Introduction à SystemC

Les types

Tarik Graba

2020-2021

Table des matières

1 Le type logique

Le type logique

En C++ le type `bool` permet de représenter une valeur logique

- (`true`, `false`).

SystemC définit, en plus, le type `sc_logic` pouvant prendre 4 valeurs

- (`0`, `1`, `Z`, `X`).

```
sc_logic u;
```

- Il peut être initialisé à partir d'un booléen, un entier ou un caractère.
- Il existe des fonctions de conversion explicite vers un booléen, un entier ou un caractère.
- Il peut être utilisé avec les opérateurs binaires (`~`, `&`, `|`, `^`) et les opérateurs d'égalité.
  - On peut même mélanger un `sc_logic` et un booléen, un entier ou un caractère.

La description complète de ce type se trouve dans la section 7.9.2 de la norme.

Les quatre valeurs correspondent aux états logiques suivant :

- `0` : faux
- `1` : vrai
- `Z` : haute impédance
- `X` : Non défini, inconnu, conflit

En interne, un `sc_logic` contient un élément du type énuméré `sc_logic_value_t` défini comme :

```
enum sc_logic_value_t {
   Log_0=0;
   Log_1,
   Log_Z,
   Log_X
};
```

Donc, ces 4 valeurs possibles sont des valeurs entières.

En plus du constructeur par défaut et du constructeur de copie, un `sc_logic` peut être construit à partir :

- d'un booléen
  - `false` -> `0` et `true`-> `1`

- d'un entier
  - seules les valeurs entières entre 0 et 3 sont autorisées
- d'un caractère
  - les caractères `'0'`, `'1'`, `'x'`, `'z'`, `'X'`, `'Z'` peuvent alors être utilisés
  - tous les autres sont équivalents à `x`

Ces constructeurs sont définis comme `explicit` et ne peuvent donc pas être utilisés pour les conversions implicites.

Par contre, les opérateurs binaires ainsi que les opérateurs d'affectation sont définis pour pouvoir utiliser les `sc_logic` avec des booléens, des caractères ou des entiers.

Il existe aussi des méthodes pour convertir ou tester ces valeurs (`to_bool`, `to_char`, `is_01`) qui peuvent être appelées si besoin.

Pour imprimer la valeur d'une variable de type `sc_logic` l'opérateur de flux « a été surchargé. En général, l'opérateur de flux est surchargé pour tous les types définis dans SystemC.

Travail à faire : mise en pratique des `sc_logic`

Dans une fonction `sc_main` :
- déclarez plusieurs variables de type `sc_logic`,
- testez différentes initialisations,
- testez différents opérateurs binaires entre `sc_logic`,
- testez différents opérateurs binaires entre des `sc_logic` et des `bool`,
- Imprimez systématiquement les résultats.

Remarque Il existe aussi un type `sc_bit` ne pouvant prendre que les 2 valeurs (`0`, `1`) mais il est déprécié depuis la version 2.2 ce SystemC. Dans ce cas, on lui préférera le type `bool` de C++.

---

2 Les vecteurs

Les bit vectors `sc_bv<N>`

Vecteur d'éléments binaires (`bool`).

```
sc_bv<8> x;
```

Le paramètre de template est un entier qui permet de définir la taille du vecteur.

Les logic vectors `sc_lv<N>`

Vecteur de variables logiques (`sc_logic`).

```
sc_lv<8> x;
```

Le paramètre de template est un entier qui permet de définir la taille du vecteur.

Les types `sc_lv` et `sc_bv` sont des vecteurs. Ils ont été conçus pour permettre l'accès efficace à leurs différents éléments. Par contre, ils ne permettent de faire des opérations arithmétiques.

Pour l'arithmétique, nous verrons par la suite des types plus adaptés.

Les vecteurs peuvent être comparés et initialisés à partir d'une chaîne de caractère. Par exemple :

```
// vecteur logic de 4 bits de large
sc_lv<4> x = "01xz";

if (x == "0000")
   ...
```

Ici chaque caractère représente un des éléments logiques.

On peut aussi initialiser un vecteur à partir d'un `bool` ou d'un `char`. Dans ce cas tous les éléments auront la même valeur. Par exemple :

```
// initialise tous les éléments d'un vecteur de 8bits à false?
sc_bv<8> x = false;
sc_bv<8> y = sc_bv<8>(false);
sc_bv<8> z(false);
   ...
```

Les vecteurs peuvent aussi être initialisés à partir d'un entier. Par exemple :

```
//
sc_bv<8> X;

X = 44;
// Imprimera 44
cout « ”x---> ” « X  « ”(” « X.to_uint() « ”)” « endl;

// Imprimera 44 aussi car X est sur 8 bits
X = 300;
cout « ”x---> ” « X  « ”(” « X.to_uint() « ”)” « endl;
   ...
```

Les opérateurs binaires (~, &, |, ^) ainsi que les opérateurs de décalage (», «) sont supportés.

De plus, certains opérateurs ont été surchargés pour permettre la sélection d'un ou plusieurs éléments. Ils peuvent être utilisés des deux cotés d'une affectation.

```
sc_bv<8> X;

X[0] = false;
sc_bit b = X[1];

X(7,4) = ”1010”;
X(3,0) = X(7,4);
```

La méthode `range` peut aussi être utilisée pour la sélection de plusieurs éléments (i.e. `X(n1,n2)` est équivalent à `X.range(n1,n2)`).

La concaténation de deux vecteurs se fait en les mettant entre parenthèses. C'est l'opérateur `','` qui est surchargé.

```
sc_bv<8> X = ”11110000”;
sc_bv<4> a;
sc_bv<4> b;

(a,b) = X;
X = (b,a);
```

La fonction `concat` peut aussi être utilisée pour la concaténation de deux éléments (i.e. `(n1,n2)` est équivalent à `concat(n1,n2)`).

D'autres méthodes sont implémentées, par exemple :

- Les méthodes de réduction :
  - `and_reduce()`
  - `or_reduce()`
  - `xor_reduce()`

  - nand_reduce()
  - nor_reduce()
  - xnor_reduce()
- Les méthodes de conversion :
  - to_int()
  - to_uint()
  - to_long()
  - to_ulong()
  - to_int64()
  - to_uint64()

Travail à faire : mise en pratique des `sc_lv` ***et*** `sc_bv`

Dans une fonction `sc_main` :
- déclarez plusieurs variables de ces deux types,
- testez différentes initialisations,
- testez différents opérateurs binaires,
- testez différentes conversions entre les deux types,
- imprimez systématiquement les résultats.

---

3 Les entiers de taille arbitraire

Remarque

Pour des raisons d'efficacité, les types `sc_lv` et `sc_bv` ne permettent pas de faire d'opérations arithmétiques.

Pour cela, la bibliothèque SystemC fournit des types permettant de manipuler des entiers de taille arbitraire.

`sc_int<N>` et `sc_uint<N>`

SystemC définit ces deux classes pour les entiers de taille inférieure ou égale à 64 bits.

```
sc_int<10> x; // entier signé sur 10 bits
sc_uint<7> y; // entier non signé sur 7 bits
```

En interne, ils utilisent les entiers 64 bits natifs de C++ pour avoir une implémentation efficace des opérateurs arithmétiques. Les opérations de troncature et d'extension sont gérées automatiquement.

Ces types supportent tous les opérateurs arithmétiques et logiques supportés par les entiers en C++. En plus, comme pour les types vecteurs, ils supportent les opérateurs de sélection de bits ou de plages de bits, de concaténation, ainsi que les opérateurs de réduction.

Ils peuvent être initialisés par :

- un entier,
- un `sc_lv` ou un `sc_bv` ce qui est utile pour pouvoir faire des opérations arithmétiques sur ces types.

En pratique, il faut privilégier les types C++ natifs. Si l'on a besoin d'une taille arbitraire qui ne correspond pas à un type natif, on utilise alors les `sc_int` et `sc_uint`. Ce n'est que si l'on a besoin de modéliser un bus 3 états (Z) qu'on utilise les `sc_lv`.

Travail à faire : mise en pratique des `sc_int` ***et*** `sc_uint`

Dans une fonction `sc_main` :
- déclarez plusieurs variables de ces deux types,
- testez différentes initialisations,
- testez différents opérateurs arithmétiques et logiques,
- testez différents conversions entre ces types et des `sc_lv`,
- imprimez systématiquement les résultats.

sc_bigint<N> et sc_biguint<N>

Si l'on a besoin de représenter des entiers de taille arbitraire supérieure à 64 bits, il faut utiliser ces types.

Ils sont prévus pour permettre des calculs sur une taille illimitée mais pour des raisons pratiques, l'implémentation de référence limite leur taille à `SC_MAX_NBITS`. Cette valeur est définie dans le fichier `kernel/sc_constants.h` et peut être modifiée si nécessaire.

4 Les nombre en virgule fixe

Les types définis

Pour représenter un nombre en virgule fixe on doit préciser sa taille et la position de la virgule.

SystemC propose deux familles de types pour représenter ces nombres :

- Les `sc_fixed` et `sc_ufixed` pour lesquels les paramètres sont statiques (définis par des templates)
- Les `sc_fix` et `sc_ufix` pour lesquels les paramètres peuvent être définis dynamiquement

---

En plus, des versions `_fast` sont définies. Ces types utilisent en interne les `doubles` natifs de C++ pour être plus efficaces mais sont limités à la taille de ceux-ci.

Les `sc_fixed` et `sc_ufixed`

```
sc_fixed <wl, iwl, q_mode, o_mode, n_bits> X;
sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> Y;
```

avec :

- `wl` la taille totale
- `iwl` la taille de la partie entière
- `q_mode` le mode de quantification
- `o_mode` le mode de saturation
- `n_bits` le nombre de bits qui saturent

Les paramètres `wl` et`iwl` doivent être définis à l'instanciaton. Les autres paramètres ont par défaut les valeurs suivantes :

- `q_mode` : `SC_TRN` // troncature vers le bas
- `o_mode` : `SC_WRAP` // modulo
- `n_bits` : `0` // aucun bit ne sature

La définition exacte de ces paramètres et des comportements associés se trouvent dans la section 7.10 du standard.

Remarque

Par défaut, les types permettant de représenter des nombres en virgule fixe, ne sont pas disponibles. Pour les utiliser il faut définir la macro `SC_INCLUDE_FX` avant d'inclure `systemc.h`.

Comment les utiliser

Ces types peuvent être initialisés à partir de `float` ou de `double`.

Toutes les opérations arithmétiques et logique sont faites en fonction des paramètres sur lesquels ils ont été définis.

Ces types peuvent, par exemple, être utilisés pour tester simplement une implémentation en virgule fixe et comparer la précision à une implémentation flottante.

Par exemple :

```
#define SC_INCLUDE_FX
#include <systemc.h>

// 1 bit pour la partie entière et 2 bits après la virgule
#define WL  3
#define IWL 1

int sc_main (int argc, char * argv[])
{

   double d1 = 0.1;
   double d2 = 0.9;
   double ds = d1 + d2;

   sc_fixed<WL,IWL> f1(d1);
   sc_fixed<WL,IWL> f2 = d2;
   sc_fixed<WL,IWL> fs = f1 + f2;

   double e = ds - double(fs);

   cout
      « "Double " « ds  « endl
      « "Fixed  " « fs  « endl
      « "Error  " « e   « endl;
```

```
    return 0;
}
```

Les sc_fix et sc_ufix

Ces deux types diffèrent des précédents par le fait que les paramètres sont modifiables dynamiquement en fonction d'un contexte. Comme les paramètres sont dynamiques, ils sont plus lents à l'exécution.

Pour définir un contexte :

```
// ici seule la taille est définie, mais tout est modifiable
sc_fxtype_params param(10,3);
sc_fxtype_context ctxt(param);

sc_fix a;   // paramètre définis dans le contexte
```

Exemple d'utilisation :

```
#define SC_INCLUDE_FX
#include <systemc.h>

double compare (const double d1, const double d2)
{
    double ds = d1 + d2;
    sc_fix f1(d1);
    sc_fix f2(d2);
    sc_fix fs = f1 + f2;

    return ds - fs;
}

int sc_main (int argc, char * argv[])
{

    double d1 = 0.1;
    double d2 = 0.9;

    // 1 bit pour la partie entière et 2 bits après la virgule
    sc_fxtype_params param1(3,1);
    // 1 bit pour la partie entière et 4 bits après la virgule
    sc_fxtype_params param2(5,1);
    // 1 bit pour la partie entière et 6 bits après la virgule
```

```
    sc_fxtype_params param3(7,1);

    sc_fxtype_context ctxt1(param1);
    cout
       « "The error with " « ctxt1.default_value()
       « " format is " «  compare(d1,d2)  « endl;

    sc_fxtype_context ctxt2(param2);
    cout
       « "The error with " « ctxt1.default_value()
       « " format is " «  compare(d1,d2)  « endl;

    sc_fxtype_context ctxt3(param3);
    cout
       « "The error with " « ctxt1.default_value()
       « " format is " «  compare(d1,d2)  « endl;

    return 0;
}
```

De façon analogue aux `sc_fixed`, le type `sc_fxtype_params` prend, entre autres, comme paramètres de construction la taille totale de la représentation ainsi que la taille de la partie entière.

Travail à faire : mise en pratique

1. Testez les exemples fournis
2. Écrire une fonction qui permet de trouver le nombre de bits minimum pour représenter **π.sin(π/256 n)** avec une précision de 1/100, 1/1000 et 1/10000 par rapport à un calcul en double précision (**n** étant un entier).

---

5 Le temps

Le type `sc_time`

SystemC définit un type pour représenter le temps de simulation ou des intervalles de temps.

Un `sc_time` est construit à partir d'un `double` et d'une unité. L'unité est une des valeurs du type énuméré `sc_time_unit`

```
sc_time periode(17.22, SC_NS);
```

Le type `sc_time_unit` est défini comme suit :

```
enum sc_time_unit {
   SC_FS = 0,  // femtosecondes
   SC_PS,      // picosecondes
   SC_NS,      // nanosecondes
   SC_US,      // microsecondes
   SC_MS,      // millisecondes
   SC_SEC      // secondes
};
```

En interne, le temps est stocké sous la forme d'un entier 64 bits qui est un multiple de la résolution temporelle. Cette valeur interne peut être récupérée en utilisant la méthode `value`.

La résolution temporelle est globale à une simulation. Par défaut elle est égale à **1 picoseconde** mais peut être modifiée en utilisant la fonction `sc_set_time_resolution`.

Quelle que soit la résolution, la constante `SC_ZERO_TIME` représente toujours un temps nul.

Le type `sc_time` supporte des opérateurs de comparaison ainsi que certains opérateurs arithmétiques. On peut ainsi additionner deux temps ou les multiplier par un double.

Exemple d'utilisation :

```
#include <systemc.h>

int sc_main (int argc, char * argv[])
{
   sc_time t1(3.51, SC_NS);
   cout « "---> " « t1.value() « ", " « t1 « endl;
```

```
    sc_time t2 = 2*t1;
    cout « "---> " « t2.value() « ", " « t2 « endl;

    cout « "La durée t" ;
    if (t1>t2)
       cout « "t1";
    else
       cout « "t2";
    cout « " est plus grande" « endl;

    cout « "---> " « SC_ZERO_TIME.value() « ", " « SC_ZERO_TIME « endl;

    return 0;
}
```
