD(my_thread,clock.pos());
  }
};
```


SC_CTHREAD: un esempio



SC_CTHREAD: un esempio (1/3)

```
#include "systemc.h"
SC_MODULE(bus) {
   sc_in<bool> clock;
   sc_in<bool> newaddr;
   sc_in<sc_uint<32> > addr;
   sc_in<bool> ready;
   sc_out<sc_uint<32> > data;
   sc_out<bool> start;
   sc_out<bool> datardy;
   sc_inout<sc_uint<8> > data8;

   sc_uint<32> tdata;
   sc_uint<32> taddr;

   void xfer();

   SC_CTOR(bus) {
    SC_CTHREAD(xfer, clock.pos());
    datardy.initialize(true);}
};
```


SC_CTHREAD: un esempio (2/3)

```
#include "bus.h"
void bus::xfer() {
   while (true) {
   // wait for a new address to appear
   do
    wait();
  while(newaddr.read()== false);
   // got a new address so process it
   taddr = addr.read();
   datardy.write(false); // cannot accept new address now
   data8 = taddr.range(7,0);
   start.write(true); // new addr for memory controller
   wait();
   // wait 1 clock between data transfers
   data8 = taddr.range(15,8);
   start.write(false);
   wait();
```

SC_CTHREAD: un esempio (3/3)

```
   data8 = taddr.range(23,16);
   wait();
   data8 = taddr.range(31,24);
   wait();
   // now wait for ready signal from memory controller
    do
wait();
while(ready.read() == false);
// now transfer memory data to databus
   tdata.range(7,0) = data8.read();
   wait();
   tdata.range(15,8) = data8.read();
   wait();
   tdata.range(23,16) = data8.read();
   wait();
   tdata.range(31,24) = data8.read();
   wait();
   data.write(tdata);
   datardy = true; // data is ready, new addresses ok
   }
}
```


reset_signal_is()

- Questo costrutto permette di monitorare una specifica condizione
- Quando la condizione si verifica il controllo viene trasferito dal punto corrente all'inizio del process

reset_signal_is()

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

```
//my_module.h
SC_CTOR(my_module)
{
SC_CTHREAD(my_cthread,clock.pos());
 reset_signal_is(reset,true);
}

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


reset_signal_is(): un esempio

```
#include "systemc.h"

SC_MODULE(data_gen) {
sc_in<bool>_clk;
sc_inout<int> data;
sc_in<bool> reset;
void gen_data();
SC_CTOR(data_gen){
SC_CTHREAD(gen_data, clk.pos());
  reset_signal_is(reset,true);
}
};
```

```
#include "datagen.h"

void gen_data() {

 data = 0;

while (true) {
data = data + 1;
wait();
data = data + 2;
wait();
data = data + 4;
wait();
}
}
```

Sensitivity

- Quando l'insieme di event che determinano la riesecuzione di un process viene indicata all'interno del costruttore si parla di ***sensitivity list statica***

sensitivite << a << b;

sensitive (c);

- E' possibile rendere sensitive un process al di fuori della sua sensitivity list. In questo caso si parla di ***sensitivity dinamica***.

- Durante la simulazione un thread process può essere sospeso in attesa di un evento *e* sul quale il process desidera aspettare.
  - ✓Questo risultato è ottenuto mediante l'uso del wait() su uno specifico event *e*

```
wait(e)
```


Sensitivity dinamica

- Un thread process può essere sospeso non solo in attesa di un evento *e*, ma anche di una combinazione opportuna di eventi

```
wait ( e1 & e2 & e3);
wait ( e1 | e2 | e3);
```

- Un process può essere sospeso per una quantità di tempo predefinita

  wait(200, SC_NS);

  o in modo equivalente

  sc_time t (200, SC_NS);
  wait(t);

Sensitivity dinamica

- La sensitivity dinamica può essere applicata anche a method process invocando la ***next_trigger()*** per indicare quale evento determinerà la prossima esecuzione del process.
- A differenza del **wait()**, la ***next_trigger()*** non sospende l'esecuzione del process, disabilita momentaneamente la sensitivity statica
- Il parametri della ***next_trigger()*** sono gli stessi della ***wait().***

Creazione di un modulo con più process

- Un modulo può essere creato definendo un insieme di process che vengono registrati nel costruttore del modulo.

```
#include "systemc.h"
SC_MODULE(due_process)
{
 sc_in<int> a;
 sc_in<int> b;
 sc_in<int> c;
 sc_in<bool> clk;
 sc_out<int> ris;
 sc_signal<int> temp;

 void proc_somma();
 void proc_differenza();

 SC_CTOR(due_process){
  SC_METHOD(proc_somma);
    sensitive << a << b;
  SC_METHOD(proc_differenza)
    sensitive_pos(clk);
 }
};
```

Moduli gerarchici

- Un modulo può essere realizzato collegando tra loro le istanze di alcuni moduli.
- La creazione di un modulo gerarchico richiede:
  - ✓ la creazione dei dati membro nel modulo di più alto livello che sono puntatori ai moduli istanziati
  - ✓ l'allocazione dei moduli istanziati all'interno del costruttore del modulo dando a ciascuna istanza un nome unico
  - ✓ il collegamento delle porte dei moduli istanziati alle porte o ai signal del modulo di più alto livello.

Gerarchia di moduli: Esempio (1/2)

```
#include "systemc.h"

SC_MODULE(mod_somma) {

 sc_in<int> a;
 sc_in<int> b;
 sc_out<int> s;

 void somma()
 {
  s.write(a.read()+b.read());
 }

 SC_CTOR(mod_somma) {
   SC_METHOD(somma);
     sensitive << a << b;
   }
 };
```

```
#include "systemc.h"

SC_MODULE(mod_differenza) {

 sc_in<int> a;
 sc_in<int> b;
 sc_in_clk clk;
 sc_out<int> diff;

 void differenza()
 {
  diff.write(a.read()-b.read());
 }

 SC_CTOR(mod_differenza) {
   SC_METHOD(differenza);
     sensitive_pos(clk);
   }
 };
```

Gerarchia di moduli: Esempio (2/2)

```
#include "systemc.h"
#include "mod_differenza.h"
#include "mod_somma.h"

SC_MODULE(due_moduli) {

sc_in<int> a;
sc_in<int> b;
sc_in<int> c;
sc_in<bool>clk;
sc_out<int> ris;

sc_signal<int> temp;
mod_somma *msom;
mod_differenza *mdif;
```

```
SC_CTOR(due_moduli) {

  msom = new mod_somma("MS");
  msom->a(a);
  msom->b(b);
  msom->s(temp);

  mdif = new mod_differenza("MD");
  (*mdif) (temp,c, clk,ris);
 }

};
```


Gerarchia di moduli

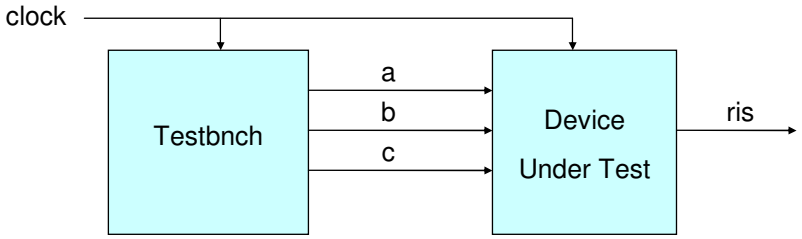
In alternativa

```
#include "systemc.h"
#include "mod_differenza.h"
#include "mod_somma.h"

SC_MODULE(due_moduli) {

sc_in<int> a;
sc_in<int> b;
sc_in<int> c;
sc_in<bool> clk;
sc_out<int> ris;

sc_signal<int> temp;
mod_somma msom;
mod_differenza mdif;
```

```
SC_CTOR(due_moduli) :
 msom("MS"), mdif("MD") {

  msom(a,b,temp);

  mdif(temp,c, clk,ris);
 }

};
```

Simulazioni

- Per realizzare la simulazione di un sistema è necessario definire un modulo di tenstbench in cui viene definito l'andamento dei segnali di ingresso.
- Tipicamente il testbench contiene un CTRHREAD process che definisce gli ingressi del Device Under Test (DUT)



Testbench: esempio

```
/* tbench.h */

#include "systemc.h"
SC_MODULE(tbench){
 sc_in<bool> clock;
 sc_out<int> a;
 sc_out<int> b;
 sc_out<int> c;

 void genera ();

 SC_CTOR(tbench) {
  SC_CTHREAD(genera, clock.pos());
 }
};
```

```
/* tbench.cpp */

 void tbench::genera()
{
  a.write(5);
  b.write(3);
  c.write(2);
  wait();
  c.write(3);
  wait();
  c.write(4);
  wait();
  c.write(5);
  wait();

 }
```

Simulazione: main

- Per potere realizzare la simulazione è necessario usare un file in cui è presente la funzione **`sc_main(int, char *arg[]).`**
- Nella sc_main() è necessario:
  - ✓ Dichiarare i channel che servono a collegare i moduli del sistema;
  - ✓ Istanziare i moduli del sistema (il DUT e il testbench)
  - ✓ Lanciare la simulazione mediante `sc_start();`
    - **`sc_start(tempo di simulazione).`**

Simulazione: Esempio (1/2)

```
#include "systemc.h"
#include "due_moduli.h"
#include "tbench.h"

int sc_main(int, char *v[])
{ /* Dichiarazione dei channel */
  sc_signal<int> a;
  sc_signal<int> b;
  sc_signal<int> c;
  sc_clock clk("Clock", 10,0.5, 0.0);
  sc_signal<int> ris;
 /* Istanziazione dei moduli */
 tbench tb("TB"); /* Istanziazione di tbench*/
 tb(clk,a,b,c);

 due_moduli dm("DueMod"); /* Istanziazione di due_moduli */
 dm(a,b,c,clk,ris);
```

Simulazione: Esempio (2/2)

```
   /* creazione ed apertura del file di trace vcd */
   sc_trace_file *tf= sc_create_vcd_trace_file("dueproc");
   /* dichiarazione degli oggetti da visualizzare */
   sc_trace(tf,clk,"CLK");
   sc_trace(tf,a,"A");
   sc_trace(tf,b,"B");
   sc_trace(tf,c,"C");
   sc_trace(tf,ris,"Ris");
   /* avvio della simulazione */
   sc_start(sc_time(160,SC_NS));
   /* chiusura del trace file vcd */
   sc_close_vcd_trace_file(tf);

return 0;
}
```


Hardware-Oriented Data-type

- I C++ data-types sono poco flessibili per la progettazione hardware
- SystemC fornisce una ricca collezione di data-type
  - ✓ Fixed-Precision
  - ✓ Arbitrary-Precision
  - ✓ Bit e Bit-Vectors
  - ✓ Four-Valued Logic e Logic-Vectors
  - ✓ Fixed-Point

Fixed-Precision Integral types

- Permettono di definire che vengono rappresentati con il numero di bit scelti dal progettista
- **`sc_int<W>`**, **`sc_uint<W>`**

  dove W rappresenta il numero di bit della sua rappresentazione

Es.

- ✓ `sc_int<4>` è un intero a 4 bit con valori da –8 a 7
- ✓ `sc_uint<6>` è un intero a 6 bit senza segno

- L'ampiezza massima è 64
- Sono i tipi di dati più veloci in SystemC

Arbitrary-precision Integral types

- Utilizzati quando non bastano 64 bit
- Esempio:

```
sc_uint<40> x,y,z,w;
w=(x*y)/z;
```

- **`sc_bigint<W>`** e **`sc_biguint<W>`**
- Es.

```
sc_uint<40> x,y,z,w;
sc_biguint<80> t=x;
t*=y;
w=t/z;
```

- Questo tipo è più lento del fixed-precision

Bit e bit vectors

- Per rappresentare i bit si può usare il tipo **`sc_bit`**
- Per rappresentare un vettore il SystemC mette a disposizione il tipo **`sc_bv<W>`**, con W illimitato.
- sc_bv non fornisce operazioni aritmetiche, è progettato per operazioni sui vettori di bit
- Sono possibili operazioni di AND (&), OR (|) e XOR(^) aventi operandi della stessa dimensione
- Sono possibili assegnamenti con stringhe o interi
  - ✓`sc_bv<16> x= "1011011011010001";`
  - ✓`sc_bv<12> y=3921;  /*111101010001*/`
- Possibili operazioni di riduzione
  - ✓`and_reduce(), or_reduce(), xor_reduce()`

Four-Valued Logic e Logic-Vectors

- **`sc_logic`** e **`sc_lv<W>`**
- Es.

```
sc_logic x,y;
sc_lv<8> A,B;
x= sc_logic_1;
y=sc_logic_Z;
A= "10101100";
B= 10;
```

Resolved types

- Sono necessari quando esistono più driver sullo stesso segnale
- sc_logic e sc_lv hanno dei corrispondenti segnali in grado di risolvere tali situazioni
  - ✓ `sc_signal_resolved` e `sc_signal_rv<W>`
- Ci sono tipi di porte per connettersi a resolved signal
  - ✓ `sc_in_resolved` e `sc_in_rv<W>`
  - ✓ `sc_out_resolved` e `sc_out_rv<W>`
  - ✓ `sc_inout_resolved` e `sc_inout_rv<W>`

Esempio tipi resolved

```
#include "systemc.h"

SC_MODULE(esempio) {

sc_in<sc_logic> a,b;
sc_out_resolved c;

void calcola1();
void calcola2();

SC_CTOR(esempio){
SC_METHOD(calcola1);
sensitive << a;
SC_METHOD(calcola2);
sensitive << b;
}
};
```

```
#include "esempio.h"


void esempio::calcola1()
{ sc_logic aa;
 aa=sc_logic_1;
 c.write(aa & a.read());
 }


void esempio::calcola2()
{ sc_logic bb;
 bb= sc_logic_0;
 c.write(bb & b.read());
 }
```