Dix-septième colloque GRETSI, Vannes, 13-17 septembre 1999

Synthèse architecturale des systèmes asynchrones

Joseph O. DEDOU, Daniel CHILLET, Olivier SENTIEYS

LASTI - ENSSAT - Université de Rennes I,
6, rue de Kérampont, BP 447, 22305 Lannion, France,
Tel: +33 2-96-46-50-30; fax : +33 2-96-46-66-75
dedou@enssat.fr

Résumé – Grâce aux nombreux avantages qu'ils possèdent, notamment : absence de clock skew, performances moyennes, capacité à avoir une faible consommation etc..., les circuits asynchrones sont présentés comme une alternative aux circuits synchrones. Aussi, le développement de méthodologies de conception et d'outils de CAO deviennent une nécessité. Dans cet article, nous présentons une méthodologie de synthèse de haut niveau dédiée aux circuits asynchrones. Pour cela, nous redéfinissons les concepts classiques de la synthèse de haut niveau tout en conservant les différentes étapes de celle-ci. En effet, avec un fonctionnement du type événementiel, et des délais dépendant des données, cette redéfinition s'avérait nécessaire.

Abstract – With low power consumption, average case performance, no clock skew etc..., asynchronous circuits can be viewed as an alternative to synchronous circuits. Therefore, it's necessary to develop design methodologies and CAD tools for those circuits. In this paper, we propose a methodology for high level synthesis of asynchronous circuits. In this order, we redefine some classical concepts of the high level synthesis, without changing its different steps. Since, the operation delays are data dependent, and the circuits behavior is not predictable, this redefinition was necessary

1 Introduction

Depuis une décennie, de nombreux travaux ont été réalisés dans le cadre de la conception des circuits asynchrones. Les méthodologies proposées peuvent être reparties en deux grandes familles. D'un coté, nous avons les méthodologies qualifiées de synthèse logique, et de l'autre, les méthodologies dites de synthèse de haut niveau.

Les méthodologies dites de synthèse logique sont essentiellement tournées vers la conception des unités de contrôle asynchrones. Dans cette catégorie, on retrouve, les méthodes basées sur les graphes [14, 10, 12, 11], et celles qui emploient les machines d'états asynchrones [13, 6]. Les premières, utilisent comme formalisme de spécification le graphe des transitions de signaux (STG) proposé par Chu [4].

D'une manière générale, la synthèse basée sur le STG se divise en trois principales étapes. Dans un premier temps, on va chercher à déterminer le graphe accessible (Reachability Graph : RG) par une analyse flot de jetons. On déduit ensuite, à partir de ce graphe accessible, le graphe d'états en associant à chaque état un code binaire. À partir du graphe d'états obtenu, on établit pour chaque signal de sortie le tableau de Karnaugh à partir duquel on extrait les équations logiques nécessaires à l'implémentation du circuit en portes logiques.

La synthèse des machines d'états asynchrones suit la même démarche que dans le cas des machines synchrones, c'est à dire : minimisation puis assignation d'états et optimisation logique.

Les méthodes dites de synthèse de haut niveau [1, 15, 2] présentent les circuits asynchrones comme un ensemble de processus communiquant à travers des canaux de communication. Ces méthodes partent d'une spécification dans un langage de haut niveau du type CSP (Tangram, Occam), puis passent par différentes étapes de transformations qui peuvent être soit, des techniques algébriques, soit des méthodes de compilation, pour aboutir à une netlist de blocs asynchrones.

Ces méthodes ne sont pas réellement du domaine de la synthèse de haut niveau, et, mis à part un algorithme présenté dans [3], il y a très peu de recherche dans ce domaine. Dans cet article, nous proposons une méthodologie de synthèse de haut niveau dédiée aux circuits asynchrones. Dans ce cas particulier, il nous apparaît nécessaire de redéfinir certains concepts classiques de la synthèse de haut-niveau tout en conservant les différentes étapes de celle-ci :

- Modélisation temporelle des opérateurs
- Ordonnancement asynchrone (dynamique)
- Modèle d'architecture asynchrone (assignation différente)
- Interfaçage avec les outils de synthèse logique

2 Synthèse architecturale : une méthodologie

La synthèse d'architecture a pour objectif de générer de manière automatique la description structurelle d'une architecture à partir de la spécification comportementale d'une application dans un langage haut niveau. Elle est composée de quatre phases principales : sélection, allocation, ordonnancement et assignation [9].

Dans cet article, nous mettons l'accent sur l'ordonnancement et l'assignation. La modélisation temporelle des

opérateurs est présentée dans [7]. Elle se base sur la détermination puis la représentation statistique du temps d'un opérateur arithmétique (additionneur, multiplieur, ...) au travers d'un histogramme. Celui ci permet de déterminer le temps moyen de fonctionnement. Il a notamment été démontré que la distribution temporelle du temps de traversée, ainsi que sa moyenne, sont très éloignées du chemin critique utilisé systématiquement dans les systèmes synchrones (voir figure 1).

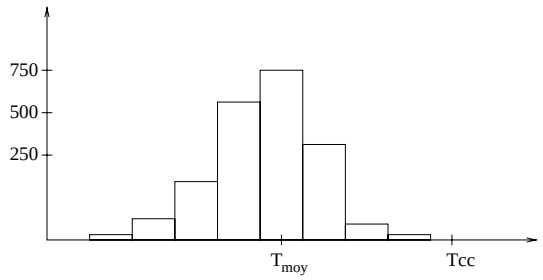


FIG. 1: *résultat de simulation d'un additionneur*

Par conséquent, un gain important sur le temps de fonctionnement doit être obtenu par le biais de l'asynchrone. Cette modélisation temporelle (temps moyen et distribution du temps de traversée) sera utilisée dans les différentes tâches de la synthèse architecturale. D'autre part, nous conservons au niveau algorithmique, une spécification comportementale séquentielle basée sur un langage classique *(C, Process VHDL, ...)* comme dans le cas des systèmes synchrones, et au niveau architectural, nous adoptons un modèle asynchrone. Le flot de conception que nous proposons est décrit à la figure 2.

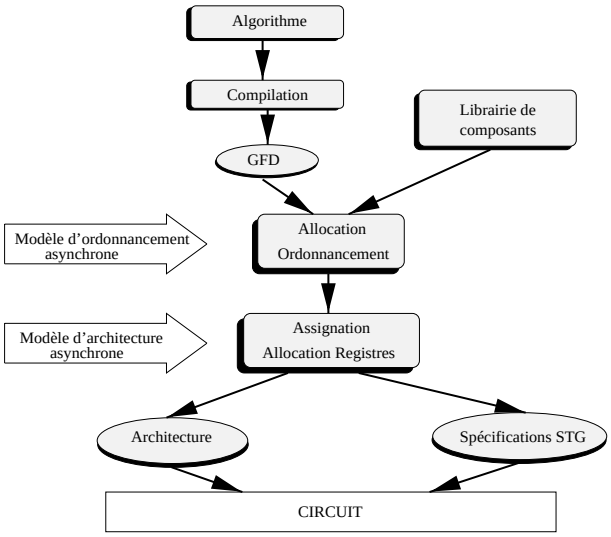


FIG. 2: *Flot de conception de la méthodologie développée*

2.1 Ordonnancement asynchrone

L'ordonnancement est l'affectation d'une date d'exécution à chaque opération de l'application. Dans le cas classique (systèmes synchrones), chaque date d'exécution correspond à un multiple du pas de contrôle (période de l'horloge). De ce point de vue, les systèmes asynchrones présentent deux différences significatives. En effet, le temps y est considéré comme une variable continue, et, le début et la fin d'une opération sont des événements qui peuvent se produire à tout instant. Par conséquent, dans ce cas particulier, en l'absence d'horloge, l'ordonnancement (asynchrone) ne doit pas être perçu comme le partitionnement des opérations à des temps discrets mais plutôt comme la définition d'un ordre partiel d'exécution des opérations. Ainsi le problème que nous cherchons à résoudre peut se résumer comme suit :

À partir d'une bibliothèque d'opérateurs matériels qui inclue les délais moyens δ_{moy} de ceux-ci, pour un graphe flot de données (GFD) G, et une contrainte de ressources connue, définir un ordre partiel d'exécution des différentes opérations du graphe, tout en minimisant au mieux l'estimation du délai moyen du système, afin de lui assurer un fonctionnement optimal.

Pour cela, nous allons dans un premier temps définir les termes suivants et établir par la suite des règles de priorités qui vont servir d'heuristiques pour l'élaboration de notre algorithme d'ordonnancement.
- $Start_{time}$: On le définit comme étant la date minimale à partir de laquelle une opération peut être exécutée.
- $Completion_{time}$: C'est la date à laquelle de fin d'exécution d'une opération donnée, elle vaut :

$$\mathbf{Completion_{time} = \delta_{moy} + Start_{time}}$$

avec δ_{moy} représentant le délai moyen d'exécution de cette opération par un opérateur donné.

Si on note pour un nœud N_i donné :
- S_{N_i} l'ensemble des nœuds successeurs de N_i
- P_{V_i} l'ensemble des nœuds prédécesseurs de N_i

On aura alors :

$$\mathbf{Start_{time}(N_i) = MAX_{N_j \in PV_i}\ Completion_{time}(N_j)}$$

Priorité-1 :
Elle a pour but de déterminer l'ordre dans lequel, les nœuds dits prêts (nœuds ordonnançables) doivent être ordonnancés. Pour cela il faut dans un premier temps, déterminer la longueur du chemin critique (L_{cc}) du GFD. On définit ensuite pour chaque nœud N_i sa profondeur $Prof(N_i)$ dans le graphe ; Elle sera déterminée à partir de l'expression suivante :

$$\mathbf{Prof(N_i) = L_{cc} - L_S(N_i)}$$

où $L_S(N_i)$ représente la longueur du chemin formé par les successeurs de V_i. Par la suite on décide d'accorder la priorité au nœud qui aura la plus petite profondeur parmi les nœuds ordonnançables (nœuds prêts).

Priorité-2 :
Elle doit permettre de choisir entre deux nœuds utilisant la même ressource et ayant la même priorité selon la priorité 1, celui qui doit être ordonnancé en premier. Dans ce cas on accordera la priorité au nœud ayant le plus petit $Start_{time}$.

De là, on établit l'algorithme d'ordonnancement dédié

aux systèmes asynchrones ci dessous.

1. Calculer la longueur du chemin critique L_{cc}, puis déterminer pour chaque nœud sa profondeur dans le graphe.
2. Pour chaque type d'opération (ex addition) établir la liste ordonnée des nœuds dits prêts (nœud ordonnançable) suivant les règles de priorités définies plus.
3. Pour chaque type d'opérations, choisir le premier nœud de la liste (nœud ayant la plus forte priorité) et l'ordonnancer c'est-à-dire trouver un opérateur disponible réalisant l'opération, et ce à partir du $Start_{time}$ du nœud pendant une durée au moins égale au délai moyen de l'opérateur, en d'autre termes un intervalle de longueur minimale δ_{moy} et de borne minimale au moins égale au $Start_{time}$ du nœud.

   Remarque: *S'il existe au moins deux opérateurs de disponibles, on portera le choix sur celui qui aura la plus petite borne minimale, et celle-ci sera le nouveau $Start_{time}$ du nœud ordonnancé.*
4. Réactualiser la liste des nœuds prêts. et reprendre à partir de l'étape 2.

Cet algorithme s'apparente à des techniques d'ordonnancement par liste, cependant nous utilisons des priorités ainsi que des modèles temporels spécifiques au concept des architectures asynchrones. Les résultats obtenus pour quelques algorithmes réguliers de traitement de signal (FFT, FIR etc ...) (tableau 1) permettent d'estimer le gain en terme de vitesse entre 45% et 65% avec une augmentation de la surface du même ordre. Toutefois, si on souhaite atteindre les mêmes performances avec des versions synchrones, il faudra utiliser plus de ressources que dans la version asynchrone, ce qui peut entraîner une augmentation de surface plus forte.

Algorithmes	ressources	Ordonnancement synchrone	Ordonnancement asynchrone (tps_{moy})
FIR-4	2 mult, 1 add	240 ns	100 ns
FFT	2 mult, 1 add, 1 sous	400 ns	260 ns
Ft elliptic	1 mult , 2 add	460 ns	370 ns
Eqt diff	2 mult, 1 addsous	380 ns	220 ns

TAB. 1: *Résultat ordonnancement synchrone et asynchrone*

2.2 Assignation

Elle consiste à affecter une opération à un opérateur, et doit en outre minimiser les problèmes de connectique interne à l'unité de traitement posés par la circulation des données. Dans les circuits asynchrones où les délais sont dépendants des données, il faut avoir une assignation dynamique afin d'optimiser l'utilisation des ressources disponibles et accroître la vitesse du système. De ce fait, on va définir la mobilité au niveau des ressources comme étant le nombre d'opérateurs (Nb_{op}) du même type. Partant de là, on établit que : si la mobilité est supérieure à un ($Nb_{op} > 1$), on aura une assignation dynamique, autrement, c'est-à-dire, si ($Nb_{op} = 1$) on aura une assignation statique comme dans le cas synchrone. Pour illustrer ce

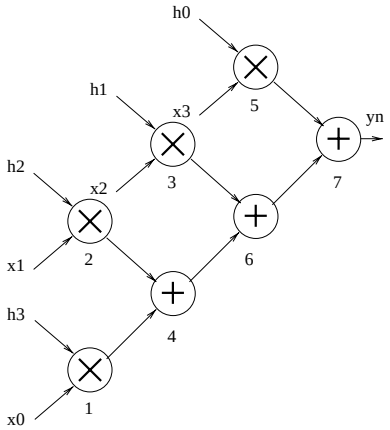


FIG. 3: *graphe flot de donnée du FIR-4*

que l'on vient de dire, nous reprenons l'exemple du FIR-4 présenté dans [8] (voir figure 3). Supposons que le délai nécessaire pour exécuter **l'opération 1** ($\mathbf{N_1}$) pour des entrées données soit supérieur au délai moyen du multiplieur (δ_{moy}) et le délai pour **l'opération 2** ($\mathbf{N_2}$) inférieur à δ_{moy} (NB : les délais sont dépendants des données). Tenant compte de ce que l'on a dit plus haut, **l'opération 3** ($\mathbf{N_3}$) peut être exécutée, soit par le *multiplieur 1*, soit par le *multiplieur 2*. Dans ce cas de figure, il serait donc souhaitable d'exécuter l'opération N_3 à l'aide du *multiplieur 2*. Cependant, celui-ci doit a priori exécuter **l'opération 5** ($\mathbf{N_5}$). Mais, si on regarde les priorités de ces deux opérations (N_3 a une priorité plus forte que N_5 ; $Prof(N_3) = 80ns$ et $Prof(N_5) = 90ns$), N_3 doit être traitée avant $\mathbf{N_4}$. Pour que cela soit possible, nous devons avoir une assignation dynamique. Celle-ci peut être obtenue à partir de l'architecture présentée à la figure 4.

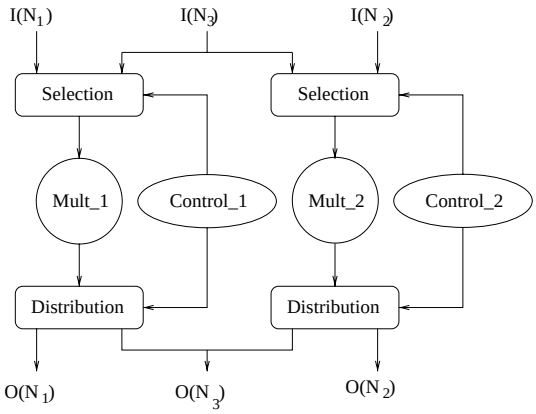
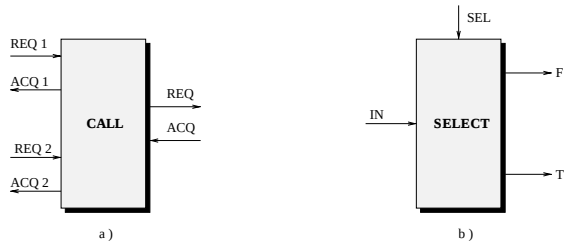


FIG. 4: *Assignation dynamique :FIR-4*

L'assignation dynamique peut être réaliser avec l'un des éléments de contrôle de base des circuits asynchrones tel que le module CALL ou encore le module SELECT (voir figure 5).



CALL: *suite à une requête sur une des deux entrées, une requête est transmise à la sortie et l'acquittement correspondant est renvoyé à l'entrée ayant provoqué la requête.*

SELECT: *envoie un signal sur une de ses deux sorties suivant l'état d'une entrée de contrôle.*

FIG. 5: *Quelques éléments asynchrones de contrôle*

3 Conclusion

Nous avons présenté dans cet article une méthodologie pour l'ordonnancement des circuits asynchrones. Elle est basée sur la connaissance des délais moyens des opérateurs de la bibliothèque contrairement aux circuits synchrones où elle est définie par rapport à la connaissance du délai pire-cas des opérations. La méthodologie présentée est intégrée dans l'environnement **BSS (Breizh Synthesis System** *http://archi.enssat.fr/bss)*. L'application de la méthode à divers algorithmes de traitement de signal montre que pour une même contrainte de ressources, un gain en terme de vitesse de l'ordre de 60 % peut être obtenu avec l'asynchronisme.

Nous avons aussi abordé la mise en œuvre d'une assignation spécifique à l'asynchrone, permettant de générer une architecture utilisant un ordonnancement statique ou dynamique. Elle permet en outre de tirer profit de la variation des délais des opérations du graphe. Ce travail est en cours de finalisation, avec en particulier la génération d'un modèle VHDL de simulation de l'architecture complète, et une interface avec les outils de synthèse PETRIFY [5]. D'autre part, pour réduire la complexité des interconnexions engendrée par l'assignation dynamique, il serait important d'introduire de nouvelles contraintes à ce niveau.

Références

[1] A.J.Martin. Programming in vlsi: From communicating process to delay-insentive circuits. Caltech-CS-TR-89-1, Departement of Computer Science, California Institut of Technologie, 1989.

[2] V. Akella and G Gopalakrishnan. SHILPA: A High-Level Synthesis System for Self-Timed Circuits. In *Int. Conf. on Computer-Aided Design (ICCAD)*, pages 587–591. IEEE Computer Society Press, November 1992.

[3] R.M. Badia and J. Cortadella. High-Level Synthesis of Asynchronous Degital Circuits: Scheduling Strategies. Technical report, Architectural of computer departement/ Catalunya Polytechnic University, November 1992.

[4] T.A Chu, C.K.C Leung, and T.S Wanuga. A Design Methodology for concurrent VLSI Systems. *Int. Conf. Computer Design (ICCD)*, 85:407–410, 1985.

[5] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transaction on Information and Systems*, E80-D(3):315–325, March 1997.

[6] A. Davis, B. Coates, and Stevens. Automatic synthesis of fast compast asynchronous control circuits. In *Asynchronous Design Methodologies, S. Furber and M. Edwards, Eds.*, volume A-28 of *IFIP Transactions*, pages 193–207. Elsevier Science Publishers, 1993.

[7] O. J. Dedou, D. Chillet, and O. Sentieys. Asynchronous Timing Model for High Level Synthesis for DSP Applications. In *SIGNAL PROCESSING IX: Theories and applications*, volume 1, pages 475–478. EUSIPCO-98, September 1998.

[8] Okito Dedou, Daniel Chillet, and Olivier Sentieys. Behavioral synthesis of asynchronous systems: a methodology. In *International Symposium on Circuits and Systems (ISCAS'99)*, June 1999.

[9] D. Gajski et al. *High-Level Synthesis - introduction to Chip and Systems Design.* kluwer Academic Publishers, 1992.

[10] M. Kishinevsky J.Cortadella, A. Kondratyev, L. Lavagno, and A. Yakovlev. Methodology and tools for state encoding in asynchronous circuit synthesis. In *Design Automation Conference*, pages 63–66, June 1996.

[11] L. Lavagno, K. Keutzer, and A.S. Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *28th ACM/IEEE Design Automation Conference*, pages 302–308, 1991.

[12] K.J. Lin and C.S. Lin. A realization algorithm of asynchronous circuits from STG. In *European Design Automation Conference (EDAC)*, pages 322–326. IEEE Computer Society Press, 1992.

[13] S Nowick. *Automatic synthesis of burst-mode asynchronous controllers.* PhD thesis, Stanford University, 1993.

[14] R. Puri and J Gu. Signal transition graph constraints for speed-independent circuit synthesis. In *Proc. International Symposium on Circuits and systems*, volume 3, pages 1686–1689. IEEE Computer Society Press, 1993.

[15] C van Berkel, J. Kessel, M. Roncken, R. Saeijs, and F. Schalij. The vlsi-programming langage tangram and its translating into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.