Introduction à la modélisation en SystemC

par J. Bhasker
(Traduction de F. Pêcheux, Septembre 2004)

1. Avertissement

Ce document est la traduction du chapitre 2 du livre de J. Bhasker, « A SystemC Primer », aux Editions Star Galaxy Publishing (ISBN 0-9650391-8-8). C'est une introduction à la modélisation en SystemC. Elle présente la structure d'un module SystemC, explique comment déclarer les ports avec leurs types, et comment décrire le comportement d'un module au travers d'un exemple complet d'additionneur 2 bits.

2. Les bases

Un module est l'entité de base pour décrire un système en SystemC. Un module peut avoir un nombre quelconque d'entrées, de sorties, ou d'entrées/sorties. Un module peut contenir un nombre quelconque de processus. Un processus sert à décrire la fonctionnalité d'un système, et permet d'émuler le comportement concurrent de ses composants. Chaque processus est dit sensible à un ensemble de signaux et de ports, et s'exécute dès qu'il y a une modification sur un des signaux ou ports de la liste de sensibilité. Les signaux sont utilisés pour les communications inter-processus. De plus, l'affectation d'une valeur à un signal (ou à un port) se fait toujours avec un délai « delta » ; un délai « delta » est un délai infiniment court servant à modéliser les relations de cause à effet de la logique électronique. Un module permet aussi de décrire la hiérarchie, ce qui permet d'instancier un module dans un autre, comme indiqué à la figure 2-1.

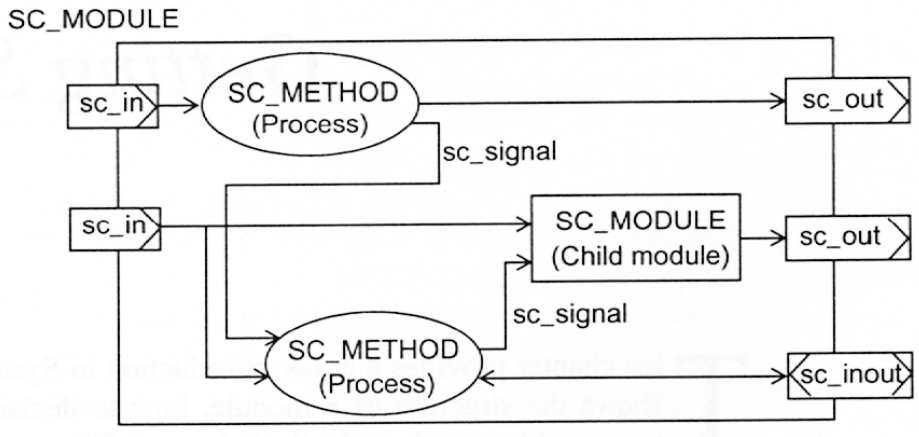

Figure 2-1 : Un module, avec ses ports, processus et signaux.

La figure 2-2 montre le circuit d'un demi-additionneur classique, qui va servir d'exemple tout au long du chapitre.

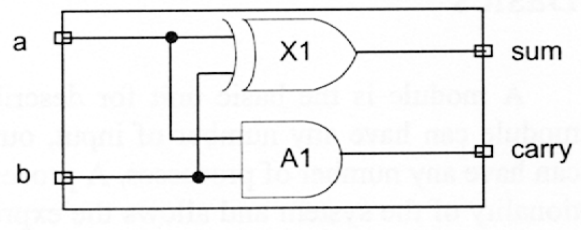

Figure 2-2 : le demi-additionneur.

Voici son modèle SystemC correspondant :

```
1. // File : half_adder.h
2. #include "systemc.h"
3.
4. SC_MODULE(half_adder)
5. {
6.     sc_in<bool> a,b;
7.     sc_out<bool> sum,carry;
8.
9.     void prc_half_adder();
10.
11.    SC_CTOR(half_adder)
12.    {
13.          SC_METHOD(prc_half_adder);
14.          sensitive << a << b;
15.    }
16.};
```

Listing 2-1 : le fichier **`half_adder.h`**.

```
1. // File : half_adder.cpp
2. #include "half_adder.h"
3.
4. void half_adder::prc_half_adder()
5. {
6.     sum = a ^ b;
7.     carry = a & b;
8. }
```

Listing 2-2 : le fichier **`half_adder.cpp`**.

Le circuit demi-additionneur est en fait contenu dans deux fichiers, **`half_adder.h`** et **`half_adder.cpp`**. Le fichier **`half_adder.h`** (fichier include C++) contient la description du module et la déclaration des processus qui le composent, tandis que le fichier **`half_adder.cpp`** (fichier source C++) contient la définition de ces processus. L'écriture d'un module SystemC respecte donc le style de programmation C++, la déclaration et la définition dans deux fichiers séparés.

La ligne 2 de **half_adder.h** est une directive d'inclusion du fichier **systemc.h**. Cette directive doit apparaître directement ou indirectement dans tout module SystemC. Le fichier **systemc.h** contient les définitions de toutes les bibliothèques de classes SystemC.

Le mot-clé **SC_MODULE** commence la déclaration d'un nouveau module SystemC. Le nom du module est ici **half_adder**. Le module dispose de deux ports d'entrée : **a** et **b**. Les ports sont déclarés comme étant de type **bool** (type standard du C++). Le module a deux ports de sortie : **sum** et **carry**. Leur type est aussi **bool**.

Le bloc **SC_CTOR** à la ligne 11 déclare les processus, et types de processus, utilisés pour décrire le comportement du module. Dans le module **half_adder**, le bloc **SC_CTOR** declare un processus de type **SC_METHOD**. Le nom associé au bloc **SC_CTOR** doit être identique à celui du module. Un processus de type **SC_METHOD** est sensible à un ensemble fini de signaux et de ports et ne peut être temporairement arrêté à l'aide d'une instruction **wait** ; les instructions **wait** , qui permettent d'arrêter un processus en plain milieu de son exécution, ne sont pas autorisées à l'intérieur d'un processus **SC_METHOD**. L'autre type de processus est le processus **SC_THREAD**, abordé plus loin.

Le nom d'un processus est défini dans la déclaration du processus **SC_METHOD**. Dans notre cas, le nom du processus est **prc_half_adder**, déclaré à la ligne 9. Le processus, qui du point de vue C++ est une fonction membre, doit retourner **void**, et ne doit pas prendre d'arguments. L'instruction sensitive qui apparaît ligne 14 sert à spécifier les signaux et ports auxquels le processus **SC_METHOD** est sensible. Ici, le processus **prc_half_adder** est sensible aux deux ports d'entrée **a** et **b**. Cela signifie que toute modification de valeur sur le port **a** ou sur le port **b** entraîne l'exécution complète de la fonction membre **prc_half_adder**. Une fois que cette fonction a été exécutée, le processus **prc_half_adder** se met en attente d'autres événements sur **a** ou **b**. Notez que la déclaration d'un processus **SC_METHOD** se termine par un point-virgule.

Le second fichier, **half_adder.cpp**, contient la définition du processus. Les deux opérations d'affectation des lignes 6 et 7 calculent les valeurs des ports de sortie **sum** et **carry**. L'affectation d'une valeur à un port se fait toujours après un délai delta. Ceci est aussi vrai pour l'affectation à un signal. Autrement dit, si le port **a** change de valeur à t=5ns, **sum** et **carry** prennent leurs nouvelles valeurs à t=5+delta ns.

3. Un autre exemple

Voyons un autre exemple, celui d'un décodeur 2 vers 4, montré à la figure 2-3.

Le module décodeur a pour nom **decoder2by4** et est défini dans le fichier **decoder2by4.h**. Il dipose de deux ports d'entrée, et d'un port de sortie. Le port d'entrée **select** est de type **sc_uint<2>**. Cela signifie que **select** correspond à un vecteur entier de 2 bits.

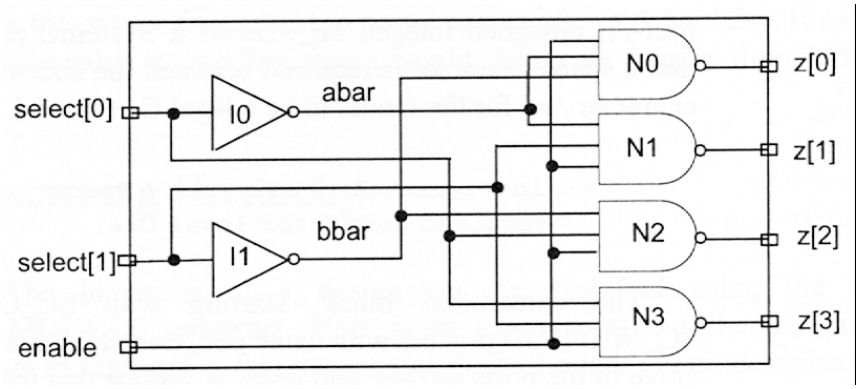

Figure 2-3 : Un décodeur 2 vers 4.

De même, le port de sortie est de type **`sc_uint<4>`**, soit donc un vecteur entier sur 4 bits. **`sc_uint`** est un type SystemC prédéfini. Notez qu'un caractère espace doit être inséré entre la taille **`<2>`** et le caractère **`>`** de fin de type, pour que le modèle soit compilable sans erreur.

```
1. // File : decoder2by4.h
2. #include "systemc.h"
3.
4. SC_MODULE(decoder2by4)
5. {
6.     sc_in<bool> enable;
7.     sc_in<sc_uint<2> > select;
8.     sc_out<sc_uint<4> > z;
9.
10.    void prc_decoder2by4();
11.
12.    SC_CTOR(decoder2by4)
13.    {
14.          SC_METHOD(prc_decoder2by4);
15.          sensitive (enable,select);
16.    }
17.};
```

Listing 2-3 : le fichier **`decoder2by4.h`**.

```
1. // File : decoder2by4.cpp
2. #include "decoder2by4.h"
3.
4. void decoder2by4::prc_decoder2by4()
5. {
6.     if (enable)
7.     {
8.           switch (select.read())
9.           {
10.                case 0: z=0xE ; break;
11.                case 1: z=0xD ; break;
12.                case 2: z=0xB ; break;
13.                case 3: z=0x7 ; break;
14.          }
15.    }
16.    else
17.          z=0xF;
18.}
```

Listing 2-4 : le fichier **`decoder2by4.cpp`**.

Le bloc constructeur, commençant par **SC_CTOR**, déclare un processus **SC_METHOD** nommé **prc_decoder2by4**. Ce processus est sensible aux ports d'entrée **select** et **enable**. Notez que la liste de sensibilité pour ce processus **SC_METHOD** est décrite ici avec un style différent, sous forme d'un appel de fonction.

```
sensitive(enable,select) ;
```

Dans le module **half_adder** décrit dans la section précédente, nous avons utilisé le style "notation par flux". Si on souhaitait faire de même ici, il faudrait écrire :

```
sensitive << select << enable ;
```

Quel que soit le style de notation utilisé, il est possible d'ajouter chaque élément à la liste de sensibilité de manière indépendante :

```
// notation par flux
sensitive << select ;
sensitive << enable ;

// notation par appel de fonction

sensitive(select) ;
sensitive(enable) ;
```

Dans la définition de la fonction, contenue dans **decoder2by4.cpp**, le comportement du décodeur est défini à l'aide des instructions **if** et **switch** du C++. La méthode **read()** est nécessaire pour lire les valeurs du port **select** (sinon vous obtiendrez une erreur de compilation). Le compilateur C++ éprouve en effet quelques difficultés dès qu'il s'agit de convertir un **sc_in<sc_uint<2> >** en un entier utilisé par l'instruction **switch**. Encore une fois, si le port **select** change de valeur à t=7ns, **z** prend sa nouvelle valeur à t=7+delta ns.

4. Description de la hiérarchie

La hiérarchie d'un système peut être aussi obtenue par le biais d'un SC_MODULE. Voici, à la figure 2-4, un exemple d'additionneur complet, qui instancie deux fois le demi-additionneur vu précédemment.

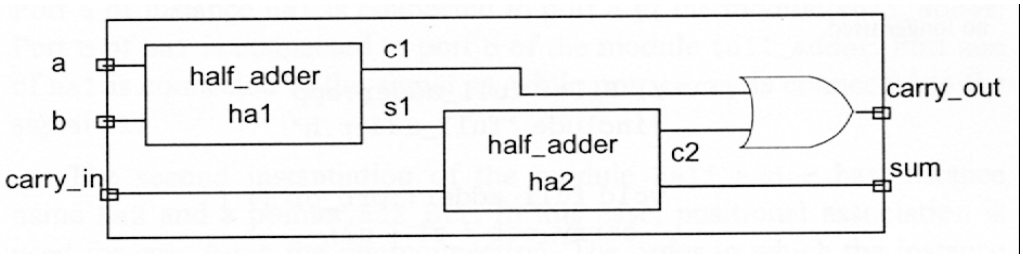

Figure 2-4 : Un additionneur complet.

Le module **full_adder** dispose de trois ports d'entrée et de deux ports de sortie, tous de type **bool**. Notez aussi dans le listing la présence d'une directive d'inclusion du fichier

half_adder.h, qui est nécessaire pour compiler correctement le **full_adder**. Il n'est pas utile ici d'inclure **systemc.h**, puisque c'est déjà fait dans **half_adder.h**.

```
1. // File : full_adder.h
2. #include "half_adder.h"
3.
4. SC_MODULE(full_adder)
5. {
6.     sc_in<bool> a,b,carry_in;
7.     sc_out<bool> sum,carry_out;
8.
9.     sc_signal<bool> c1,s1,c2;
10.    void prc_or();
11.
12.    half_adder *ha1_ptr,*ha2_ptr;
13.
14.    SC_CTOR(full_adder)
15.    {
16.          ha1_ptr=new half_adder("ha1");
17.          // Named association:
18.          ha1_ptr->a(a);
19.          ha1_ptr->b(b);
20.          ha1_ptr->sum(s1);
21.          ha1_ptr->carry(c1);
22.
23.          ha2_ptr=new half_adder("ha2");
24.          // Named association:
25.          ha2_ptr->a(s1);
26.          ha2_ptr->b(carry_in);
27.          ha2_ptr->sum(sum);
28.          ha2_ptr->carry(c2);
29.          // Positional association:
30.          //(*ha2_ptr)(s1,carry_in,sum,c2);
31.
32.          SC_METHOD(prc_or);
33.          sensitive << c1 << c2;
34.    }
35.
36.    // A destructor
37.    ~full_adder()
38.    {
39.          delete ha1_ptr;
40.          delete ha2_ptr;
41.    }
42.};
```

Listing 2-5 : le fichier **full_adder.h**.

```
1. // File : full_adder.cpp
2. #include "full_adder.h"
3.
4. void full_adder::prc_or()
5. {
6.     carry_out = c1 | c2;
7. }
```

Listing 2-6 : le fichier **full_adder.cpp**.

La ligne :

```
sc_signal<bool> c1,s1,c2;
```

déclare des signaux locaux dans le module **full_adder**, de type **bool**. Les signaux sont utilisés comme médium de communication entre les processus et les instances de modules. La ligne :

```
half_adder *ha1_ptr,*ha2_ptr;
```

déclare deux pointeurs vers le module **half_adder**, un pour chaque instance.

Le constructeur **SC_CTOR** du module **full_adder** contient deux instanciations de modules et une déclaration de processus **SC_METHOD**. La première instanciation :

```
ha1_ptr = new half_adder("ha1");
```

crée une nouvelle instance de module **half_adder** avec pour nom **"ha1"**. Le pointeur retourné est affecté à la variable **ha1_ptr**, et est utilisé pour connecter les ports et signaux externes aux ports de l'instance. Cette interconnexion peut se faire de deux manières :

- l'association positionnelle,
- l'association nommée.

Pour l'instance **ha1**, nous utilisons l'association nommée. Le port **a** de l'instance **ha1** est connecté au port **a** du module **full_adder**. Le port **b** de **ha1** est connecté au port **b** du module **full_adder**. Le port **sum** de **ha1** est connecté au signal **s1**, tandis que le port **carry** est connecté au signal **c1**.

La deuxième instanciation du module **half_adder** a pour nom **ha2** et pour pointeur **ha2_ptr**. Dans ce second cas, nous utilisons l'association positionnelle. L'ordre dans lequel les ports sont connectés durant l'association est fondamental. Les connexions sont faites suivant l'ordre des ports déclarés dans le module instancié, et séparés par l'opérateur **<<**. Le port **a** de l'instance **ha2** est connecté au signal **s1**, le port **b** de **ha2** au port **carry_in** de **full_adder**, et ainsi de suite.

Le bloc **SC_CTOR** déclare aussi un processus **SC_METHOD** nommé **prc_or**, qui est sensible aux signaux **c1** et **c2**. Le comportement de ce processus est décrit dans **full_adder.cpp**. Ce processus réalise le ou logique entre les deux retenues intermédiaires et sert à générer la sortie **carry_out**.

Ce module dispose d'un destructeur. Le destructeur supprime de la mémoire ce qui a été créé avec l'opérateur **new** dans le constructeur **SC_CTOR**. Il est important de faire cela pour éviter les fuites mémoire. Pour les utilisateurs connaissant peu le C++, voici à quoi ressemble un destructeur :

```
~module_name()
{
      delete ptr1 ;
```

```
      delete ptr2 ;
      …
      // ptr1 et ptr2 sont des pointeurs correspondant à des zones allouées
dans le SC_CTOR
}
```

Notez que le destructeur n'est utile que lorsque de la mémoire est allouée dans un bloc **`SC_CTOR`**.

5. Vérification de la fonctionnalité

Une fois que l'on a écrit un modèle SystemC, comment tester sa fonctionnalité ? SystemC propose un environnement de simulation et un ensemble de fonctions pour faire cela. Nous allons maintenant aborder la génération d'une horloge interne, et la capture des traces de simulation.

Examinons à nouveau le module **`full_adder`**, et imaginons que nous souhaitons appliquer toutes les valeurs possibles d'entrée sur ce module, toutes les 5 ns. Chaque valeur d'entrée est un *pattern*, et les valeurs de sortie du module sont enregistrées à chaque fois qu'il y a une modification sur les entrées ou sorties de **`full_adder`**. Tout ce environnement de test est décrit dans les fichiers suivants : **`driver.h`**, **`driver.cpp`**, **`monitor.h`**, **`monitor.cpp`** et **`full_adder_main.cpp`**.

```
1. // File : driver.h
2. #include "systemc.h"
3.
4. SC_MODULE(driver)
5. {
6.     sc_out<bool> d_a,d_b,d_cin;
7.
8.     void prc_driver();
9.
10.    SC_CTOR(driver)
11.    {
12.          SC_THREAD(prc_driver);
13.    }
14.};
```

Listing 2-7 : le fichier **`driver.h`**.

```
1. // File : driver.cpp
2. #include "driver.h"
3.
4. void driver::prc_driver()
5. {
6.     sc_uint<3> pattern;
7.     pattern=0;
8.
9.     while (1)
10.    {
11.          d_a=pattern[0];
12.          d_b=pattern[1];
13.          d_cin=pattern[2];
14.          wait(5,SC_NS);
15.          pattern++;
```

```
16.   }
17.}
```

Listing 2-8 : le fichier **`driver.cpp`**.

```
1. // File : monitor.h
2. #include "systemc.h"
3.
4. SC_MODULE(monitor)
5. {
6.    sc_in<bool> m_a,m_b,m_cin,m_sum,m_cout;
7.
8.    void prc_monitor();
9.
10.   SC_CTOR(monitor)
11.   {
12.         SC_METHOD(prc_monitor);
13.         sensitive << m_a << m_b << m_cin << m_sum << m_cout;
14.   }
15.};
```

Listing 2-9 : le fichier **`monitor.h`**.

```
1. // File : monitor.cpp
2. #include "monitor.h"
3.
4. void monitor::prc_monitor()
5. {
6.    cout << "At time " << sc_time_stamp() << "::";
7.    cout << "(a, b, carry_in): ";
8.    cout << m_a << m_b << m_cin;
9.    cout << " (sum, carry_out): " << m_sum << m_cout << endl;
10.}
```

Listing 2-10 : le fichier **`monitor.cpp`**.

```
1. // File : full_adder_main.cpp
2. #include "driver.h"
3. #include "monitor.h"
4. #include "full_adder.h"
5.
6. int sc_main(int argc,char *argv[])
7. {
8.    sc_signal<bool> t_a, t_b, t_cin, t_sum, t_cout;
9.
10.   full_adder f1("FullAdderWithHalfAdder");
11.   // Connect using positional association:
12.   f1 << t_a << t_b << t_cin << t_sum << t_cout;
13.
14.   driver d1("GenerateWaveforms");
15.   // Connect using named association:
16.   d1.d_a(t_a);
17.   d1.d_b(t_b);
18.   d1.d_cin(t_cin);
19.
20.   monitor mo1("MonitorWaveforms");
21.   mo1 << t_a << t_b << t_cin << t_sum << t_cout;
22.
23.   sc_start(100,SC_NS);
```

```
24.
25.   return(0);
26.}
```

Listing 2-11 : le fichier **full_adder_main.cpp**.

Pour simuler un modèle SystemC, nous devons tout d'abord écrire une fonction **sc_main()** dans le fichier **full_adder_main.cpp**, le programme principal. Cette fonction prend deux arguments en paramètre : **argc**, le nombre d'arguments de la ligne de commande, et **argv**, un tableau contenant ces arguments.

Le module **full_adder** est testé en écrivant un module **driver** et un module **monitor** . Le module **driver** génère les patterns d'entrée, un toutes les 5 ns. Le module **monitor** affiche les valeurs de tous les ports de **full_adder**, dès que l'un d'entre eux change de valeur.

Examinons tout d'abord le module **driver**. Ce module dispose de trois ports de sortie, et d'aucun port d'entrée. Le bloc **SC_CTOR** de module déclare un processus de type **SC_THREAD**. Un tel processus peut être interrompu par des instructions **wait**. Une instruction **wait** peut attendre pendant un certain temps, ou attendre qu'une certaine condition sur des signaux ou des ports se réalise, ou toute autre combinaison. Le processus **SC_THREAD** définit une variable locale **pattern**, de type entier sur 3 bits. La boucle **while** permet les itérations et sert à affecter chaque *pattern* aux ports de sortie. A chaque tour de boucle, **pattern** est incrémenté. **pattern** est un exemple de variable, qui s'oppose à la notion de signal ou de port. Une variable est affectée à temps nul (pas de délai delta). En d'autres termes, une affectation pour une variable se produit instantanément.

SystemC autorise la sélection de bit dans une variable de type **sc_uint**, grâce à l'opérateur **[]**. Ainsi, **pattern[0]** représente le bit 0 de l'entier non signé. Les trois affectations ont pour effet d'affecter le bit 0 du pattern au port **d_a**, le bit 1 au port **d_b**, et le bit 2 au port **d_cin**. L'instruction **wait**, dernière instruction de la boucle, permet au processus de « s'endormir » pendant 5 ns. **SC_NS** est l'unité de temps et représente des nanosecondes.

Le module **monitor** n'a que des ports d'entrée. Il suit les évolutions de toutes les entrées et sorties de l'instance de module **full_adder**. Ce suivi est fait au moyen d'un processus **SC_METHOD** qui est appelé à chaque modification d'un de ses ports d'entrée. La méthode **prc_monitor** est appelée pour afficher toutes ces valeurs. La fonction prédéfinie **sc_time_stamp()** retourne la valeur courante de l'horloge de simulation.

C'est dans la fonction **sc_main()** que tous les composants **driver**, **full_adder** et **monitor** sont interconnectés. Notez qu'au début de ce fichier, trois directives d'inclusion permettent au fichier **full_adder_main.cpp** de connaître ces modules.

L'instruction

```
sc_start(100, SC_NS) ;
```

lance la simulation pendant 100 nanosecondes.

Le résultat produit est le suivant :

```
             SystemC 2.0.1 --- Sep 26 2004 19:02:05
        Copyright (c) 1996-2002 by all Contributors
                     ALL RIGHTS RESERVED
At time 0 s::(a, b, carry_in): 000 (sum, carry_out): 00
At time 5 ns::(a, b, carry_in): 100 (sum, carry_out): 00
At time 5 ns::(a, b, carry_in): 100 (sum, carry_out): 10
At time 10 ns::(a, b, carry_in): 010 (sum, carry_out): 10
At time 15 ns::(a, b, carry_in): 110 (sum, carry_out): 10
At time 15 ns::(a, b, carry_in): 110 (sum, carry_out): 01
At time 20 ns::(a, b, carry_in): 001 (sum, carry_out): 01
At time 20 ns::(a, b, carry_in): 001 (sum, carry_out): 11
At time 20 ns::(a, b, carry_in): 001 (sum, carry_out): 10
At time 25 ns::(a, b, carry_in): 101 (sum, carry_out): 10
At time 25 ns::(a, b, carry_in): 101 (sum, carry_out): 00
At time 25 ns::(a, b, carry_in): 101 (sum, carry_out): 01
At time 30 ns::(a, b, carry_in): 011 (sum, carry_out): 01
At time 35 ns::(a, b, carry_in): 111 (sum, carry_out): 01
At time 35 ns::(a, b, carry_in): 111 (sum, carry_out): 11
At time 40 ns::(a, b, carry_in): 000 (sum, carry_out): 11
At time 40 ns::(a, b, carry_in): 000 (sum, carry_out): 01
At time 40 ns::(a, b, carry_in): 000 (sum, carry_out): 00
At time 45 ns::(a, b, carry_in): 100 (sum, carry_out): 00
```

Notez qu'il y a plusieurs lignes avec la même valeur de temps. Ce qui est tout à fait normal puisque l'on appelle le **`monitor`** à chaque variation sur une entrée ou une sortie. Des telles variations peuvent survenir après un ou plusieurs délais delta pour un temps de simulation donné. Si vous souhaitez n'afficher des valeurs que lorsque les ports de sortie varient, il faut réduire la liste de sensibilité dans le processus monitor à :

```
sensitive << m_sum << m_cout ;
```

6. Petit ajout personnel

Comme tout projet C++, la compilation d'un exécutable de simualtion SystemC est grandement facilitée par l'usage d'un fichier **`Makefile`**. Les exemples donnés avec la distribution SystemC 2.0.1 s'appuient sur deux fichiers **`Makefile.defs`** et **`Makefile.linux`**. Pour compiler le projet, il suffit de faire :

```
make -f Makefile.linux
```

Pour effacer toute trace des compilations, entrez :

```
make -f Makefile.linux clean
```

Dans le cas où vous utilisez une autre distribution de SystemC que celle installée dans **`/users/outil/m1archi/systemc-2.0.1`**, il faut modifier la ligne 2 du fichier **`Makefile.defs`**. Pour adapter le fichier **`Makefile.linux`** à vos projets, il suffit de modifier le contenu de sa ligne 11.

```
1. ## Variable that points to SystemC installation path
```

```
2. SYSTEMC = /users/outil/m1archi/systemc-2.0.1
3.
4.
5. INCDIR = -I. -I.. -I$(SYSTEMC)/include
6. LIBDIR = -L. -L.. -L$(SYSTEMC)/lib-$(TARGET_ARCH)
7.
8. LIBS   =  -lsystemc -lm $(EXTRA_LIBS)
9.
10.
11.EXE    = $(MODULE).x
12.
13..SUFFIXES: .cc .cpp .o .x
14.
15.$(EXE): $(OBJS) $(SYSTEMC)/lib-$(TARGET_ARCH)/libsystemc.a
16.     $(CC) $(CFLAGS) $(INCDIR) $(LIBDIR) -o $@ $(OBJS) $(LIBS) 2>&1 | c++filt
17.
18..cpp.o:
19.     $(CC) $(CFLAGS) $(INCDIR) -c $<
20.
21..cc.o:
22.     $(CC) $(CFLAGS) $(INCDIR) -c $<
23.
24.clean::
25.     rm -f $(OBJS) *~ $(EXE) core
26.
27.ultraclean: clean
28.     rm -f Makefile.deps
29.
30.Makefile.deps:
31.#    $(CC) $(CFLAGS) $(INCDIR) -M $(SRCS) >> Makefile.deps
32.
33.#include Makefile.deps
```

Listing 2-12 : le fichier **`Makefile.defs`**.

```
1. TARGET_ARCH = linux
2.
3. CC     = g++
4. OPT    = -O3
5. DEBUG  = -g
6. OTHER  = -Wall -Wno-deprecated
7. CFLAGS =  $(OTHER) $(DEBUG)
8. # CFLAGS = $(DEBUG) $(OTHER)
9.
10.MODULE = run
11.SRCS = driver.cpp full_adder.cpp full_adder_main.cpp half_adder.cpp monitor.cpp
12.OBJS = $(SRCS:.cpp=.o)
13.
14.include ./Makefile.defs
```

Listing 2-13 : le fichier **`Makefile.linux`**.

7. Pour générer un fichier de trace au format VCD (pour `gtkwave`)

Pendant la simulation, il est possible de générer des fichiers de trace au format VCD pour visualisation ultérieure à l'aide de **`gtkwave`**. Pour cela, il faut modifier les dernières lignes du fichier principal, **`full_adder_main.cpp`** dans notre exemple. Cela se fait en 4 étapes.

Etape 1: Il faut ouvrir un fichier de trace à l'aide de la fonction **`sc_create_vcd_trace_file()`**

Etape 2 : A l'aide de la fonction **`sc_trace()`**, ajouter les signaux à tracer. Le premier arcgument est le pointeur vers le fichier de trace ouvert à l'étape 1. Le deuxième argument de

la fonction est le **sc_signal** à tracer. Il est possible de spécifier un signal interne, contenu dans une instance, en donnant son cheminom. Dans notre exemple, pour tracer le signaux **c2** de l'instance **f1** du full_adder, il entrer **f1.c2.** Le troisième argument est la chaîne de caractères représentant le signal dans le fichier gtkwave. Il ne faut pas insérer d'espaces dans cette chaîne.

Etape 3 : Une fois que le simulateur est averti de la liste des signaux à tracer, il faut lancer la boucle de simulation proprement dite, à l'aide des fonctions **sc_start()** ou **sc_cycle()** (auquel cas il faut avoir fait un **sc_initialize()** avant). A chaque fois qu'un signal change de valeur, cette modification est propagée dans le fichier de traces.

Etape 4: Enfin, il faut fermer le fichier ouvert à l'aide de la fonction **sc_close_vcd_trace_file()**

```
sc_initialize();

//       (1) Ouvrir le fichier de trace, au format VCD

sc_trace_file *tfile=sc_create_vcd_trace_file("results");

//       (2) Rajouter les signaux que l'on souhaite tracer
//       Le premier argument est un pointeur de fichier
//       Le deuxième est le nom du signal.
//       Le troisième est le nom avec lequel le signal va être
//       affiché par gtkwave. Les espaces dans le nom sont
//       interdits


sc_trace(tfile,t_a,"t_a") ;
sc_trace(tfile,t_b,"t_b") ;
sc_trace(tfile,t_cin,"t_cin") ;
sc_trace(tfile,t_sum,"t_sum") ;
sc_trace(tfile,t_cout,"t_cout") ;

...      (3) Lancer la simulation : sc_start() ou sc_cycle()

//       (4) Fermer le fichier de traces

sc_close_vcd_trace_file(tfile);

return(0);
```

Listing 2-14 : Pour générer un fichier de traces.