Introduction à SystemC

Niveaux d'abstraction et raffinement

Tarik Graba

P4 2015-2016

Télécom ParisTech

Table des matières

1 Niveaux d'abstraction et raffinement

Fonctionnel vers RTL

En SystemC on peut partir d'une description fonctionnelle et aller jusqu'à une représentation RTL.

Comment utiliser les processus de SystemC pour faire cela.

Les SC_METHOD correspondent aux processus qu'on trouve dans les autres langages HDL. Elles permettent donc de modéliser au niveau RTL.

Pour des modélisations plus haut niveau, les SC_THREAD doivent être utilisés. Ils permettent de passer d'une modélisation fonctionnelle à une modélisation précise au bit et cycle près (CABA). Les SC_THREAD permettent de faire ces modifications avec le minimum de modification.

Étapes de raffinement

1. Décrire l'algorithme
2. Écrire une version fonctionnelle
3. Définir l'interface du module
   - les entrées/sortie
   - un éventuel protocole
4. Encapsuler la fonction dans un SC_THREAD
   - la version fonctionnelle servant de référence
5. Ajouter une information sur le temps en ajoutant des `wait`
   - en nombre de cycles ou en temps absolu
6. Refaire une version RTL... en utilisant des SC_METHOD
   - la version précédente servant de référence .

Une fonction logicielle prend des arguments et renvoie un résultat. Un module matériel a des entrées et des sorties. En plus, un protocole particulier peut être utilisé pour indiquer que les entrées/sorties sont prêtes.

```
int f(int i, int j ...)
{
```

```
   return xx;
}
```

La première chose à faire est donc de définir un `sc_module` dont les entrées/sortie correspondent au protocole qui sera utilisé. En suite, la fonction peut être encapsulée dans un `SC_THREAD`.

```
sc_module tt {
   sc_in<int> i;
   sc_in<int> j;
   ...
   sc_out<int> xx;

   void mthread()
   {
   // Attendre que les entrées soient prêtes
   ...
   // Exécuter la fonction
   xx = f(i,j,..)
   // prévenir que la sortie est prête
   }

   SC_CTOR()
   {
      SC_THREAD(mthread);
      ...
   }
}
```

Dans ce `SC_THREAD` on attend que les entrées soient prête, puis on appelle juste la fonction que nous voulons modéliser. Ceci permet d'avoir un premier modèle fonctionnel et respectant le protocole prévu. Par contre, aucune notion de temps n'existe pour l'instant.

Dans un vrai module matériel, l'exécution de la fonction prend du temps (nombre de cycles dans une implémentation séquentielle, par exemple). Si on veut ajouter une notion de temps à notre modèle, on peut, dans un `SC_THREAD` appeler la méthode `wait()` entre l'appel à de la fonction et le renvoie du résultat. Le `SC_THREAD` se mettra alors en veille et le résultat ne sera disponible qu'au bout du temps précisé.

Par exemple

```
sc_module tt {
   sc_in<int> i;
   sc_in<int> j;
   ...
   sc_out<int> xx;

   void mthread()
   {
```

```
   // Attendre que les entrées soient prêtes
   ...
   // Exécuter la fonction
   xx = f(i,j,..)
   // Ajouter la latence
   wait(xx,SC_NS);
   // prévenir que la sortie est prête
   }

   SC_CTOR()
   {
      SC_THREAD(mthread);
      ...
   }
}
```

Pour de la logique séquentiel, les temps de mise en veille du processus doivent être des multiples de la période d'horloge. De plus, pour garantir le synchronisme, doivent dépendre de l'activité d'un signal d'horloge. Dans ce cas, les SC_CTHREAD peuvent simplifier certaines écritures.

Par exemple

```
sc_module tt {
   // L'horloge
   sc_in<bool> clk;
   // Les autres I/O
   sc_in<int> i;
   sc_in<int> j;
   ...
   sc_out<int> xx;

   // Latence en nombre de cycles
   static const unsigned int LAT = xxx;

   void mthread()
   {
   // Attendre que les entrées soient prêtes
   ...
   // Exécuter la fonction
   xx = f(i,j,..)
   // Ajouter la latence
   for (int i=0, i<LAT, i++)
      wait();
   // prévenir que la sortie est prête
   }
```

```
   SC_CTOR()
   {
      SC_THREAD(mthread);
      sensitive << clk.pos();
      ...
   }
}
```

Par exemple avec un SC_CTHREAD

```
sc_module tt {
   // L'horloge
   sc_in<bool> clk;
   // Les autres I/O
   sc_in<int> i;
   sc_in<int> j;
   ...
   sc_out<int> xx;

   // Latence en nombre de cycles
   static const unsigned int LAT = xxx;

   void mthread()
   {
   // Attendre que les entrées soient prêtes
   ...
   // Exécuter la fonction
   xx = f(i,j,..)
   // Ajouter la latence
   wait(LAT);
   // prévenir que la sortie est prête
   }

   SC_CTOR()
   {
      SC_CTHREAD(mthread, clk.pos());
      ...
   }
}
```

Pour augmenter la précision du modèle, il faut pouvoir tracer temporellement l'état des variables internes de la fonction modélisée. Pour cela, il faut pouvoir insérer des attentes (`wait`) dans le corps de la fonction elle-même. Il suffit pour cela que la fonction soit elle-même une méthode du `sc_module`

Par exemple

```
sc_module tt {
   // L'horloge
   sc_in<bool> clk;
   // Les autres I/O
   sc_in<int> i;
   sc_in<int> j;
   ...
   sc_out<int> xx;

   int f(int i, intj, ...)
   {
      // étape 1
      ....
      wait(x);
      // étape 2
      ....
      wait(y);
      // ...
      // étape n
      ....
      wait(z);
      return xx;
   }
   void mthread()
   {
   // Attendre que les entrées soient prêtes
   ...
   // Exécuter la fonction
   xx = f(i,j,..)
   // prévenir que la sortie est prête
   }

   SC_CTOR()
   {
      SC_CTHREAD(mthread, clk.pos());
      ...
   }
}
```

2 Exemple pratique, Calculer le PGCD

Choix de l'algorithme

Nous voulons modéliser un module matériel calculant le PGCD de deux nombres entiers.

L'algorithme d'Euclide peut être utilisé pour cela. Il existe deux variantes de cet algorithme utilisant :

- des divisions successives ou
- des soustractions successives.

Pour une implémentation matérielle, nous utiliserons la variante avec des soustractions car elle utilisera moins de ressources.

Travail à faire :

Écrire une fonction qui prend deux entiers non signés et qui renvoie leur PGCD en utilisant la variante en soustraction successive de l'algorithme d'Euclide.

Définir les interfaces

Comme l'algorithme est itératif, nous allons l'implémenter en utilisant de la logique séquentielle.

Le module aura l'interface représentée dans le schéma suivant :

- Les données sont des données non signées représentées sur 8 bits.
- Des signaux de contrôle sont ajoutés :
  - le signal `ready` passe à 1, durant 1 cycle, quand les entrées sont prêtes,
  - le signal `valid` passe à 1, durant 1 cycle, quand le résultat est prêt,
  - si `ready` passe à 1 durant le calcul, le comportement est indéterminé.
  - le résultat n'a pas a être maintenu au delà du cycle durant lequel `valid` est à 1

Travail à faire

1. Écrire un module ayant cette interface.
2. Implémentez le comportement du module en respectant le protocole (`ready`/`valid`).

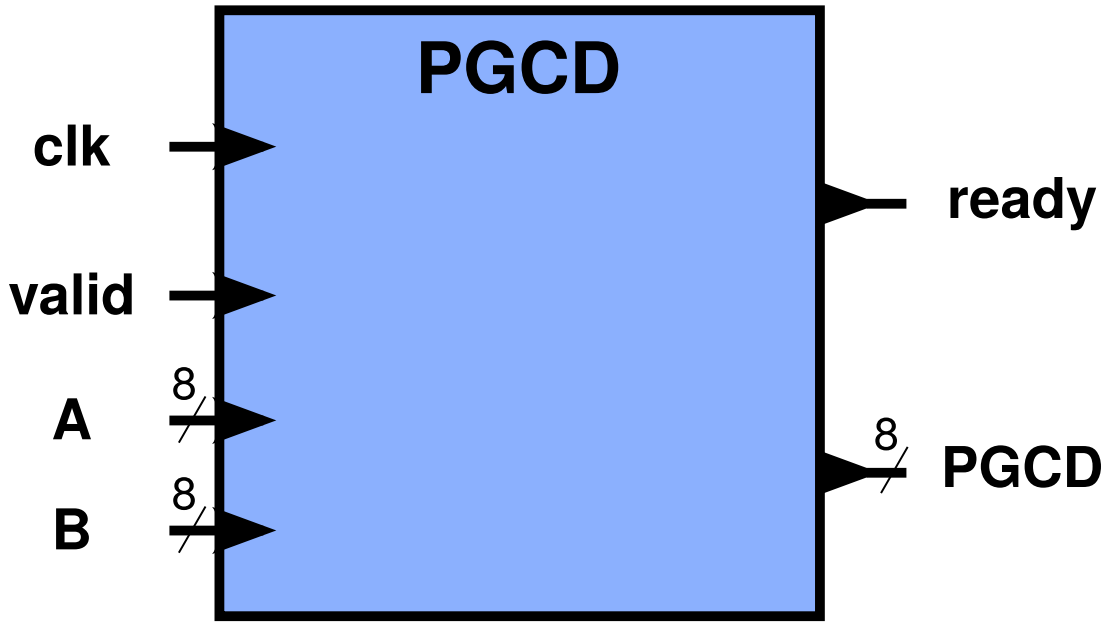

FIG. 2.1: Interface du module PGCD

- Utilisez pour cela des `SC_CTHREAD`.

3. Instanciez le module dans un `sc_main` où vous testerez exhaustivement le calcul.
   - Vous comparera le résultat renvoyé par le module au résultat de l'exécution de la fonction de la première étape.
4. Générez les traces des différents signaux.

Modèle temporel

Modifiez le modèle pour avoir un comportement temporel réaliste.

Le nombre d'étapes de calcul pour calculer le PGCD en utilisant l'algorithme d'Euclide dépend des données en entrée.

Dans une implémentation séquentielle simple, chaque étape de calcul peut se faire en un cycle d'horloge.

Travail à faire

1. Modifiez le modèle précédent pour que le comportement temporel soient plus réaliste.
   - Nous voulons obtenir un modèle CABA.
2. Le test précédent devrait fonctionner sans modification.

3. Générez les traces.

Modèle RTL

Cette figure représente le chemin de donnée d'une implémentation possible de cet algorithme.

Travail à faire

1. En utilisant exclusivement des `SC_METHOD` proposez une implémentation RTL du module calculant le PGCD.
    - Séparez le chemin de donnée du contrôle
2. Ce module doit avoir une interface compatible avec les versions précédentes.
    - Vous devriez pouvoir tester en parallèle la CABA
    - Observez-vous de différences temporelles ? Expliquez les.
3. Générez les traces.

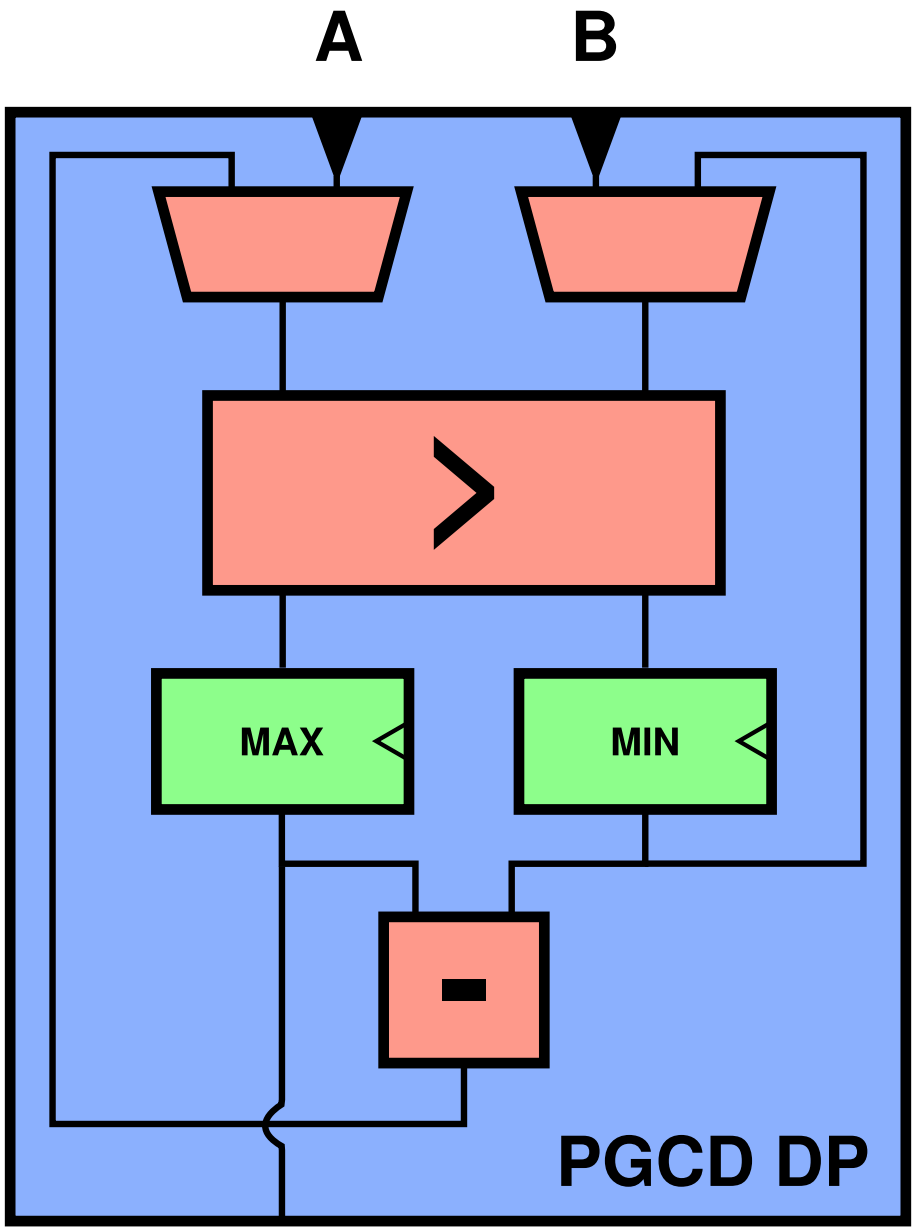

Fig. 2.2: Datapath du PGCD