# Représentation matérielle des opérateurs du langage C pour la synthèse haut niveau

François Donnet, Ivan Augé
Laboratoire ASIM/LIP6
Université Pierre et Marie Curie (UPMC)
4, place Jussieu, 75252 Paris cedex 5

E-mail: `francois.donnet@lip6.fr, auge@iie.cnam.fr`

## Résumé

*Au cours des dernières années, la synthèse de circuits a progressé vers un plus grand niveau d'abstraction, mais les outils existants utilisent souvent des langages d'une très fine granularité dont les caractéristiques sont héritées de la conception VLSI bas niveau. Utiliser un langage plus généraliste et permissif correspond mieux aux besoins du concepteur de modèles haut niveau. Le langage C bénéficie d'une large diffusion et d'une très forte adaptabilité. Il est déjà utilisé pour la phase de simulation précédant la synthèse, le lien entre simulation et synthèse s'en trouve donc renforcé.*

## 1. Introduction

Plusieurs systèmes de synthèse de haut niveau ont été proposés au cours des dernières années. La description en entrée est généralement une description procédurale, utilisant un langage de description hardware comme le VHDL ou le C++ (comme pour *SystemC* [1]), et la sortie est une architecture de niveau Register-Transfer Level. Cet article repose sur l'expérience de l'ensemble de CAO `disydent` développé au LIP6. Le système est écrit comme un ensemble de tâches asynchrones à gros grains commmunicant via des Fifos. Ce modèle nous ramène à un graphe de Kahn [2]. La description fonctionnelle de chaque tâche est faite en langage C enrichit par des primitives de communication de lecture/écriture bloquantes qui assurent la synchronisation entre les tâches. Ces fonctions d'accès reposent sur des threads POSIX. L'ensemble des tâches peut être compilé pour produire un exécutable servant à valider leurs bons fonctionnements et leurs communications. La synthèse d'un ou de plusieurs des coprocesseurs est faite par l'outil *UGH* -User Guided High level synthesis-. Cet outil prend en entrée une tâche de la description fonctionnelle, un chemin de données simplifié -notamment pas de taille, ni de multiplexeurs ni de constantes- et une fréquence cible (généralement la fréquence du système). Il produit une description RTL synthétisable du data-path et de l'automate à états finis. Il génère aussi un modèle C matériel pour la simulation de *CASS* -Cycle Accurate System Simulator-.
La première partie traite du fonctionnement de la synthèse. La section suivante décrit l'entrée C supportée par la synthèse et les opérateurs matériels correspondants. La dernière partie détaille le dimensionnement de ce matériel.

## 2. Vue générale de la synthèse

Les Fifos étant les seules entrées/sorties du modèle, on peut donc s'affranchir de la déclaration de son interface. L'architecture sera un chemin de données contrôlé par une machine à états finis.

```
/* thread posix du PGCD pour la simulation et la synthèse */
void pgcd_main() {
    ugh_inChannel32  Din;
    ugh_outChannel32 Dout;
    uint32           Ra, Rb;
    while( 1 ) {                    /* process */
        ugh_read( Din,  &Ra );
        ugh_read( Din,  &Rb );

        while( Ra != Rb ) {
            if ( Ra < Rb ) Rb = Rb−Ra;
            else           Ra = Ra−Rb;
        }
        ugh_write( Dout, &Ra );
    }
}
```

FIG. **1. Description C du PGCD.**

### 2.1. Formalisation de la description

Dans la première phase, le C est compilé en un modèle formel basé sur les réseaux de Pétri temporels [3]. Le réseau de Pétri représente la structure de contrôle du flux de données. Chaque processus est composé de places et de transitions. Les places font référence aux états du processus et les transitions sont les actions exécutées en passant à l'état suivant. La machine à états est directement déductible de la forme du réseau de Pétri [4].

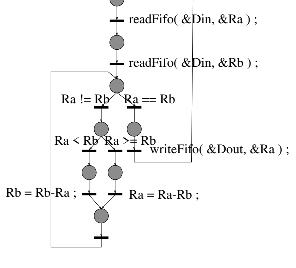


FIG. **2. Le PGCD en réseau de Pétri.**

## 2.2. Chemin de données

Le chemin de données est contrôlé par la machine à états grâce aux multiplexeurs présents devant chaque entrée d'instance. Les expressions contenues dans les transitions sont traduites par une suite d'opérateurs arithmétiques ou logiques dans le chemin de données (cf. l'exemple sur le PGCD Fig. 3). Pour respecter la séquentialité des instructions, les variables intermédiaires sont représentées par des registres qui stockeront les valeurs intermédiaires à chaque changement d'état du contrôle. Le chemin de données est construit en fonction de la description C et des contraintes matérielles définies par le concepteur. Ce dernier peut donc contraindre très fortement l'utilisation de certains objets et peut décider de l'aspect global du chemin de données.

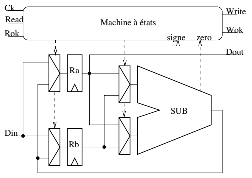


FIG. **3. Synthèse du PGCD.**

# 3. Subset du langage C

Il reste des contraintes de conception intrinsèques à la synthèse de haut niveau [5]. Voici le sous ensemble du langage C accepté pour la synthèse.

## 3.1. Les instructions

Toutes les instructions de branchements et de tests sont synthétisables.

**Branchements cablés** Il est possible de marquer un branchement à l'aide du mot clef *ugh_wired* dans le test de la description comportementale. Dans ce cas, l'automate garde sa séquentialité et c'est au data-path de contrôler les actions de branchement. Ceci afin d'économiser deux états effectifs dans la fsm (exemple sur le *PGCD* Fig. 4).

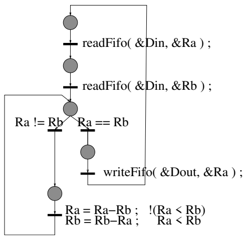


FIG. **4. Réduction du réseau de Pétri avec un branchement cablé.**

Ceci entraine une augmentation de la taille du chemin de données qui doit gérer ces branchements cablés (Fig. 5).

## 3.2. Les types

On peut considérer qu'il existe deux types de variables : les variables entières internes et les variables

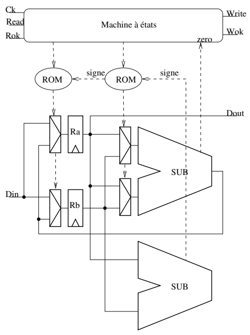


FIG. **5. Augmentation du data-path avec un branchement cablé.**

d'entrées/sorties. A l'instar du VHDL, il est nécessaire de pouvoir définir ces types au bit près. Or on dispose d'une liste de types prédéfinis symbolisant cette taille :

– vecteur 23 bits signé
  typedef int int23 ;
– vecteur 28 bits non signé
  typedef unsigned int uint28 ;
– Fifo entrante de 20 bits de largeur
  typedef unsigned int ugh_inChannel20 ;
– Fifo sortante de 16 bits de largeur
  typedef unsigned int ugh_outChannel16 ;

**Ports** Les ports du modèle sont des variables déclarées en tant que *Channel*. Le sens du canal précise le sens du port. On remarque qu'il ne peut y avoir de ports *inout* car les fifos qui sont connectées à ces ports ne peuvent l'être. Les *Channels* sont interdits dans les expressions régulières hormis en paramètre des fonctions d'accès qui leur sont dédiées (cf. les fonctions prédéfinies Chap.3.4).

**Entiers** Ces variables sont censées représenter des registres. Mais elles peuvent aussi être temporaires et être transformées en connectique. Si la variable est déclarée en tant que constante, on n'a pas besoin d'élément mémorisant, un simple générateur de constantes suffit.

**Flottants** Nous avons choisi de ne pas représenter les réels car nous destinons l'outil *UGH* exclusivement à la synthèse de coprocesseur sur entiers.

**Structures** Les structures ne peuvent être composées que de variables simples ou de structures. Elles représentent une seule entité matérielle. Le champ d'une structure n'est autre qu'un groupe particulier de bits du registre représentant cette structure. Lorsque l'on utilise une structure en C, on veut pouvoir accéder en lecture et en écriture à un champ précis. L'écriture dans un champ ne peut être fait par la seule modification d'une partie des bits d'un élément mémorisant. En effet, tous ses bits doivent être modifiés en même temps, on procède donc à une réécriture sur les bits non modifiés.

**Tableaux** Ils peuvent avoir une réalité hardware s'ils ne contiennent que des variables simples ou des structures. S'ils sont multi-dimensionnels, les indices sont agrégés pour ramener le tableau à une seule dimension. Ils peuvent représenter des bancs de registres ou des RAMs. Si le tableau est déclaré en tant que constante, celui-ci sera naturellement synthétisé en ROM.

**Pointeurs** Ils ne sont tolérés que pour déclarer des paramètres de fonction. Ils ont la même fonctionnalité que le type *inout* du VHDL. Il faudra alors, lors de l'appel de telle fonction, utiliser l'opérateur de référencement *&* pour ces paramètres.

## 3.3. Les opérateurs

Les opérateurs sont ceux du C et obéissent, à quelques exceptions près, aux mêmes contraintes.

**De référencements** L'opérateur d'indirection * est interdit alors que celui de référencement *&* n'est toléré que dans les appels de fonction (cf. les pointeurs Chap.3.2).

**Arithmétiques** La division et le reste, s'ils ne peuvent être simplifiés et absorbés, ne sont pas acceptés dans le subset C. L'addition et la soustraction nécessitent respectivement un additionneur et un soustracteur ou à fortiori, un additionneur-soustracteur ou une ALU. Tandis que la multiplication nécessite obligatoirement un multiplieur. Le moins unaire peut se faire sur un soustracteur *0-a* ou bien un additionneur $\overline{a} + 1$.

**Bit à bit** Losqu'elles ne sont pas absorbées par des constantes, les opérations (*&*, |, ˆ, ˜) sont instanciées par de la logique. Elles peuvent à la rigueur être faites sur des ALU.

**Booléens** Les opérateurs booléens *&&*, || et *!* du C ne sont pas directement transposables en matériel. En effet, il faut tout d'abord ramener les opérandes sur un bit pour pouvoir faire l'opération effective. La perte de la préemptivité pour les opérateurs binaires n'a presque aucun coût temporel pour le matériel puisque l'on procède toujours à l'évaluation en simultané des deux opérandes. S'il existe un effet de bord dans la seconde opérande, la préemptivité de l'opérateur rend cette assignation conditionnée par la valeur de la première opérande. Ceci est irréalisable en matériel et donc interdit dans le subset C.

**De décalages** Ils nécessitent des décaleurs gauche et droite, ce dernier devant être arithmétique si l'opérande est signée.

**Conditionnels** Cette expression ( *? :*) est transformée en un multiplexeur avec le résultat du test pour commande. On ne peut avoir d'effet de bord dans les termes de cette expression (cf. la préemptivité des booléens).

**D'égalités** == et *!*= nécessitent un comparateur sortant un résultat sur 1 bit. Ce comparateur n'est autre qu'une instanciation d'une chaîne de *xor*.

**Relationnels** Les inégalités <, >=, >, <= nécessitent un modèle sachant faire la soustraction pour respectivement faire $a - b < 0$, $\overline{a - b < 0}$, $b - a < 0$ et $\overline{b - a < 0}$. Les flags diffèrent selon que l'on compare des signés et des non signés.

- a et b signés, le résultat est la retenue sortante.
- a et b non signés, le résultat est *(OV xor NG)* où *OV* est l'overflow et *NG* le bit de poids fort du résultat.
- dans le cas contraire, le C recourt à une conversion de la plus petite opérande et toujours en faveur des non signés s'il y a égalité.

`Remarque` **:** En C, $a-b < 0$ diffère de $a < b$ car on perd le bit de carry s'il y a un dépassement de capacité dans la soustraction.

**Préfixés et post-fixés** Les opérateurs ++ ou −− sont assimilés à des effets de bord. Ils sont remplacés par des affectations qui sont placées respectivement avant et après l'expression.

**D'affectations** A moins que la destination ne soit une variable temporaire, ils représentent le changement de valeur d'un élément mémorisant. Les affectations combinant un opérateur et une assignation (+=,>>=,*&*= par exemple) sont décomposées en affectations simples avec introduction de l'opérateur dans le membre de droite.

**Le cast** Il limite l'accès en lecture aux bits de poids faibles d'un registre ou d'une expression. Si l'on veut obtenir les poids forts, on doit faire un masque à l'aide des opérateurs bit à bit.

**La virgule** Elle est assimilée à un point virgule. L'opérande gauche, qui doit contenir un effet de bord, est dissociée de l'expression pour être placée avant ou après celle-ci.

## 3.4. Les fonctions

**main** La fonction principale possède une boucle infinie qui symbolise le process. Les autres fonctions sont la plupart du temps destinées à être incluses dans la fonction principale.
Les variables globales ou statiques sont initialisées une seule fois (initialisées à *0* si elles ne possèdent pas de valeur d'initialisation).

**Inlined** Le code de ces fonctions est inclus dans l'automate principal à chacun de leur appel et sont donc absorbées dans le graphe de Pétri. Leurs variables locales ainsi que leurs paramètres sont préfixés du nom de la fonction pour éviter les collisions et deviennent des variables globales. Les paramètres de type pointeur, quant à eux, sont des variables temporaires et sont remplacés dans le code de la fonction par les variables constituant les paramètres de l'appel. Si les variables locales non statiques bénéficient d'une initialisation, l'assignation qui lui correspond est copiée dans le code pour chaque appel.

**Prédéfinies** Des fonctions d'entrées/sorties *ugh_readChannel(Channel, target)* et *ugh_writeChannel(Channel, source)* qui sont respectivement la lecture et l'écriture dans une Fifo.
Des fonctions de décalages pour une plus grande maîtrise de la taille de l'opérateur par l'utilisateur : *ugh_shl(size, data, shift)*, *ugh_shr(size, data, shift)* et *ugh_shra(size, data, shift)* décaleur à gauche, à droite et arithmétique. L'opérande *size* servant à déterminer précisément la taille de l'opérateur.
Des fonctions de concaténations : *ugh_catshr( size_data, size_in, data, in, shift)* qui décale à droite *data* de *shift* bits et concatène à gauche avec *in* sur autant de bits et *ugh_shlcat( size_data, size_in, data, in, shift)* qui est l'action en sens inverse.

**Autres** Seules les roms sont acceptées comme fonctions indépendantes. Elles ne prennent qu'un seul paramètre qui est leur clef. Elles sont composées uniquement de tests sur cette clef et des valeurs de retour pour chacune de ces comparaisons.

# 4. Dimensionnement en matériel des opérateurs C

Le dimensionnement d'un opérateur se fait en trois étapes chronologiques. Au final, on obtient la taille $t_s$ de notre opérateur et les tailles $t'_a$, $t'_b$ des entrées découlant des contraintes matérielles [6].

## 4.1. Taille du résultat

Pour conserver le résultat de l'opération (la retenue sortante), on est obligé d'agrandir la taille de la sortie (Tab. 1).

| Operateurs C | Taille du résultat $t_r$ |
|---|---|
| $a * b$ | $t_a + t_b$ |
| $a + b$, $a - b$ | $Max(t_a, t_b) + 1$ |
| $!a$, $a\&\&b$, $a\|\|b$, $a == b$, $a! = b$<br>$a < b$, $a <= b$, $a > b$, $a >= b$ | 1 |
| $\tilde{}a$, $-a$ | $t_a$ |
| $a\&b$ | $Min(t_a, t_b)$ |
| $a\|b$, $a\hat{}b$ | $Max(t_a, t_b)$ |
| $a << b$ | $t_a + (2^{t_b} + 1)$ |
| $a >> b$ | $Max(t_a, 2^{t_b} + 1)$ |
| $c?a : b$ | $Max(t_a, t_b)$ |
| $a/b$ | $t_a$ |
| $a\%b$ | $t_b$ |
| $a = b$, $(a)b$, $b.a$ | $t_a$ |

TAB. **1. Taille maximale du résultat d'une opération**

Dans certains cas (l'assignation par exemple), la taille de l'opérande peut être plus grande que celle acceptée par l'opérateur. On tronque alors les bits de poids forts de l'opérande.

## 4.2. Dépassement de capacité

Il peut arriver un dépassement de capacité en C, lorsque lors d'une opération arithmétique ou d'un décalage, le résultat dépasse la capacité d'un *integer* i.e 32 bits. Pour être en accord avec le C, on se doit de rester sur cette dimension de 32 bits. Les bits excédants seront donc éliminés comme lors d'un dépassement de capacité. Cependant si l'une des opérandes est supérieure à 32 bits, on décide de sortir des contraintes du C et on établit nos propres règles sur la dimension du résultat (Tab. 2).

| Operateurs C | Conditions | Taille sortie $t_s$ |
|---|---|---|
| $a * b$, $a + b$<br>$a - b$ | $t_a <= 32$,<br>$t_b <= 32$, $t_r > 32$ | 32 |
| $a << b$, $a >> b$ | $t_r > 32$ | $Max(32, t_a)$ |

TAB. **2. Taille de la sortie de l'opérateur matériel**

On peut remarquer une différence de règle selon les opérateurs. En effet, on ne peut pas se permettre de garder l'intégralité des bits excédentaires d'un décaleur.
Au final, c'est la taille de la sortie $t_s$ que l'on prendra pour dimensionner nos opérateurs matériels.

## 4.3. Contraintes des modèles matériels sur les opérandes

Le dimensionnement des sorties des opérateurs amènent des contraintes sur la taille des entrées (Tab. 3). Les opérandes seront étendues si nécessaire. Si l'expression est signée, l'extension se fait en prenant le bit de poids fort sinon elle est étendue avec la constante *0*.

| Operateurs C | Taille des entrées |
|---|---|
| $a\&b$ | $t'_a = Min(t_a, t_b)$<br>$t'_b = Min(t_a, t_b)$ |
| $a + b$, $a - b$, $a == b$, $a! = b$<br>$a < b$, $a <= b$, $a > b$, $a >= b$<br>$a\|b$, $a\hat{}b$, $c?a : b$ | $t'_a = Max(t_a, t_b)$<br>$t'_b = Max(t_a, t_b)$ |
| $a = b$ | $t'_b = t_a$ |
| $a << b$, $a >> b$ | $t'_a = t_s$<br>$t'_b = \lfloor log_2(t_s + 1) \rfloor$ |

TAB. **3. Redimensionnement des entrées de l'opérateur matériel selon $t_s$**

# 5. Conclusion

En supprimant l'étape de conception de modèle en vue de la synthèse, le concepteur s'affranchit, entre autres, des contraintes de typage qui lui sont liées tout en bénéficiant de la flexibilité du langage C. L'outil de synthèse *UGH* et de simulation *CASS* créent un environnement C alternatif au VHDL.
Nous avons pu ainsi mener à bien la simulation et la synthèse sur un même flot de données pour un décodeur de Hoffman.

# Références


[1] G. Economakos, P. Oikonomakos, I. Panagopoulos, I. Poulakis, G. Papakonstantinou. *Behavioral Synthesis with SystemC*. Proceedings in DATE 2001, p. 21-25, Munich, Germany, March 2001.

[2] G. Kahn. *The semantics of a simple language for parallel programming*. Information Processing 74, pages 471-475, Stockolm, Sweden, Aug. 1974.

[3] L. Jacomme. *Analyse sémantique de descriptions VHDL synchrones en vue de la synthèse*. Thèse de doctorat de l'Université Pierre et Marie Curie, Paris, Octobre 1999.

[4] A. Murata. *Petri Nets : Properties Analysis and Applications*. Proceedings in IEEE, vol. 77 N. 4, 541-580, April 1989.

[5] G. Pelissier, R. Hersemeule, STMicroelectronics CMSA team. How to accelerate C to VHDL design flows. Designer's Forum, DATE2001, p 114-116, March 2001.

[6] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, T. Sherwood, HP Laboratories. *Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators*. 5th International Workshop on Software and Compilers for Embedded Systems, March 2001.

[7] Synopsys, CoWare, Frontier Design. *SystemC Version 1.0 User's Guide*. 2000.

[8] A. Ghosh, J. Kunkel, S. Lao. *Hardware Synthesis from C/C++*. Proceedings of the DATE 99, Munich, Germany, 1999.

[9] G. De Micheli. *Hardware Synthesis from C/C++ Models*. Proceedings of the DATE 99, Munich, Germany, 1999.