

SoC (System On Chip) Conception Haut niveau

Chapitre 2 : SystemC: Modélisation conjointe Hw/Sw des SoCs

Dr. Faten BEN ABDALLAH

Faten.benabdallah@enit.utm.tn

Année Universitaire 2024/2025



II. SystemC: Modélisation conjointe Hw/Sw des SoCs

- ➔ • **Introduction à SystemC**
 - **Elements du langage**
 - **Notion de Concurrency**
 - **Flot SLD & niveaux d'abstraction**

C++ Vs Codesign?



➤ C++ ne supporte pas:

- ▢ Communication hardware : Signaux, protocoles, ...
- ▢ Notion de temps : opérations séquencées par le temps (cycles, délais en ns ...)
- ▢ Concurrences: HW et SW opèrent en parallèle
- ▢ Réactivité: le HW répond au stimuli et il a une interaction constante avec son environnement,
- ▢ Types de données HW : Bit, bit-vector, multi-valued logic (0, 1, « x », « z »), signed et unsigned, integer, fixed-pointed, float ...
- ▢ Un cœur de simulation intégré!!

SystemC approbation ?

- ❑ Développé à l'origine par Synopsys, Frontier Design et Coware
- ❑ OSCI (Open SystemC initiative) lancé en Sep 27, 1999
- ❑ Supporté par plus que 45 sociétés avec 10 membres

Petit historique

- 2000 : SystemC 1.0 (RTL)
- 2001 : SystemC 2.0 (Communications abstraites)
- 2004 : Débuts de la bibliothèque TLM OSCI
- Mars 2007 : SystemC 2.2 (meilleur support TLM)
- Juin 2008 : TLM 2.0
- 2012: SystemC 2.3 (bibliothèque TLM intégrée)
- 2016: SystemC 2.3.2
- 2018: SystemC 2.3.3

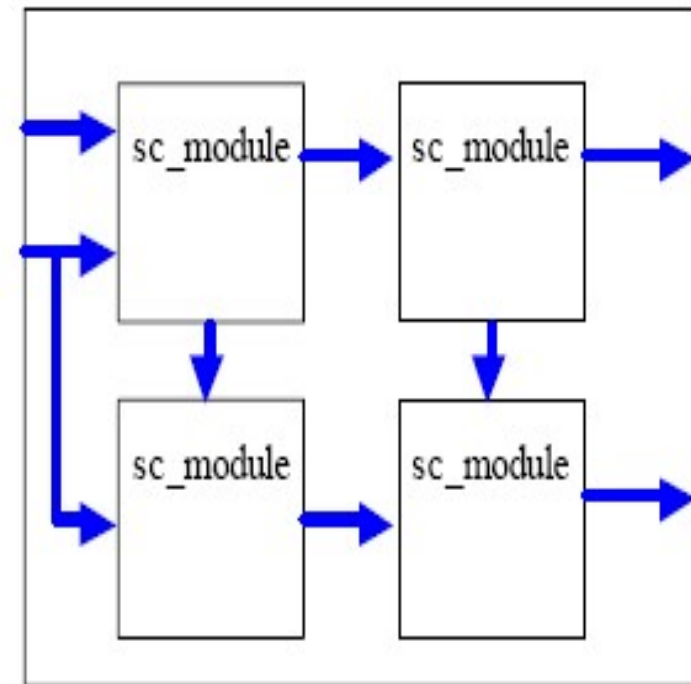
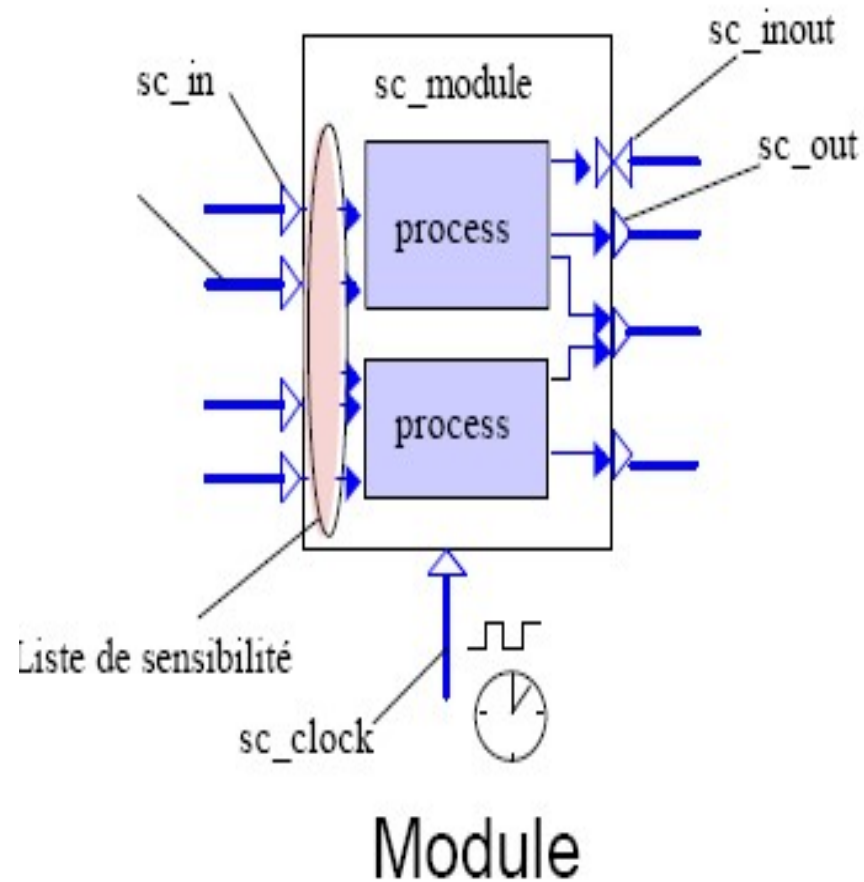
En cours de préparation : SystemC 3.0 (?)

Modélisation OS Temps-réel (RTOS)

SystemC language ou extension du C++ ?

- Il s'agit d'une extension du langage C++, sous forme d'une bibliothèque des classes C++.
- Cette bibliothèque contient aussi en standard un moteur de simulation événementiel rapide
- Les types de données usuels C++ ont été enrichis par des types de données adaptées à la modélisation de matériel (logique, décimaux virgule fixe,...).
- La généricité des types est fournie grâce aux « Templates »

Modélisation en SystemC



Hiérarchie de modules



II. SystemC: Modélisation conjointe Hw/Sw des SoCs

- Introduction à SystemC
- ➔ • **Elements du langage**
- Notion de Concurrency
- Flot SLD & niveaux d'abstraction

II. SystemC: Modélisation conjointe Hw/Sw des SoCs

Elements du langage

- Type des données
- Modules, ports et signaux
- Notion du temps

Types de données C++ => SystemC

- Les types de C++ peuvent être utilisés mais ne sont pas adéquats pour le hardware:

- `long, int, short, char, unsigned long, unsigned int, unsigned short, unsigned char, float, double, long double, and bool.`

- SystemC fournit d'autres types qui sont nécessaires pour la modélisation des systèmes hard-soft. Ils commencent par le préfixe « **SC_** »:

- Scalar boolean types: `sc_logic, sc_bit`
- Vector boolean types: `sc_bv<length>, sc_lv<length>`
- Integer types: `sc_int<length>, sc_uint<length>, sc_bigint<length>, sc_biguint<length>`
- Fixed point types: `sc_fixed, sc_ufixed`

Les types de données arithmétiques

Syntax:

<code>sc_int<length></code>	<code>variable_name ;</code>
<code>sc_uint<length></code>	<code>variable_name ;</code>
<code>sc_bigint<length></code>	<code>variable_name ;</code>
<code>sc_biguint<length></code>	<code>variable_name ;</code>

length:

- Specifies the number of elements in the array
- Must be greater than 0
- Must be compile time constant
- Use `[]` to bit select and `range()` to part select.
- Rightmost is LSB(0), Leftmost is MSB (n-1)

```
sc_int<5> a; // a is a 5-bit signed integer
sc_uint<44> b; // b is a 44-bit unsigned integer
sc_int<5> c;
c = 13; // c gets 01101, c[4] = 0, c[3] = 1, ..., c[0] = 1
bool d;
d = c[4]; // d gets 0
d = c[3]; // d gets 1
sc_int<3> e;
e = c.range(3, 1); // e gets 110 - interpreted as -2
```

Introduction aux classes *sc_bit* et *sc_bv* (1/3)

- *sc_bit* (*bit*)
 - de valeur 0 ou 1 / *sc_logic_0* ou *sc_logic_1*
 - en VHDL: *bit*
- *sc_bv*
 - vecteur de *bit*
 - en VHDL: *bit_vector*

sc_bit name ... ;
sc_bv<*bitwidth*> name ... ;

variable name : *bit*;



variable name : *bit_vector*(0 to *bitwidth*-1);
variable name : *bit_vector*(*bitwidth*-1 downto 0);

Exemple

```
sc_bit flag(SC_LOGIC_1);  
sc_bv<5> positions = "01101";  
sc_bv<6> mask = "100111";
```

```
positions.range(3,2) = "00";  
positions[2] = mask[0] ^ flag;
```

```
variable flag : bit := '1';  
variable positions : bit_vector(0 to 4) := "01101";  
variable mask : bit_vector(0 to 5) := "100111";
```

```
position(2 to 3) := "00";  
positions(2) := mask(0) xor flag;
```



Opérandes des classes *sc_bit* et *sc_bv* (2/3)

operator	function	usage	bit	bit_vector
&	bitwise AND	expr1 & expr2	√	√
	bitwise OR	expr1 expr2	√	√
^	bitwise XOR	expr1 ^ expr2	√	√
~	bitwise NOT	~expr	√	√
<<	bitwise shift left	expr << constant		√
>>	bitwise shift right	expr >> constant		√
=	assignment	value_holder = expr	√	√
&=	compound AND assignment	value_holder &= expr	√	√
=	compound OR assignment	value_holder = expr	√	√
^=	compound XOR assignment	value_holder ^= expr	√	√
==	equality	expr1 == expr2	√	√
!=	inequality	expr1 != expr2	√	√
[]	bit selection	variable[index]		√
(,)	concatenation	(expr1, expr2, expr3)		√

// Bit example

```
bool ready;
sc_bit flag = sc_bit('0');
```

```
ready = ready & flag;
```

```
if (ready == flag)
```

```
...
```

// Bit vector example

```
sc_bv<8> ctrl_bus;
ctrl_bus[5] = '0' & ctrl_bus[6];
```

```
ctrl_bus << 2; // multiply by 4
```

Méthodes des classes *sc_bit* et *sc_bv* (3/3)

method	function	usage
range()	range selection	var.range(index1,index2)
and_reduce()	reduction AND	var.and_reduce()
nand_reduce()	reduction NAND	var.nand_reduce()
or_reduce()	reduction OR	var.or_reduce()
nor_reduce()	reduction NOR	var.nor_reduce()
xor_reduce()	reduction XOR	var.xor_reduce()
xnor_reduce()	reduction XNOR	var.xnor_reduce()

// Bit vector example

```
sc_bv<8> ctrl_bus;
sc_bv<4> mult;
```

```
ctrl_bus.range(0,3) = ctrl_bus.range(7,4);
mult = (ctrl_bus[0], ctrl_bus[0], ctrl_bus[0], ctrl_bus[1]);
```

```
ctrl_bus[0] = ctrl_bus.and_reduce();
ctrl_bus[1] = mult.or_reduce();
```

sc_bv<5> active = position & mask; → variable active : bit_vector(4 downto 0);
active := positions and mask;



sc_bv<1> all = active.and_reduce(); → variable all : bit_vector(0 to 0);
all := active(0) and active(1) and active(2) and active(3) and active(4);

Introduction aux classes *sc_logic*

et *sc_lv* (1/2)

- *sc_logic*
 - de valeur 0, 1, Z ou X / *sc_logic_0*, *sc_logic_1*, *sc_logicZ* ou *sc_logic_X*
 - *sc_dt*: *Log_1*, *Log_0*, *Log_Z* ou *Log_X*,
 - en VHDL: *std_logic*
- *sc_lv*
 - *range* (), *and_reduce*(), *or_reduce*(), *nand_reduce*(), *nor_reduce*(), *xor_reduce*()
 - en VHDL: *std_logic_vector*

```
sc_logic      name ... ;  
sc_lv<bitwidth> name ... ;
```

```
variable name : std_logic;  
variable name : std_logic_vector(0 to bitwidth-1);  
variable name : std_logic_vector(bitwidth-1 downto 0);
```



Exemple

```
sc_logic buf ;  
sc_lv<8> data_drive("ZZ01XZ1Z");
```

```
variable buf : std_logic := 'Z';  
variable data_drive : std_logic_vector(7 downto 0) := "ZZ01XZ1Z";
```

```
data_drive.range (5,4) = "ZZ"; // ZZZZXZ1Z  
buf = '1';
```

```
data_drive(5 downto 4) := "ZZ";  
buf := '1'; -- ZZZZXZ1Z
```



Table de résolution des classes *sc_logic* et *sc_lv* (2/2)

AND

&	'0'	'1'	'X'	'Z'
'0'	'0'	'0'	'0'	'0'
'1'	'0'	'1'	'X'	'X'
'X'	'0'	'X'	'X'	'X'
'Z'	'0'	'X'	'X'	'X'

NOT

~	'0'	'1'	'X'	'Z'
'0'	'1'	'0'	'X'	'X'
'1'	'0'	'1'	'X'	'X'
'X'	'X'	'X'	'X'	'X'
'Z'	'X'	'X'	'X'	'X'

```
sc_logic pulse, trig;
sc_bit select = LOG_1;
```

```
pulse != select;
select = trig;
```

```
trig = SC_LOGIC_Z; // This is identical to :
trig = sc_logic('Z');
```

OR

	'0'	'1'	'X'	'Z'
'0'	'0'	'1'	'X'	'X'
'1'	'1'	'1'	'1'	'1'
'X'	'0'	'1'	'X'	'X'
'Z'	'0'	'1'	'X'	'X'

XOR

^	'0'	'1'	'X'	'Z'
'0'	'0'	'1'	'X'	'X'
'1'	'1'	'0'	'X'	'X'
'X'	'X'	'X'	'X'	'X'
'Z'	'X'	'X'	'X'	'X'

```
bool wrn;
sc_logic pena (SC_LOGIC_1); // Initialize to '1'
```

```
wrn = pena.to_bool();
```

```
if (pena.to_bool())
    cout << "pena is " << pena << endl;
```

Types de données Vs Efficacité

- *Plus d'efficacité en terme d'usage de mémoires,*
- *Plus d'efficacité en terme de vitesse d'exécution du simulateur.*

Fastest Native C/C++ Data Types (e.g., `int`, `double` and `bool`)

`sc_int<>`, `sc_uint<>`

`sc_bit`, `sc_bv<>`

`sc_logic`, `sc_lv<>`

`sc_bigint<>`, `sc_biguint<>`

`sc_fixed_fast<>`, `sc_fix_fast`,
`sc_ufixed_fast<>`, `sc_ufix_fast`

Slowest `sc_fixed<>`, `sc_fix`, `sc_ufixed<>`, `sc_ufix`

Fonctions pré-définies en SystemC

Bit Selection	<code>bit(<i>idx</i>), [<i>idx</i>]</code>
Range Selection	<code>range(<i>high</i>,<i>low</i>), (<i>high</i>,<i>low</i>)</code>
Conversion (to C++ types)	<code>to_double(), to_int(), to_int64(), to_long(), to_uint(), to_uint64(), to_ulong(), to_string(<i>type</i>)</code>
Testing	<code>is_zero(), is_neg(), length()</code>
Bit Reduction	<code>and_reduce(), nand_reduce(), or_reduce(), nor_reduce(), xor_reduce(), xnor_reduce()</code>

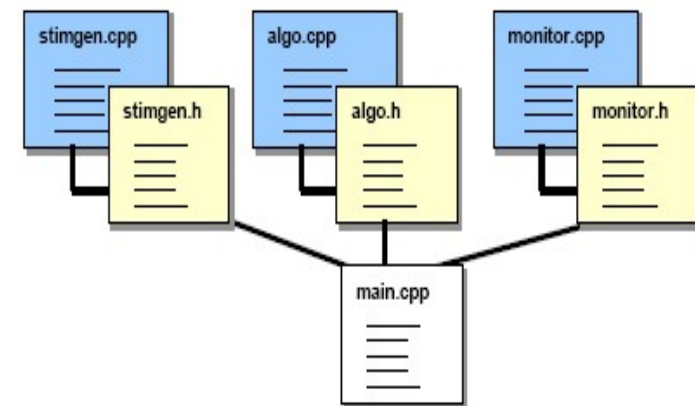
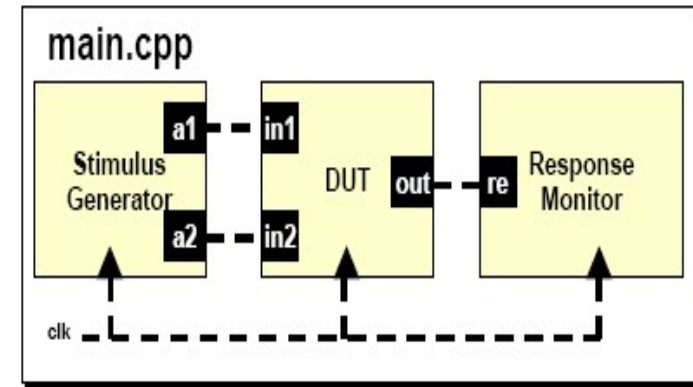
II. SystemC: Modélisation conjointe Hw/Sw des SoCs

Elements du langage

- Type de données
- Modules, ports et signaux
- Notion du temps

Module

- Un module (**SC_MODULE**) est un « conteneur ». C'est le bloc basique du SystemC.
 - Comme *entity* en VHDL/*module* en Verilog
- L'interface du module est dans le fichier header (ending.h)
- Fonctionnalités dans le CPP
- Module Contient:
 - Port
 - Signaux et variables internes
 - Processus de différents types
 - Des méthodes C++
 - Instances d'autres modules
 - Constructeur



Comment définit on un module en SystemC ?

Le langage SystemC est une extension du C++,

➔ Les modules sont définis sous la forme de classes C++,

⊙ Les modules possèdent,

➔ Une définition (type classe),

➔ Un constructeur,

➔ Un destructeur,

➔ Des attributs et méthodes,

⊙ Afin de simplifier la vie des concepteurs, des macros ont été définies.

La déclaration de la classe

```
#include "systemc.h"

class module_un : public sc_module{
private:
    // La zone privée

public:
    module_un(sc_module_name name) : sc_module(name) {
        cout << "Constructeur 1" << endl;
    }

    SC_HAS_PROCESS(module_un);

    ~module_un( ) {
        cout << "Destructeur" << endl;
    }
};
```

Le constructeur

Le destructeur

Macros: Une simplification de la syntaxe du langage C++

Description objet laborieuse ! Les constructeur doivent toujours appeler le constructeur du parent.

L'utilisation des macros définies par SystemC permet de simplifier l'écriture du code sources des modules.

```
#include "systemc.h"
```

```
class module_un : public sc_module{
```

```
private:
```

```
    // La zone privée
```

```
public:
```

```
    module_un(sc_module_name name) : sc_module(name){  
        cout << "Constructeur 1" << endl;  
    }
```

```
    SC_HAS_PROCESS(module_un);
```

```
    ~module_un( ){  
        cout << "Destructeur" << endl;  
    }
```

```
};
```

```
SC_MODULE(module_deux)
```

```
{  
private:
```

```
    // La zone privée
```

```
public:
```

```
    SC_CTOR(module_deux)  
    {  
    }
```

```
    ~module_deux( ){  
        cout << "Destructeur" << endl;  
    }
```

```
};
```

Syntaxe Module

Syntax:

```
SC_MODULE(module_name) {  
    // body of module  
};
```

EXAMPLE:

```
SC_MODULE(my_module) {  
    // body of module  
};
```

Code in header file:
my_module.h

Note the semicolon at the
end of the module definition

Rq: SC_MODULE est une macro

define SC_MODULE(x) class x: SC_Module

Ports

- Les ports sont déclarés dans le module
 - La direction du port est spécifiée par son type:
 - Input **sc_in<>**
 - Output **sc_out<>**
 - Inout **sc_inout<>**
- Le type de donnée est passé en paramètre template:



```
SC_MODULE (foo)
{
    sc_in<sc_bit> i1, i2;
    sc_out<bool> o1;
    sc_inout<bool> io2;
    ...
}
```



```
entity foo is
    port(i1, i2 : in bit;
         o1 : out bit;
         io2 : inout bit);
end entity;
```

Signaux

- Déclarés à l'intérieur du module ou à l'extérieur.
- Les données sont passées en paramètre (template)

```
SC_MODULE(my_mod) {  
    sc_signal<t_data>    signal_name;  
};
```

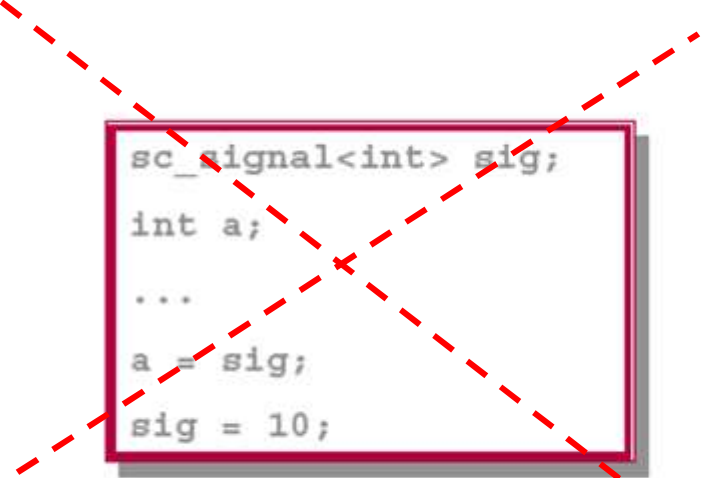

Exemple ports et Signaux

Les ports et les signaux sont deux données membres du SC_MODULE

```
SC_MODULE(module_name) {  
    // ports  
    sc_in<int> a;  
    sc_out<bool> b;  
    sc_inout<sc_bit> c;  
  
    // signals  
    sc_signal<int> d;  
    sc_signal<char> e;  
    sc_signal<sc_int<10> > f;  
  
    // rest  
};
```

Note: a space is required by the C++ compiler

Lecture /Ecriture port et signaux



```
sc_signal<int> sig;  
int a;  
...  
a = sig;  
sig = 10;
```

```
sc_signal<int> sig;  
int a;  
...  
a = sig.read();  
sig.write(10);
```

recommended

Variables

- Déclarés seulement à l'intérieur du module.
- Les données sont passées en paramètre (template)

Example:

```
SC_MODULE(count) {  
    // ports & signals not shown  
    int count_val; // internal data stroage  
    sc_int<8> mem[512] // array of sc_int  
    // Body of module not shown  
};
```

Constructeur

- **Each module has a constructor**
 - **Called at the instantiation of the module**
 - initialize internal data structure (e.g. check port-signal binding)
 - initialize all data members of the module to a known state
 - **Instance name passed as argument**
 - useful for debugging/error reporting
 - **Processes are registered inside the constructor (more later)**
 - **Sub-modules are instantiated inside the constructor (more later)**

Example:

```
SC_MODULE (my_module) {  
    // Ports, internal signals, processes, other methods  
    // Constructor  
    SC_CTOR(my_module) {  
        // process registration & declarations of sensitivity lists  
        // module instantiations & port connection declarations  
    }  
};
```

Code in header file:

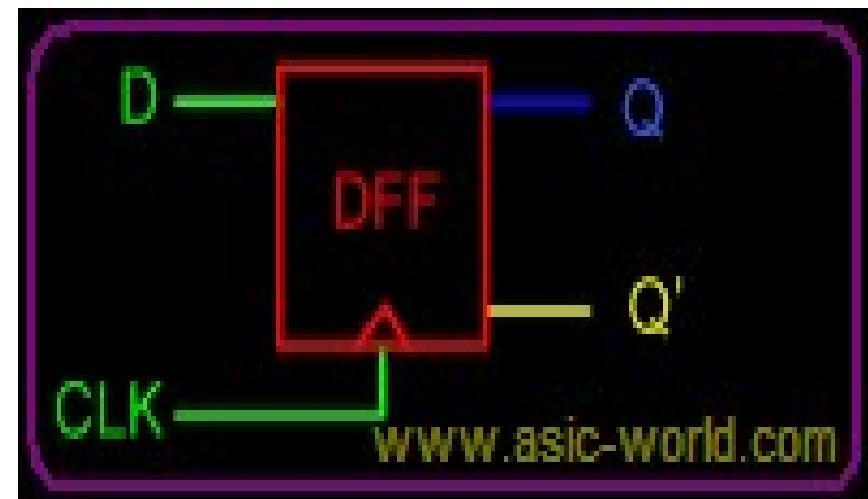
module_name.h

Liste de sensibilité

- To define the sensitivity list for any process type use
 - **sensitive** with the **()** operator
 - takes a single port or signal as an argument
 - e.g. `sensitive(sig1); sensitive(sig2); sensitive(sig3);`
 - **sensitive** with stream notation (**operator <<**)
 - takes an arbitrary number of arguments
 - e.g. `sensitive << sig1 << sig2 << sig3;`
 - **sensitive_pos** with either **()** or **<<** operator
 - defines sensitivity to positive edge of boolean signal or clock
 - e.g. `sensitive_pos << clk;`
 - **sensitive_neg** with either **()** or **<<** operator
 - defines sensitivity to negative edge of boolean signal or clock
 - e.g. `sensitive_neg << clk;`

Exemple 1 : Bascule D (DFF.h)

```
1 // D-FF Code
2 #include "systemc.h"
3
4 SC_MODULE(d_ff) {
5     sc_in<sc_logic> din;
6     sc_in_clk clock;
7     sc_out<sc_logic> dout;
8
9     void doit() {
10         dout.write(din.read());
11     }
12
13     SC_CTOR(d_ff) {
14         SC_METHOD(doit);
15         sensitive_pos << clock;
16     }
17 };
```



II. SystemC: Modélisation conjointe Hw/Sw des SoCs

Elements du langage

- Type de données
- Modules, ports et signaux
- Notion du temps

Notion du temps: Le type SC_time

Units

SC_SEC	seconds
SC_MS	milliseconds
SC_US	microseconds
SC_NS	nanoseconds
SC_PS	picoseconds
SC_FS	femtoseconds

default : t1(0, SC_SEC)



1

```
sc_time last_clk;
```

2

```
sc_time period (8.2, SC_NS); // period = 8.2 ns
```

3

```
sc_time measure (period); // clk = 8.2 ns
```

```
last_clk = sc_time(2, SC_US); // last_clk = 2 us
```


Notion du temps

- ✦ `sc_start()` method : performs simulation

```
void sc_start();  
void sc_start( const sc_time& );  
void sc_start( double, sc_time_unit );
```



```
sc_start(); // Run forever  
sc_start(SC_ZERO_TIME); // Run 1 delta delay  
sc_start(8, SC_MS); // Run 8 ms
```

- ✦ `sc_stop()` method : stop simulation

- ✦ `sc_time_stamp()` method : current time

```
sc_time t = sc_time_stamp();
```

- ✦ `sc_simulation_time()` method : current time as a double

```
double t = sc_simulation_time();
```

- ✦ `sc_delta_count()` method : counts the number of delta cycles
(return a value of uint64 type)

- ✦ `sc_set_time_resolution()` method : resolution (positive power of ten)

- ✦ `sc_get_time_resolution()` method : get the time resolution

- ✦ `sc_set_default_time_unit()` method : default time unit (power of ten)

- ✦ `sc_get_default_time_unit()` method : get default time unit

Exemple `sc_time_stamp()`

✦ Example : `sc_time_stamp()`

```
1 #include <systemc.h>
2
3 int sc_main (int argc, char* argv[]) {
4     cout<<"Current time is "<< sc_time_stamp () << endl;
5     sc_start(1);
6     cout<<"Current time is "<< sc_time_stamp () << endl;
7     sc_start(100);
8     cout<<"Current time is "<< sc_time_stamp () << endl;
9     sc_stop ();
10    cout<<"Current time is "<< sc_time_stamp () << endl;
11    return 0; // Terminate simulation
12 }
```

You could download file `sc_time_stamp.cpp` [here](#)

✦ Simulation Output : `sc_time_stamp()`

```
Current time is 0 s
Current time is 1 ns
Current time is 101 ns
SystemC: simulation stopped by user.
Current time is 101 ns
```

La fonction `sc_main`

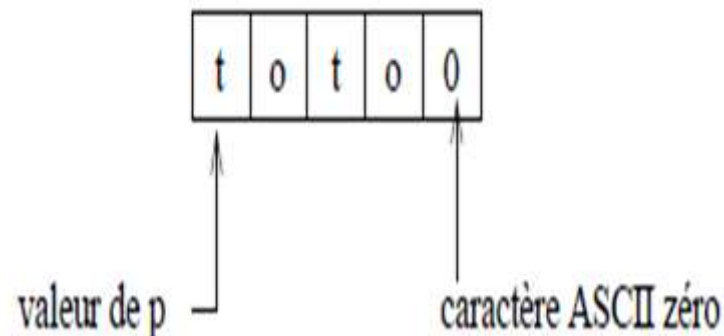
Le prototype de la fonction **`sc_main`** est:

`int sc_main(int argc, char*argv[]);`

Les variables `argc`(argument count) et `argv`(argument value) représentent respectivement le nombre d'arguments en ligne de commande et leurs valeurs.

`Char*` : pointeur vers des caractères. Ce type est souvent utilisé dans le langage C pour désigner une chaîne de caractères se terminant par 0.

Exemple `char* P=" toto"` fait pointer p vers une zone mémoire de 5 caractères.



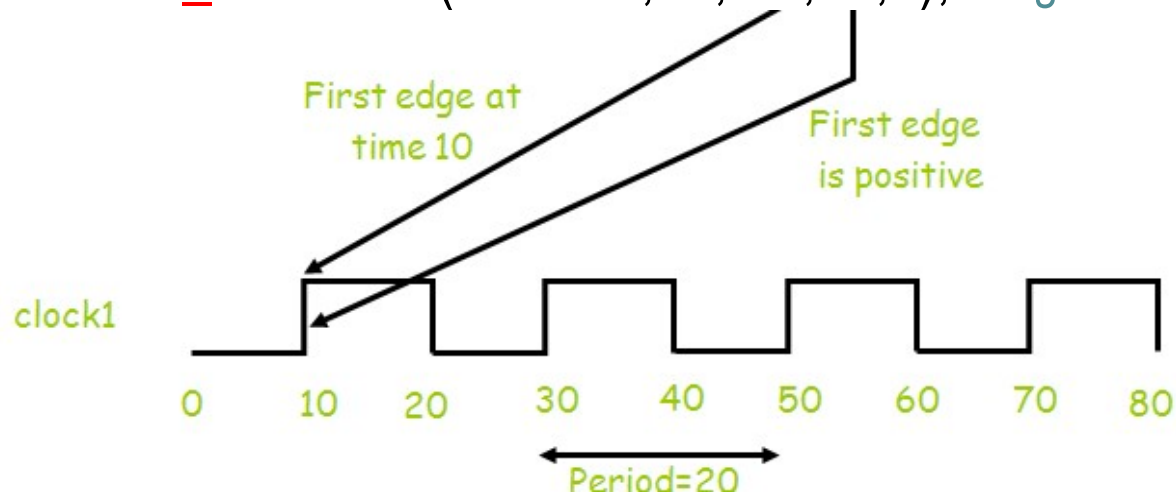
Exercice 1

Donner le résultat d'exécution
du code suivant:

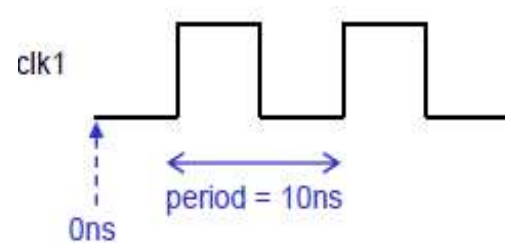
```
3 int sc_main (int argc, char* argv[]) {
4     // sc_set_time_resol... should be called before sc_time
5     // variable decleration
6     sc_set_time_resolution(1, SC_PS) ;
7     // Declare the sc_time variables
8     sc_time          t1(10, SC_NS) ;
9     sc_time          t2(5, SC_PS) ;
10    sc_time          t3, t4(1, SC_PS), t5(1, SC_PS) ;
11    // Print all the variables
12    cout << "Value of t1 " << t1.to_string() << endl;
13    cout << "Value of t2 " << t2.to_string() << endl;
14    cout << "Value of t3 " << t3.to_string() << endl;
15    cout << "Value of t4 " << t4.to_string() << endl;
16    cout << "Value of t5 " << t5.to_string() << endl;
17    // Start the sim
18    sc_start(0) ;
19    sc_start(1) ;
20    // Get the current time
21    t3 = sc_time_stamp() ;
22    cout << "Value of t3 " << t3.to_string() << endl;
23    // Run for some more time
24    sc_start(20) ;
25    // Get the current time
26    t4 = sc_time_stamp() ;
27    // Print the variables for new time
28    cout << "Value of t4 " << t4.to_string() << endl;
29    // This is how you do arth operation
30    t5 = t4 - t3;
31    cout << "Value of t5 " << t5.to_string() << endl;
32    // This is how we do compare operation
33    if (t5 > t2) {
34        cout << " t5 is greated then t2" << endl;
35    } else {
36        cout << " t2 is greated then t5" << endl;
37    }
38    return 1;
39 }
```

Clocks (1/2)

- L'horloge hérite de la classe C++- **sc_clock**
- **sc_clock**(*name, period, duty_cycle, start_time, positive_first*)
 - *name*: clock name, type: char*, default value: none
 - *period*: clock period, type: double, default value: 1
 - *duty_cycle*: clock duty_cycle, type: double, default value: 0.5
 - *start_time*: time of first edge, type: double, default value: 0
 - *positive_first*: first edge positive, type: bool, default value: true
- **sc_in_clk** clk1; //creating clock port
- **sc_clock** clk1("clock1",20,0.5,10,1); //generating clock

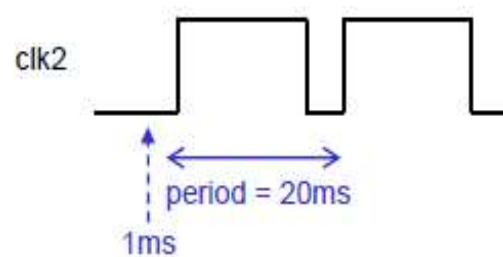


Clocks (2/2)



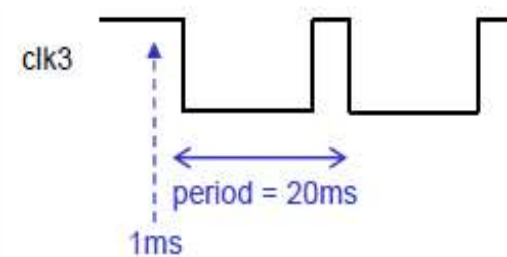
```
sc_clock("clk1", 10, SC_NS);
```

Period = 10 ns
Duty cycle = 50% (default)
StartTime = 0 s (default)
Posedge_first = true (default)



```
sc_clock("clk2", 20, SC_MS, 0.75, 1, SC_MS, true);
```

Period = 20 ms
Duty cycle = 75%
StartTime = 1 ms
Posedge_first = true



```
sc_clock("clk3", 20, SC_MS, 0.25, 1, SC_MS, false);
```

Period = 20 ms
Duty cycle = 25%
StartTime = 1 ms
Posedge_first = false

Counter Example

```
int sc_main(int argc, char* argv[])
{
    sc_clock clk1("clk1", 10, SC_MS); // Period = 10 ms
    counter my_counter("counter1");
    my_counter.clk(clk1);

    sc_start(200, SC_MS);
    return 0;
}
```

```
SC_MODULE(counter)
{
    sc_in_clk clk;
    int count;

    SC_CTOR(counter): count(0)
    {
        SC_METHOD(counter_method);
        sensitive<<clk.pos();
        dont_initialize();
    }

    void counter_method()
    {
        cout<<"@ " << sc_time_stamp() << " count= " << count++<<endl;
    }
};
```



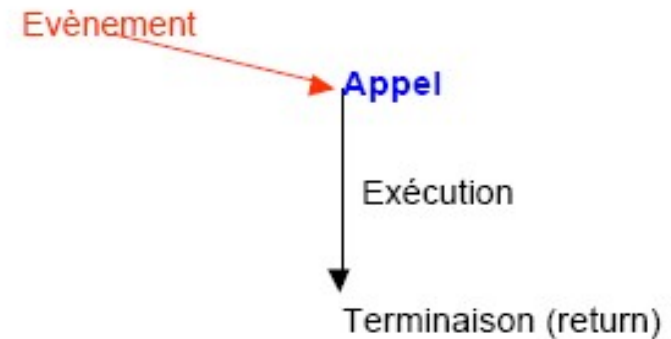
II. SystemC: Modélisation conjointe Hw/Sw des SoCs

- Introduction à SystemC
- Elements du langage
- ➔ • **Notion de Concurrency**
- Flot SLD & niveaux d'abstraction

Processus (équivalent process en VHDL)

■ SC_METHOD

Méthode C++



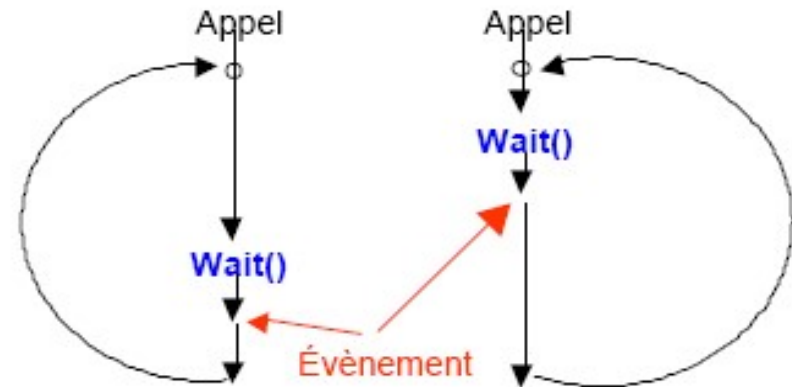
**Évènement = variation signal
sur liste de sensibilité statique ou dynamique**

■ SC_THREAD

Thread sensible à tout signal

■ SC_CTHREAD


Thread sensible uniquement
à un front de l'horloge.



SC_THREAD

- Un **SC_THREAD** peut être suspendu par des instructions wait()
⇒ liste de sensibilité dynamique
- Un SC_THREAD doit contenir une boucle infinie afin d'être ré-exécuté.

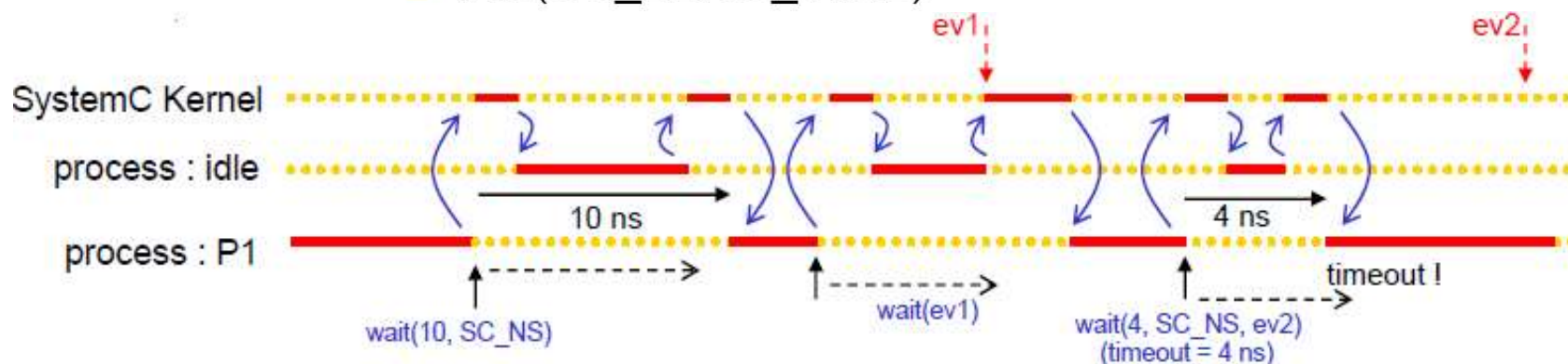
 wait() //wait on events in static sensitivity list

 wait(e1| e2| e3) //wait on events e1,e2 or e3
// No way to know which event occurs !

 wait(200,SC_NS) // wait for 200 ns

 wait(200,SC_NS,e1&e2&e3) // wait on events e1,e2&
e3, timeout after 200 ns

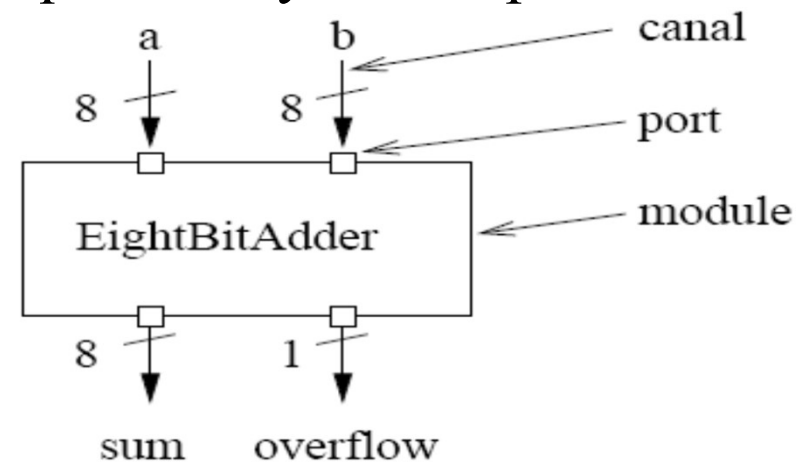
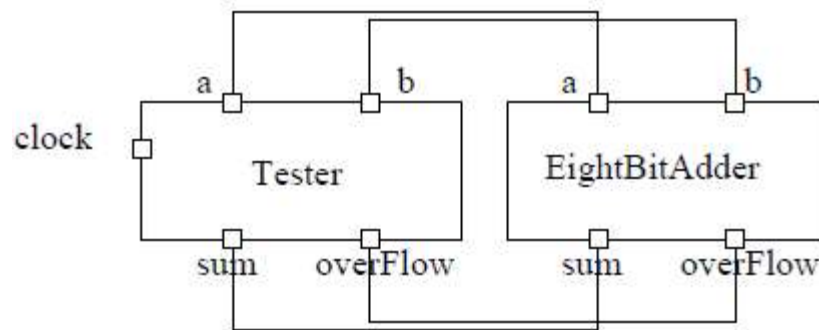
 wait(SC_ZERO_TIME)



EXERCICE 2

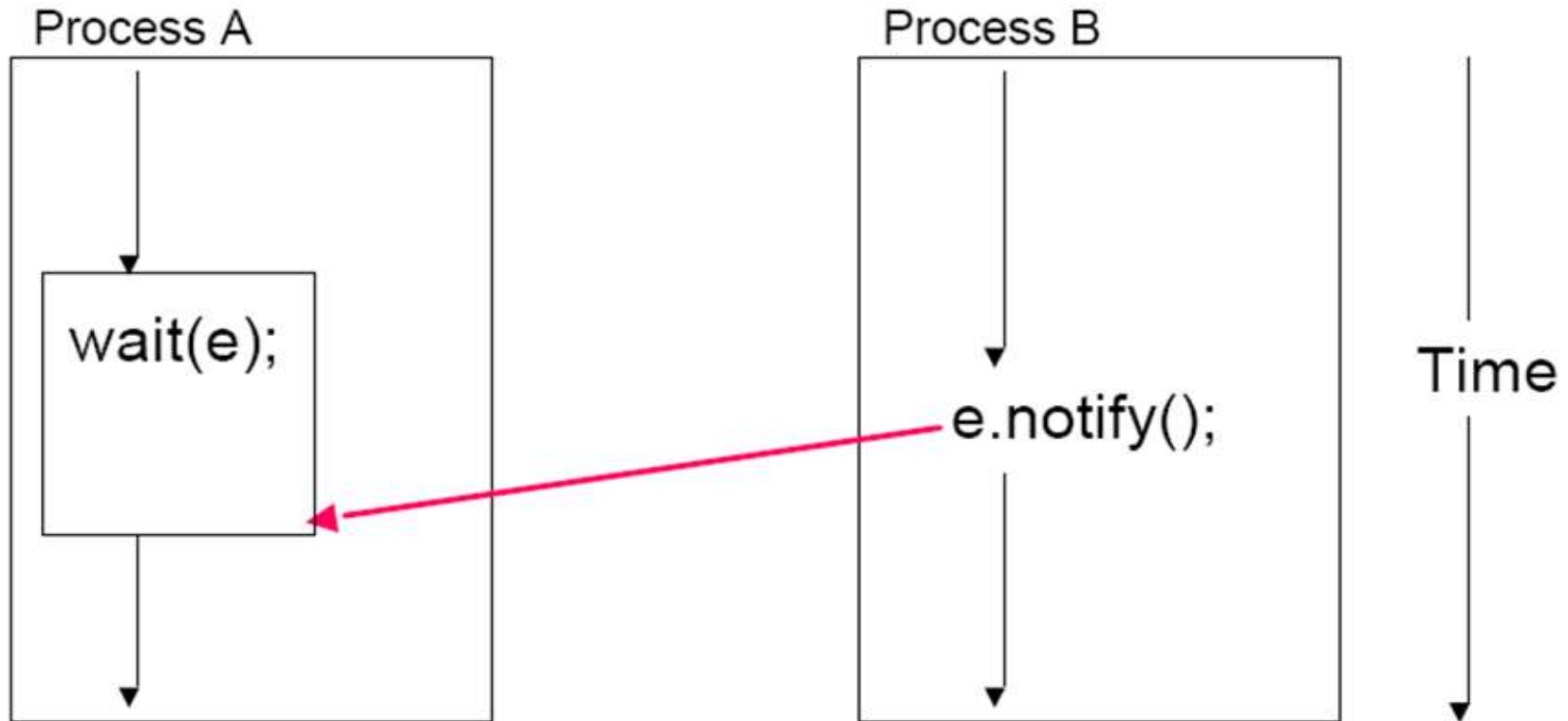
1- Ecrire le code SystemC correspondant à un additionneur 8 bits comme présenté ci-dessous.

L'additionneur sur 8 bits peut être modélisé par un module ayant deux entrées (entières codés sur 8 bits) et deux sorties : une sortie entière codée sur 8 bits donnant le résultat et un bit (booléen) supplémentaire nommé overflow indiquant s'il y a eu dépassement de la capacité



2. Ecrire le code en SystemC du module nommé « Tester » qui envoie les données à additionner, puis récupère le résultat et vérifie le bon fonctionnement.

Synchronisation



La classe: sc_event

- SystemC utilise la classe sc_event pour modéliser un événement.

sc_event e1, e2;

- Un événement n'a pas de valeur ni de durée.
- 2 actions sont possibles avec un événement:
 - L'attendre: **wait (e1);**
 - Le déclencher: **e2.notify()** ; *déclencher au prochain changement sur la liste de sensibilité*

e2.notify(10, SC_NS) ; *déclencher après 10 ns*

e2.notify(0) ; *déclencher au delta cycle suivant*

e2.cancel(0) ; *Arrêter l'événement e2*

Exercice 3:

Donner le résultat d'exécution du code suivant:

```
1  #include <systemc.h>
2
3  SC_MODULE (events) {
4      sc_in<bool> clock;
5
6      sc_event  e1;
7      sc_event  e2;
8
9      void do_test1() {
10         while (true) {
11             // Wait for posedge of clock
12             wait();
13             cout << "@" << sc_time_stamp() << " Starting test"<<endl;
14             // Wait for posedge of clock
15             wait();
16             cout << "@" << sc_time_stamp() << " Triggering e1"<<endl;
17             // Trigger event e1
18             e1.notify(5, SC_NS);
19             // Wait for posedge of clock
20             wait();
21             // Wait for event e2
22             wait(e2);
23             cout << "@" << sc_time_stamp() << " Got Trigger e2"<<endl;
24             // Wait for posedge of clock
25             wait();
26             cout<<"Terminating Simulation"<<endl;
27             sc_stop(); // sc_stop triggers end of simulation
28         }
29     }
30 }
```

```

31 void do_test2() {
32     while (true) {
33         // Wait for event e2
34         wait(e1);
35         cout << "@" << sc_time_stamp() << " Got Trigger e1"<<endl;
36         // Wait for 3 posedge of clock
37         wait(3);
38         cout << "@" << sc_time_stamp() << " Triggering e2"<<endl;
39         // Trigger event e2
40         e2.notify();
41     }
42 }
43
44 SC_CTOR(events) {
45     SC_CTHREAD(do_test1, clock.pos());
46     SC_CTHREAD(do_test2, clock.pos());
47 }
48 };
49
50 int sc_main (int argc, char* argv[]) {
51     sc_clock clock ("my_clock", 1, 0.5);
52
53     events object("events");
54     object.clock (clock.signal());
55
56     sc_start(0); // First time called will init scheduler
57     sc_start(); // Run the simulation till sc_stop is encountered
58     return 0; // Terminate simulation
59 }

```


Liste de sensibilité dynamique

SC_METHOD

```
SC_MODULE(M)
```

```
{
```

```
    sc_signal<int> sig;
```

```
    sc_event e1, e2;
```

```
    SC_CTOR(M)
```

```
    {
```

```
        SC_METHOD(entry_method);
```

```
        sensitive << sig;
```

```
    }
```

```
void entry_method() // Run first at initialization.
```

```
{
```

```
    if (sig == 0)
```

```
        next_trigger(e1 | e2); // Trigger on event e1 or event e2 next time
```

```
    else if (sig == 1)
```

```
        next_trigger(1, SC_MS); // Time-out after 1 millisecond.
```

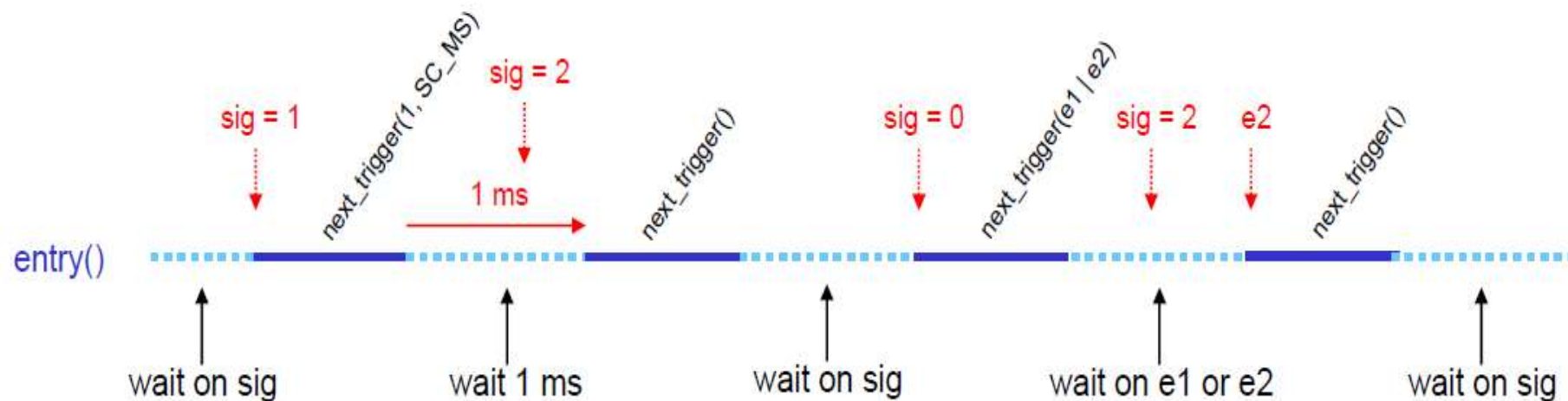
```
    else
```

```
        next_trigger(); // Trigger on signal sig next time.
```

```
}
```

```
...
```

```
};
```



SC_CTHREAD : clocked process



- Un **SC_CTHREAD** n'est sensible qu'à l'horloge
- Un **SC_CTHREAD** utilise la méthode ***wait_until()*** qui suspend l'exécution du process jusqu'à ce qu'une "*expr*" *booléenne* est vraie .

sc_signal<bool> reset

wait_until(reset.delayed()== true);

- Quand la condition est vraie, le contrôle du processus est transféré du point d'exécution courant du process ou l'occurrence de la condition est prise en question.

SC_CTHREAD : clocked process

- Un **SC_CTHREAD** n'est sensible qu'à l'horloge
- Un **SC_CTHREAD** utilise la méthode ***wait_until()*** qui suspend l'exécution du process jusqu'à ce qu'une "*expr*" *booléenne* est vraie .

Sc_signal<bool> reset

wait_until(reset.delayed()== true);

- Quand la condition est vraie, le contrôle du processus est transféré du point d'exécution courant du process ou l'occurrence de la condition est prise en question.
- Les expressions . **delayed()** sont testés à chaque front actif du process
- Le type de données . **delayed()** doit être "bool".

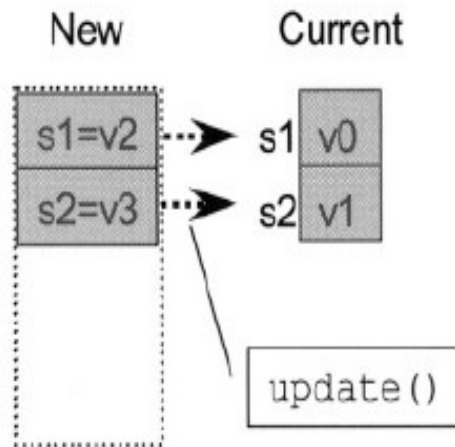
Example: SC_CTHREAD

```
// datagen.h
#include "systemc.h"
SC_MODULE(data_gen) {
    sc_in_clk clk;
    sc_inout<int> data;
    sc_in<bool> reset;
    void gen_data();
    SC_CTOR(data_gen){
        SC_CTHREAD(gen_data, clk.pos());
    }
};
```

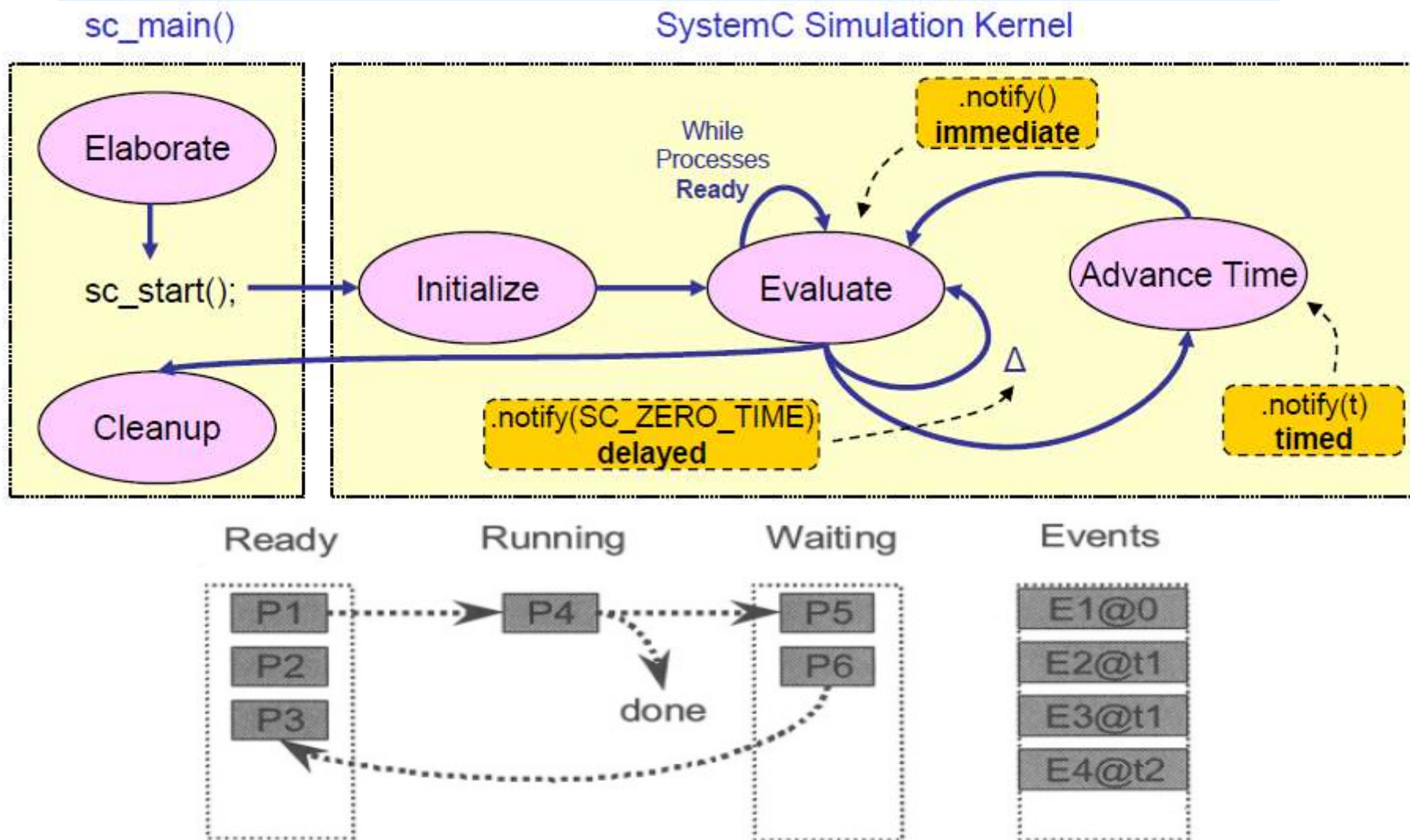
```
// datagen.cc
#include "datagen.h"
void data_gen::gen_data() {
    wait_until (reset.delayed() == true);
    if (reset == true) {
        data = 0;
    }
    while (true) {
        data = data + 1;
        wait();
        data = data + 2;
        wait();
        data = data + 4;
        wait();
    }
}
```

SC_SIGNAL

- Les signaux (sc_signal) utilisent la phase « update » pour la **synchronisation**.
- Pour se synchroniser chaque signal a 2 endroits de stockage: la valeur courante « current » et la valeur nouvelle « new ».
- Quand un process écrit une nouvelle valeur dans un « signal », cette valeur est stockée dans pile « new » (c'est la phase evaluate)
- A près un delta cycle, la valeur du signal est copiée de la pile new à la pile current.



Noyau de simulation SystemC



Le cycle **evaluate-update** est connu sous le nom delta cycle

Phases d'écriture sur un signal



- **Phase d'évaluation**: Quand un process écrit une nouvelle valeur dans un « signal », cette valeur est stockée dans pile « new »
- **Phase update**: la suspension de la phase évaluation par une instruction « wait .. » permet la mise à jour du signal càd copier la valeur du signal de la pile new à la pile current.
- **Phase Advance time**: Assure l'avancement temporel afin de retourner à la phase evaluation sinon de sortir complètement de l'exécution.

Example

```
int count, a;
sc_signal<int> count_sig;
// 1er delta cycle
count_sig.write(10);
count=11;
a=3;
cout<<sc_time_stamp()<<" count_sig ="<<count_sig<<endl;
wait(SC_ZERO_TIME);
// 2ème delta cycle
cout<<sc_time_stamp()<<" count_sig ="<<count_sig<<endl;
count_sig.write(20);
count=count_sig.read();
cout<<sc_time_stamp()<<" count"<<count<<endl;
Wait(10, SC_NS);
// 3ème delta cycle
a= count_sig.read();
cout<<sc_time_stamp()<<" a="<<a<<endl;
```

Exercice 4

Donner le résultat de l'exécution du code SystemC suivant :

```
// main.cpp
#include "exercice_signal.h"

int sc_main(int argc, char* argv[])
{
    sc_clock clk1("clk1", 10, SC_NS);
    exercice_signal mon_exercice("mon_exercice1");
    mon_exercice.clk(clk1);
    sc_start(150, SC_NS);
    return 0;
}
```

```
//exercice_signal.h
#include <systemc.h>
SC_MODULE(exercice_signal)
{
    sc_in_clk clk;
    enum etat {RESTART, COUNT};
    sc_signal<int> sig;
    sc_signal<etat> etat1;
    int n;

    SC_CTOR(exercice_signal)
    {
        SC_THREAD(user)
            sensitive <<sig;
            SC_CTHREAD(counter, clk.pos());
    }
    void user(void);
    void counter(void);
};
```

Suite Exercice 4

```
//exercice_signal.cpp
#include "exercice_signal.h"
void exercice_signal::user(void)
{
    while (true)
    {
        cout << sc_time_stamp() << " user: sig="
        << sig << endl;
        etat1.write (RESTART);
        cout << sc_time_stamp() << " user: restart
        counting" << endl;
        wait (SC_ZERO_TIME);
        etat1.write (COUNT);
        wait(100, SC_NS);
    }
}
void exercice_signal::counter(void)
{
    while (true)
    {
        cout << sc_time_stamp() << " counter:
        receiving clock edge" << endl;
```

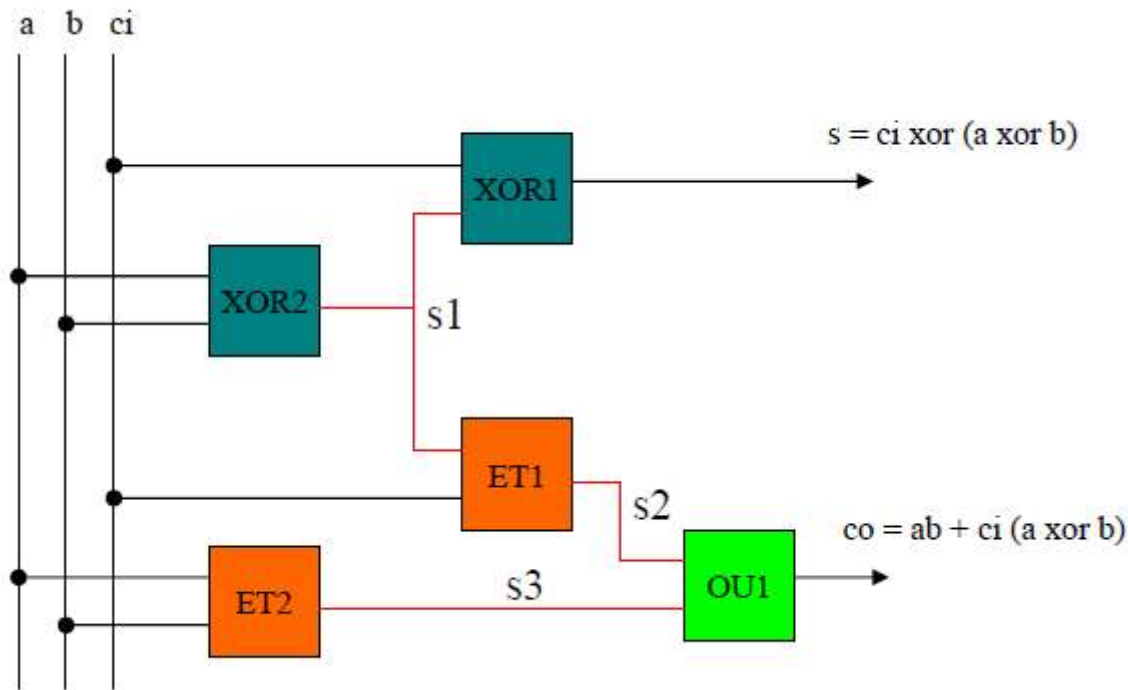
```
        if (etat1 == RESTART)
            n = 0;
        else
            n++;
        sig.write(n);
        cout << sc_time_stamp() << "
        counter: sig=" << sig << endl;
        wait ();
    }
}
```


Exemple utilisation de SC_SIGNAL:

Architecture structurelle

Additionneur à 1 bit:

- Ce montage contient 5 portes: 2 XOR, 2 ET, 1 OU
- Il contient aussi 3 signaux; S1, S2 et S3



Description porte ET à 2 entrées

```
//et.h
SC_MODULE(et) {
    sc_in<bool> a, b;
    sc_out<bool> s;
    void proc_and() {
        s.write(a.read() & b.read()); //s = a & b;
    }
    SC_CTOR(et) {
        SC_METHOD(proc_and);
        sensitive<< a << b;
    }
};
```

Description porte 0U à 2 entrées

```
SC_MODULE(ou) {  
    sc_in<bool> e1, e2;  
    sc_out<bool> s;  
    void proc_ou() {  
        s.write(e1.read() | e2.read()) //s = e1 | e2;  
    }  
    SC_CTOR(ou) {  
        SC_METHOD(proc_ou);  
        sensitive<<e1<<e2;  
    }  
};
```

Description porte XOR à 2 entrées

```
SC_MODULE(xor) {  
    sc_in<bool> e1, e2;  
    sc_out<bool> s;  
    void proc_xor() {  
        s.write(e1.read() ^ e2.read()); //s = e1^e2;  
    }  
    SC_CTOR(xor) {  
        SC_METHOD(proc_xor);  
        sensitive<<e1<<e2;  
    }  
};
```

Assemblage: Additionneur 1 bit

```
//add.h
#include "et.h"
#include "ou.h"
#include "xor.h"

SC_MODULE(add) {
    sc_in<bool> a, b, ci;
    sc_out<bool> s, co;

    sc_signal<bool> s1, s2, s3;

    et *ET1;
    et *ET2;

    ou *OU1;

    xor *XOR1;
    xor *XOR2;
```

Définit une variable qui est un pointeur sur la classe fille du module « et ».

Assemblage: Additionneur 1 bit

```
SC_CTOR(add) {  
    ET1 = new et("ET1");  
    ET2 = new et("ET2");  
  
    OU1 = new ou("OU1");  
  
    XOR1 = new xor("XOR1");  
    XOR2 = new xor("XOR2");  
  
    ET1->a(ci);    ET1->b(s1); ET1->s(s2);  
  
    ET2->a(a);    ET2->b(b);  ET2->s(s3);  
  
    XOR1->a(ci);  XOR1->b(s1); XOR1->s(s);  
  
    XOR2->a(a);   XOR2->b(b);  XOR2->s(s1);  
  
    OU1->a(s2);   OU1->b(s3);  OU1->s(co);  
  
    }  
};
```

Créer une instance de module fils



Code du programme principal

```
//main.cpp
#include <systemc.h>
#include "add.h"

int sc_main(int argc, char * argv[]) {
    add ADD("ADD");
    sc_signal<bool> a, b, ci, s, co;

    ADD.a(a);
    ADD.b(b);
    ADD.ci(ci);
    ADD.s(s);
    ADD.co(co);

    return (0);
}
```

Signal et trace

Le channel SC_SIGNAL permet tracer la forme du signal dans un fichier **.vcd**

```
sc_clock clock("clk", 1, SC_NS);  
sc_signal<int> shiftreg_in;  
sc_signal<int> shiftreg_out;  
...
```

filename



elaboration phase

```
sc_trace_file *tf = sc_create_vcd_trace_file("wave");  
  
sc_write_comment(tf, "Simulation of Shift Reg at Elaboration Time Resolution");  
  
sc_trace(tf, clock.signal(), "Clock");  
sc_trace(tf, shiftreg_in, "shiftreg_in");  
sc_trace(tf, shiftreg_out, "shiftreg_out");
```

end of simulation

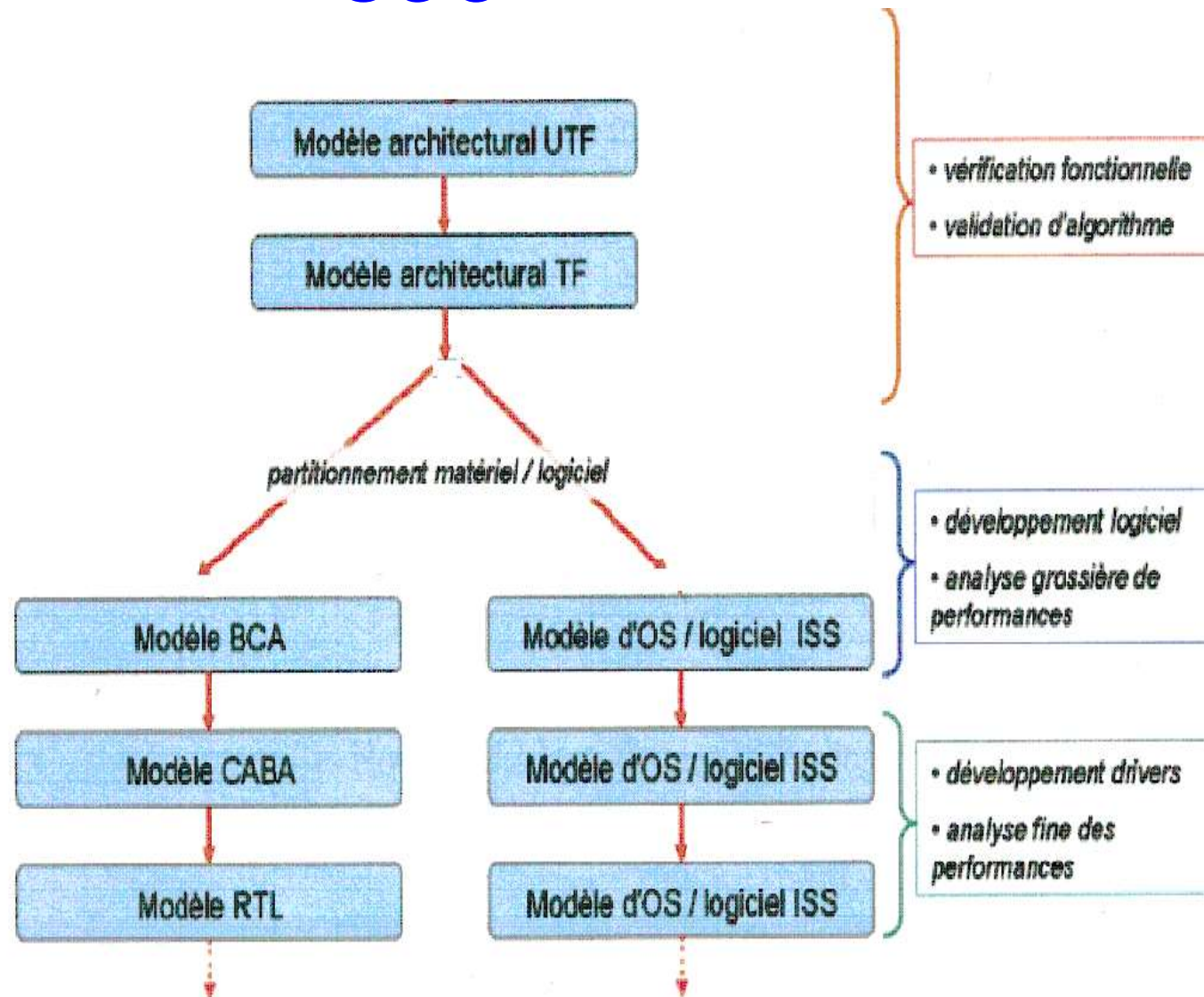
→ sc_close_vcd_trace_file(tf);



II. SystemC: Modélisation conjointe Hw/Sw des SoCs

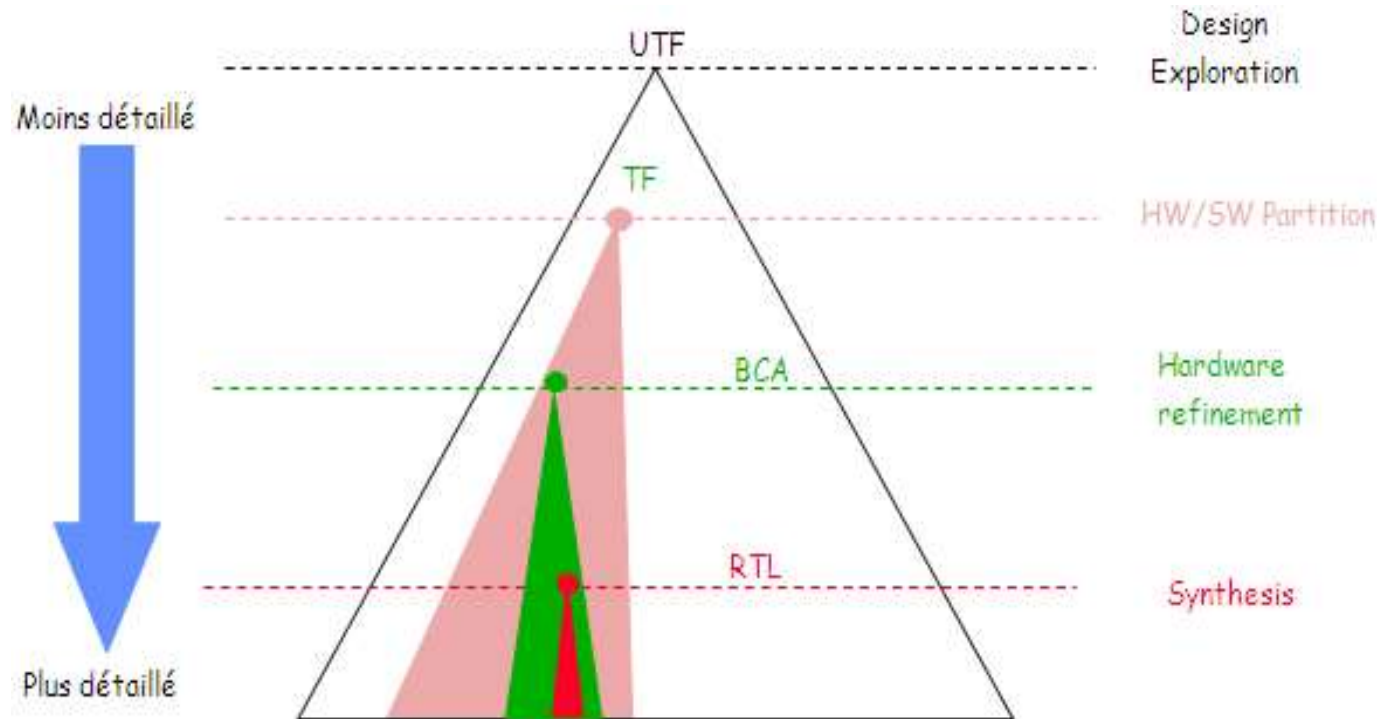
- Introduction à SystemC
- Elements du langage
- Notion de Concurrency
- ➔ • **Flot SLD & niveaux d'abstraction**

Flot SLD (System Level Design) d'un SOC



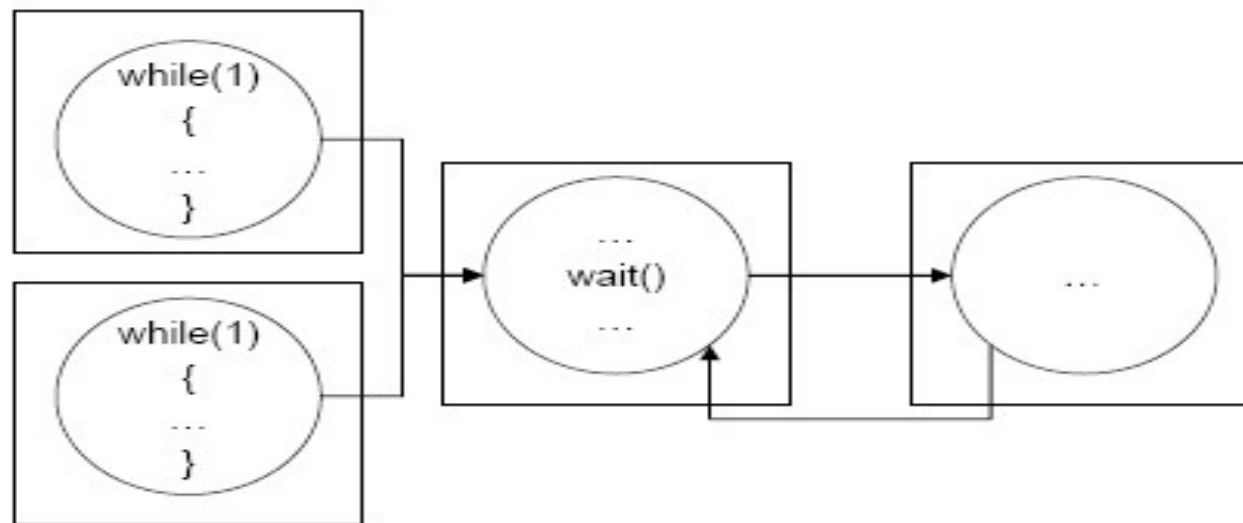
Flot de conception SOC

- Untimed Functional (UTF) Level
- Timed Functional (TF) Level
- Bus-Cycle Accurate (BCA) Level
- Register Transfer Level (RTL)



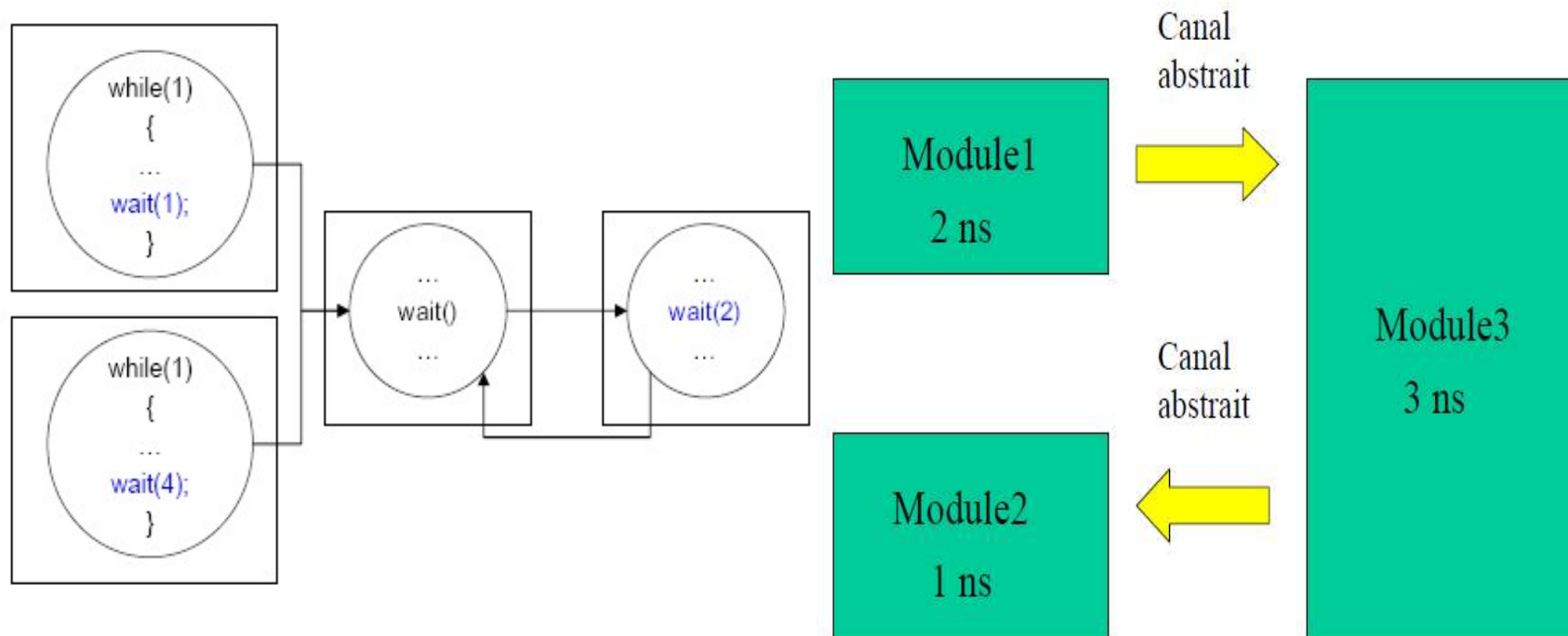
Modèle *UnTimed Fonctionnel Level* (UTF)

- Le modèle ne comporte aucune notion de durée d'exécution, mais seulement un ordre éventuel dans l'exécution des événements.
- Chaque événement s'exécute en un temps nul. Seul compte l'ordonnancement des événements



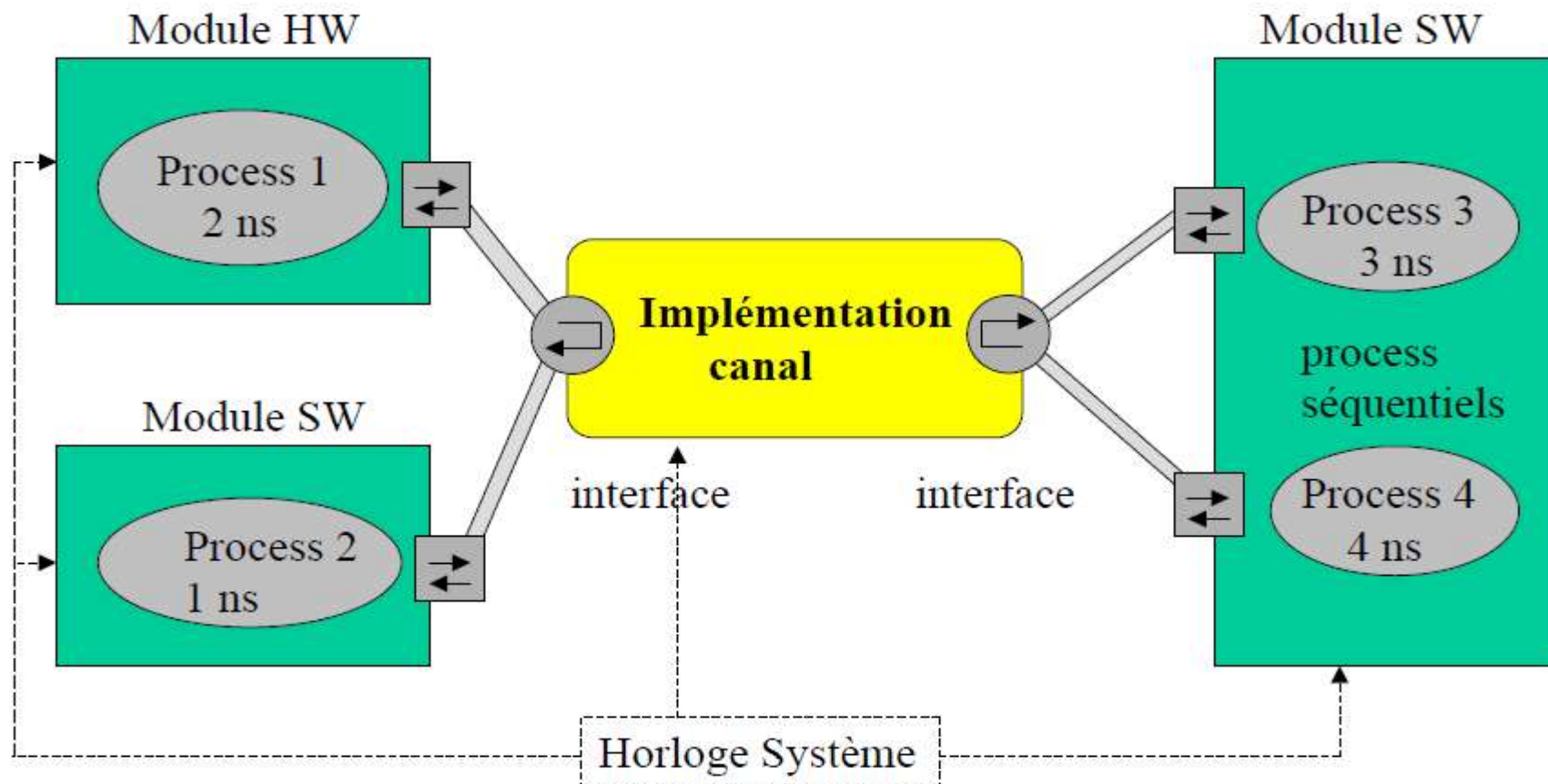
Modèle *Timed Functionnel Level* (TF)

- Le modèle comporte des notions de durée(temps d'exécution des processus, latence, temps de propagation...)
- Le temps est modélisé en utilisant **wait** (délai) mais les modules ne sont pas « clockés ».



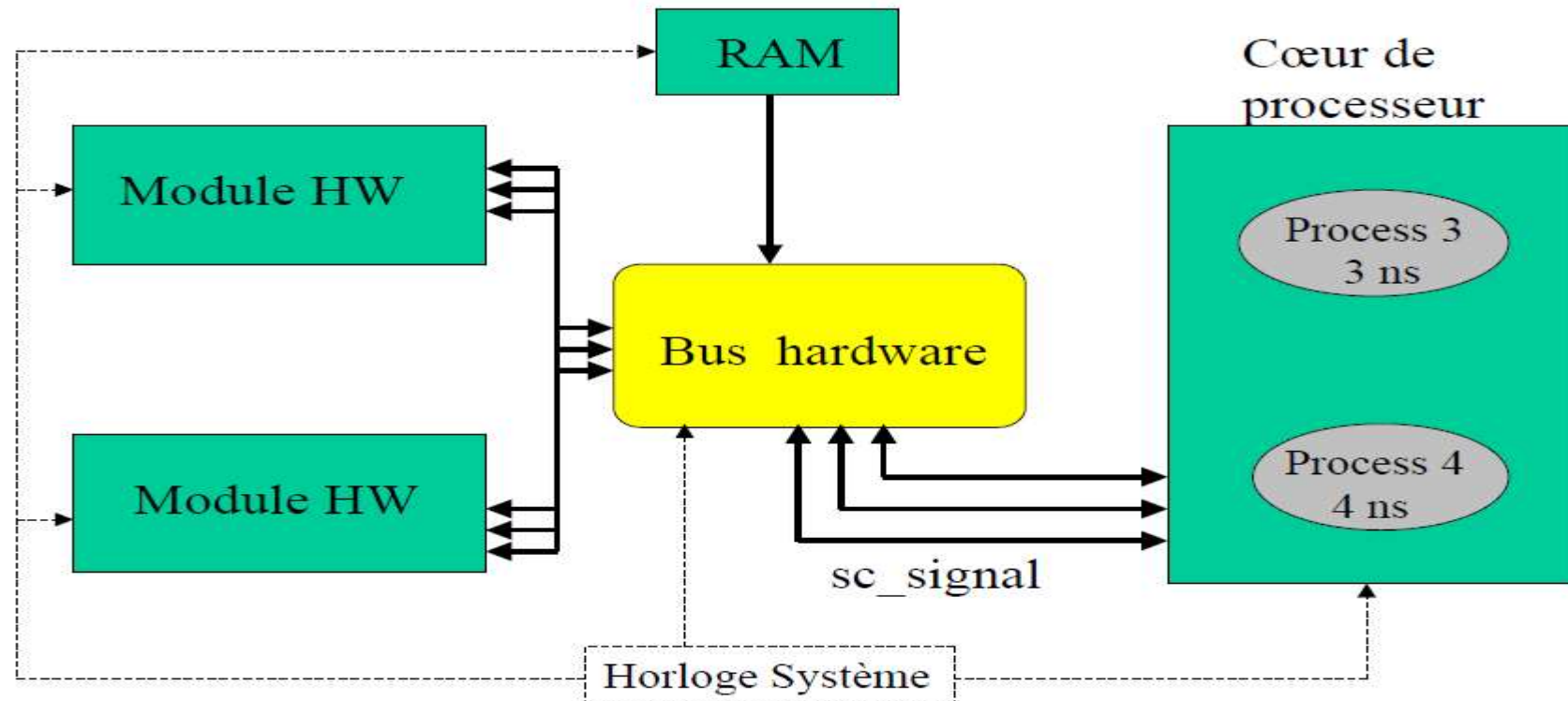
Modèle Bus Cycle Accurate Level (BCA)

- BCA s'applique à l'interface d'un modèle
- Les modules sont « clockés »
- Les transactions sur le bus sont modélisées précisément



Modèle Cycle Accurate Bit Accurate Level (CABA)

- CABA s'applique à l'interface et à la fonctionnalité d'un modèle.
- La modélisation est précise au bit (fil) près.
- Ce niveau d'abstraction est aussi appelé **Pin Accurate Level (PA)**



Modèle RTL

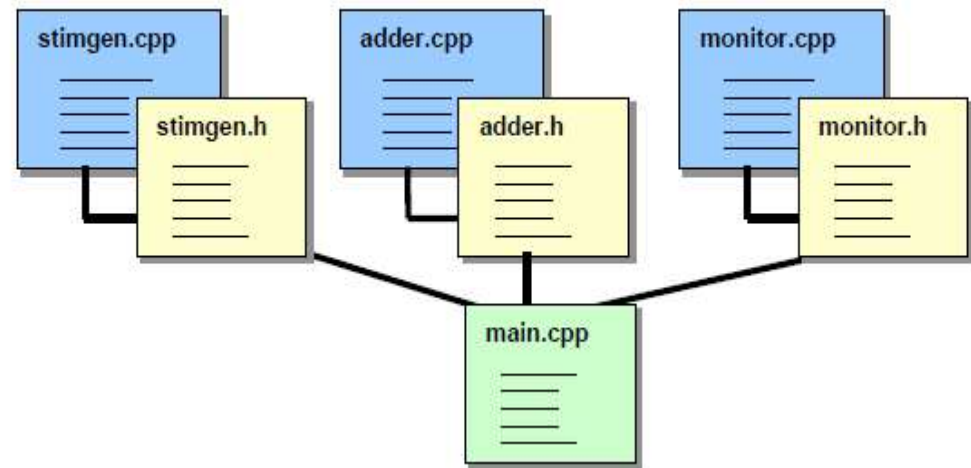
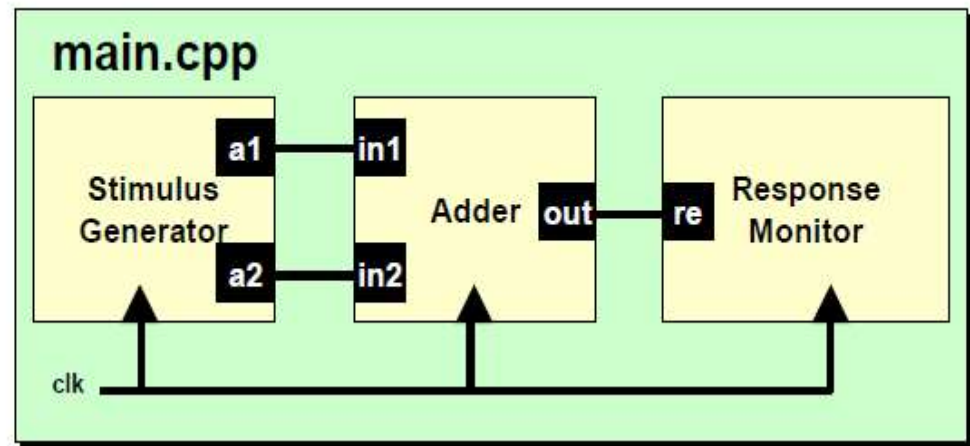
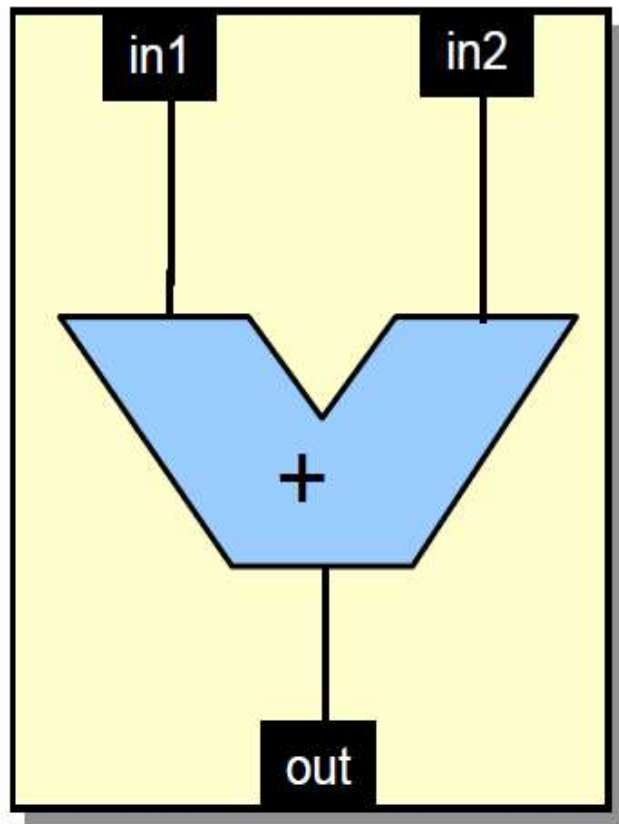


Le niveau transfert de registre (RTL) permet de définir le fonctionnement réel du système et permet également la synthèse de celui-ci (sous certaines conditions).

S'applique à l'interface et à la fonctionnalité d'un modèle matériel.

Chaque bit, chaque cycle, chaque registre du système est modélisé.

Exemple d'illustration



Niveau UTF

```
// header file adder.h
typedef int T_ADD;

SC_MODULE(adder) {

    // Input ports
    sc_in<T_ADD>    in1;
    sc_in<T_ADD>    in2;
    // Output port
    sc_out<T_ADD>    out;

    // Constructor
    SC_CTOR(adder) {
        SC_METHOD(add);
        sensitive << in1 << in2;
    }
    // Functionality of the process
    void add() ;
};
```

```
// Implementation file adder.cpp
#include "systemc.h"
#include "adder.h"

void adder::add()
{
    out.write(in1.read() + in2.read() );
}
```

Niveau TF

```
// header file adder.h
typedef int T_ADD;

SC_MODULE(adder) {

    // Input ports
    sc_in<T_ADD>    in1;
    sc_in<T_ADD>    in2;
    // Output port
    sc_out<T_ADD>    out;

    // Constructor
    SC_CTOR(adder) {
        SC_THREAD(add);
    }

    // Functionality of the process
    void add();
};
```

```
// Implementation file adder.cpp
#include "systemc.h"
#include "adder.h"

void adder::add()
{
    while (true) {
        out.write(in1.read() + in2.read());
        // assign a run-time to process
        wait(10, SC_NS);
    }
}
```

Niveau BCA

```
// header file adder.h

typedef sc_int<8> T_ADD;

SC_MODULE(adder) {
    // Clock introduced
    sc_in_clk      clk;
    // Input ports
    sc_in<T_ADD>    in1;
    sc_in<T_ADD>    in2;
    // Output port
    sc_out<T_ADD>   out;

    // Constructor
    SC_CTOR(adder) {
        SC_CTHREAD(adder, clk.pos());
    }

    // Functionality of the process
    void add();
};
```

```
// Implementation file adder.cc
#include "systemc.h"
#include "adder.h"

void adder::add()
{
    // initialization
    T_ADD __in1 = 0;
    T_ADD __in2 = 0;
    T_ADD __out = 0;

    out.write(__out);
    wait();

    // infinite loop
    while(1) {
        __in1 = in1.read();
        __in2 = in2.read();
        __out = __in1 + __in2;
        out.write(__out);
        wait();
    }
}
```

Niveau CABA

```
// header file adder.h

SC_MODULE(adder) {
    // Clock introduced
    sc_in_clk      clk;
    // Input ports
    sc_in<sc_lv<8>> in1;
    sc_in<sc_lv<8>> in2;
    // Output port
    sc_out<sc_lv<8>> out;

    // internal signal
    sc_signal<int>    sum;

    int __in1, __in2;
    SC_CTOR(adder) {
        SC_METHOD(add);
        sensitive_pos << clk;
    }

    // Functionality of the process
    void add();
};
```

```
// Implementation file adder.cc
#include "systemc.h"
#include "adder.h"

void adder::add()
{
    __in1 = in1.read().to_int();
    __in2 = in2.read().to_int();

    sum.write( __in1 + __in2 );

    out.write( static_cast<sc_lv<8>>(sum.read()));
}
```

Niveau RTL

```
// header file adder.h

SC_MODULE(adder) {
    // Clock introduced
    sc_in_clk      clk;
    // Input ports
    sc_in<sc_lv<8>> in1;
    sc_in<sc_lv<8>> in2;
    // Output port
    sc_out<sc_lv<8>> out;

    // internal signal
    sc_signal<int>    sum;

    int __in1, __in2;
    SC_CTOR(adder) {
        SC_METHOD(add);
        sensitive << in1 << in2;
        SC_METHOD(reg);
        sensitive_pos << clk;
    }

    // Functionality of the process
    void add();
    void reg();
};
```

```
// Implementation file adder.cc
#include "systemc.h"
#include "adder.h"

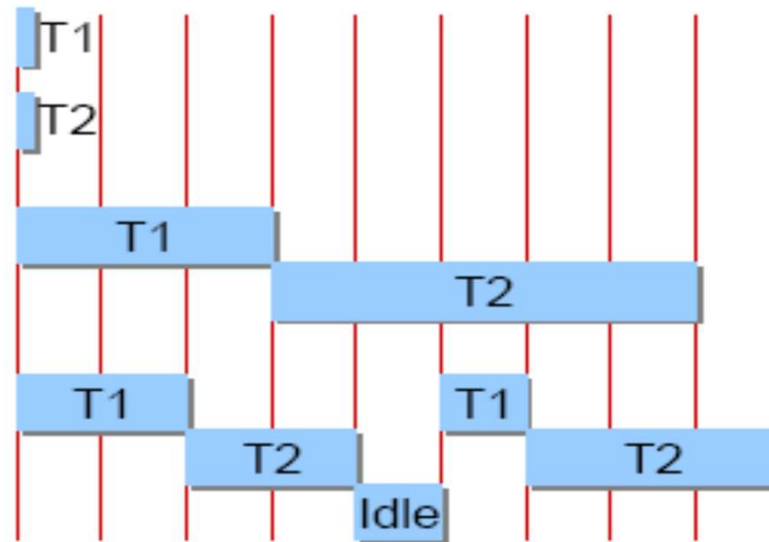
void adder::add()
{
    __in1 = in1.read().to_int();
    __in2 = in2.read().to_int();

    sum.write( __in1 + __in2 );
}

void adder::reg()
{
    out.write( static_cast<sc_lv<8>>(sum.read()));
}
```

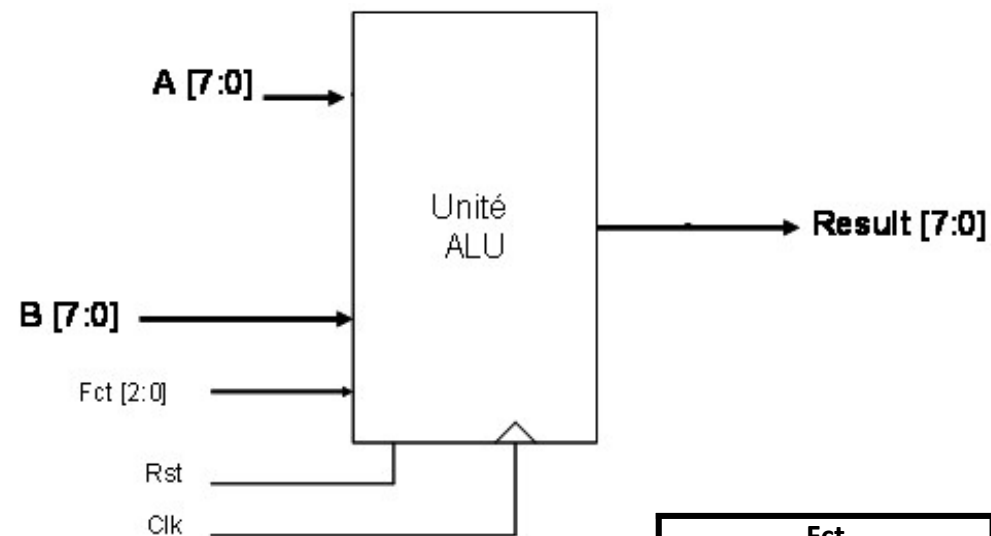

Exercice: Niveaux d'abstraction

Soit trois niveaux d'abstraction sur la figure suivante où pour chaque niveau est donnée la trace de 2 SC_THREAD T1 et T2. Donnez le nom associé à chaque niveau d'abstraction. Considérez une période d'horloge entre chaque ligne verticale.



Exercice 5

Modéliser en SystemC l'unité ALU au niveau **RTL**.



Fct	Opération
000	Ne rien faire
001	Addition
010	Soustraction
011	Multiplication
100	Division
101	ET
110	OU
111	NON (A)