Synthèse logique de circuits asynchrones micropipeline

A. Rezzag

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|__/__/__/__/__/__/__/__/__/__|

T H E S E

pour obtenir le grade de

DOCTEUR DE l'INPG

Spécialité : Microélectronique

préparée au laboratoire **TIMA** dans le cadre de
l'Ecole Doctorale d'« Electronique, Electrotechnique, Automatique, Télécommunications, Signal »

présentée et soutenue publiquement

par

Amine Rezzag

Le 13 décembre 2004

Titre :

Synthèse Logique de Circuits Asynchrones Micropipeline

Directeur de Thèse : Marc Renaudin
Codirecteur : Laurent Fesquet

JURY

M. Pierre Gentil,	Président
M. Amara Amara,	Rapporteur
M. Olivier Sentieys,	Rapporteur
M. Marc Renaudin,	Directeur
M. Laurent Fesquet,	Codirecteur

RESUME

Les circuits asynchrones ont des caractéristiques qui les démarquent nettement des circuits synchrones : modularité quasi-parfaite, absence d'horloge, contrôle local. Ils tendent à constituer une sérieuse alternative pour pallier aux problèmes posés par l'intégration en silicium d'applications de plus en plus complexes. Le goulot d'étranglement principal pour l'adoption de la conception des circuits asynchrones se situe au niveau du manque de méthodologies et d'outils puissants pour ce type de conception. Ce travail de thèse porte sur la définition d'une méthodologie de conception de circuits intégrés asynchrones micropipeline. La synthèse micropipeline est une approche qui exploite à la fois les outils commerciaux de synthèse pour le chemin de données, et la synthèse de contrôleurs asynchrones pour le contrôle (« STG » avec Petrify , « BURST MODE » avec Minimalist).

La méthodologie générale pour la modélisation et la synthèse de circuits asynchrones est basée sur la spécification dite DTL (*Data Transfer Level*) qui définit une façon d'écrire les codes sources garantissant une synthèse rapide et systématique pouvant cibler plusieurs styles de circuits asynchrones. Cette méthode de conception part d'une spécification basée sur un langage de haut niveau (CHP ou *Concurrent Hardware Processes*). Elle permet en sortie de générer des circuits en portes logiques élémentaires et en portes de Muller. Il a été procédé à un prototypage de cette méthode de synthèse. Ce prototype est conçu pour être intégré dans l'outil de conception automatique de circuits asynchrones TAST (*Tima Asynchronous Synthesis Tool*) dont le synthétiseur génère des circuits asynchrones QDI, pour l'étendre à la génération de circuit micropipelines. Par ailleurs, la méthodologie de synthèse a été étendue à l'utilisation de différents types de contrôleurs asynchrones susceptibles d'en améliorer les performances en termes de vitesse et de consommation.

MOTS-CLES

Circuits asynchrones, méthodologie de conception, synthèse de circuits asynchrones, circuits micropipeline, synthèse de contrôleurs asynchrones, processus concurrents communicants, réseau de Pétri, protocoles de communications, langage CHP, graphes de flot de données, équations de dépendances, outils de synthèse.

ABSTRACT

The inherent asynchronous circuit features (modularity, clockless system, local control) brings a serious alternative to face the problems encountered by the silicon integration of more and more complex applications. The main bottleneck to adopt the asynchronous logic is due to the lack of methodologies and efficient tools for this kind of design. The thesis works aim to define a micropipeline asynchronous design methodology. The micropipeline synthesis approach use both commercial tools for data path synthesis and specific tools for asynchronous control synthesis (« STG » using Petrify, « BURST MODE » using Minimalist).

The overall methodology for the modelling and the synthesis of asynchronous circuits is based on the DTL specification (Data Transfer Level) which assumes a restriction of source code allowing a rapid and systematic synthesis and targeting several kinds of asynchronous circuits. This design methodology starts from a high level programming language named CHP (Concurrent Hardware Processes) and generates a gate netlist composed of elementary logic and Muller gates. This synthesis methodology has been prototyped. This prototype has been designed for its integration in the TAST automatic asynchronous design flow (Tima Asynchronous Synthesis Tool) which generate QDI circuits, to spread it in the generation of micropipelines circuits. Furthermore, the synthesis methodology has been extended for different kinds of asynchronous controller to improve performances such as speed and energy consumption.

KEYWORDS

Asynchronous circuits, Design Methodology, Asynchronous circuits synthesis, Micropipeline circuits, Asynchronous Controlers synthesis, Concurrent communicating processes, Pétri nets, Communication protocols, CHP langage, Data flow graphs, Dépendancies equations, Synthesis tools.

A ceux qui m'ont donné la vie, m'ont bercé, m'ont éduqué, et m'ont montré le chemin,

A mes parents.

A celles à qui je dois tant,

A mes sœurs.

A la mémoire de mes Grands-Parents Mohamed et Yamina.

A mes Grands-Parents Slimane et Meriem.

A Amé Mohamed.

A mon frère Nabil et ses enfants.

A la mémoire de Moez Ouni.

Remerciements

La présente thèse a été effectuée au laboratoire TIMA (*Techniques de l'Informatique et de la Microélectronique pour l'Architecture des Ordinateurs*) sur le site Viallet de l'*Institut Polytechnique National de Grenoble* (INPG). Je remercie M. Bernard Courtois directeur de ce laboratoire pour m'y avoir accueilli.

Je remercie mon directeur de thèse M. Marc Renaudin Professeur à l'*Ecole Nationale d'Electronique et de Radioélectricité de Grenoble* (ENSERG) pour m'avoir accueilli au sein de son groupe de recherche CIS (*Concurrent Integrated Systems*). Je lui suis particulièrement reconnaissant pour son support technique pertinent et fort précieux, pour sa confiance et pour ses encouragements.

Je témoigne ma gratitude à M. Laurent Fesquet, Maître de conférence ENSERG pour sa grande disponibilité, pour les éclairages techniques qu'il m'a prodigué, pour son soutien constant et déterminant et pour l'ensemble de ses qualités humaines.

Je remercie M. Amara Amara, Professeur à l'*Institut Supérieur d'Electronique de Paris* (ISEP), et M. Olivier Sentieys, Professeur à l'*Ecole Nationale Supérieure des Sciences Appliquées et de Technologie de Lannion* (ENSSAT), pour avoir eu l'amabilité d'accepter de rapporter mes travaux de thèse, et pour la qualité de leurs analyses. J'adresse également mes remerciements à M. Pierre Gentil, Professeur à l'INPG pour m'avoir fait l'honneur de présider le jury de cette thèse.

Mes remerciements vont également à tous les membres de TIMA, CMP (*Circuits Multi-Projets*), et CIME (*Centre Inter-Universitaire de Microélectronique*) : Professeurs, Maitres de conférence, doctorants, administrateurs réseaux, et secrétaires, pour leur aide et leur assistance.

Toute ma gratitude à M. Alexandre Chagoya, à Kholdoun Torki, et à Hubert Delori pour leur disponibilité et leur gentillesse.

Je tiens bien entendu à remercier chaleureusement tous les membres de l'équipe CIS pour leur sympathie et leur convivialité permanentes, et pour leur penchant agréable à organiser par-ci par-là des réunions culinaires toujours délicieuses. Je cite par ordre alphabétique, Alain, An Vuh (le soleil se lève bien à l'est), Antoine, Arnaud 1, Arnaud 2 (cœur à prendre), Aurélien, Bertrand (c'est pour quand la F1 ?), Cédric, David, Dhanistha (la bonne humeur c'est par là), Estelle, Fabien (riro-thérapeute), Fraidy, Gautier, Gilles (merci pour le clin d'œil), Isabelle, JB (ça roule toujours ?), Jérôme (tout va bien ?), Joao et Marianna (nos brésiliens préférés), Livier, Manu (c'est lourd ?), Mohamed (à ta santé), Phillipe, Robin (personnage de légende), Salim (Evian c'est bien une eau de source ?), Sophie, Thibaut, Vivien, Yann (à la réalisation), et Yannick.

Ma profonde reconnaissance à Amine, Dan, Farid et Malek (mes amis de toujours), Kamel (mon fidèle alter ego), Karim (mes hommages), Menouer (ma révérence), Mohamed et Lamia, Mustapha (c'est quoi ton prochain article ?), Nacer (eureka !), Nabil et ses enfants, Nadir et Rezika (mes sportifs préférés), Rachid et Karim (mes autres amis de toujours), Tarik et Hana, Tarik Kannat, Mr et Mme Weber, Zakou et Faty (mille mercis).

Mes remerciements à Ahcène, Amel, Amer, Eric, Diana, Emile, Ghiath, Greg, Kheirredine, Latif, Samy, Sébastien et Sylvie.

Que celles et ceux que j'aurais oubliés ici me pardonnent.

Enfin je ne remercierais jamais assez mes chers parents ainsi que mes sœurs pour l'amour qu'ils me témoignent chaque jour et pour la confiance qu'ils m'accordent. Ma pensée va également à l'ensemble de ma famille, petits et grands, ainsi qu'à tous mes proches.

Table des matières

INTRODUCTION

Dans le dessein d'obtenir des systèmes plus performants, compacts, et consommant un minimum de puissance, les niveaux d'intégration dans l'industrie des circuits intégrés sont de plus en plus élevés et permettent de la sorte d'intégrer des applications complexes sur une seule puce. Une conséquence de cet état de fait est que la logique synchrone doit alors gérer un certain nombre d'obstacles majeurs tels la distribution d'horloge, la consommation, le bruit, ou encore la modularité. Face à ces contraintes, la logique asynchrone gagne en popularité et semble constituer une alternative. En effet, la modularité quasi-parfaite des circuits asynchrones assure une « réutilisabilité » et une intégration aisées dans les systèmes complexes. Par ailleurs, l'absence d'horloge et l'autonomie du contrôle local facilitent l'utilisation de blocs asynchrones dans ces mêmes circuits.

De ce fait, l'industrie du semi-conducteur ne cache pas son intérêt pour le domaine, tel que le montre les travaux de recherche et de veille réalisés par les sociétés de service de conception de circuits « Cirrus Logic » ou « LSI Logic » ou des équipementiers « Philips » ou « Infinéon » pour traquer les techniques faible consommation. Les grands groupes américains tels qu'Intel, Sun ou HP travaillent via la conception asynchrone à améliorer les performances de leurs réalisations synchrones en termes de vitesse et de consommation. Les fondeurs y voient un moyen pour tenter de valoriser les technologies toujours plus performantes en échappant aux contraintes imposées par l'horloge. Il s'agit donc aujourd'hui d'une technologie émergente dont on peut difficilement assurer le succès industriel malgré les atouts cités plus haut.

En effet, le goulot d'étranglement pour l'adoption de la conception des circuits asynchrones se situe principalement au niveau du manque de méthodologies et d'outils puissants pour ce type de conception, même si des progrès importants ont été fait dans les domaines de la synthèse automatique, la testabilité, l'analyse du timing et la vérification. Contrairement à la conception synchrone qui a atteint un seuil de maturité et d'automatisation tel que le secteur industriel n'est pas prêt à un retour en arrière en terme de méthodologies (outils) et de productivité, la conception asynchrone reste encore le domaine de spécialistes utilisant des outils faiblement automatisés. De ce fait la diffusion de la logique asynchrone dans le secteur industriel ne pourra être effective qu'à la condition que des outils du niveau d'automatisation des outils disponibles en synchrone soient développés et que des ingénieurs soient formés à leur utilisation.

Il existe aujourd'hui quelques outils de conception dévolus à la synthèse de circuits asynchrones mais aucun n'est commercialisé. Philips est le seul industriel à posséder un environnement de conception complet [VAN 93] pour les circuits asynchrones mais il reste d'un usage privé. D'autres industriels ont développé des outils pour répondre à des développements ponctuels dans le domaine [MAR 94], [SPR 94], [CHU 93]. Les laboratoires universitaires possèdent des outils de haut niveau [BAR 00], [MAR 90b], [CORT 02], [FUH 99] mais qui souffrent d'une faible compatibilité avec les environnements commerciaux et qui trop souvent ne couvrent pas l'ensemble du cycle de conception. Leur usage permet cependant à la communauté universitaire de concevoir des circuits fonctionnels.

Dans le passé la communauté asynchrone a majoritairement porté ses efforts sur le développement d'outils pour la conception de contrôleurs. Ces outils peuvent être qualifiés de bas niveau car ils manipulent des fonctions logiques et sont focalisés sur la suppression d'aléas (notons à ce niveau que la façon de traiter les aléas est l'une des différences majeures entre la conception de circuits synchrones et asynchrones). Les outils qui adoptent les graphes (CD, STG, Burst Mode) comme formalisme de spécification rentrent dans ce cadre. Ils

permettent la réalisation efficace de machines à états ou contrôleurs de faible taille, mais sont par contre inefficaces pour concevoir des systèmes et architectures complexes.

Ces dernières années la communauté s'est davantage tournée vers les approches basées langages. Les précurseurs dans ce domaine sont Alain Martin (langage CHP, [MAR 93]), Kees Van Berkel (langage Tangram, [KES 00]), Erik Brundvand (langage Occam, [BRU 89]), et Venkatech Akella (système Shilpa [AKE 92][GOP 92]). Egalement, l'université de Manchester a développé le langage Balsa [BAR 00]. Par ailleurs, fréquents sont les travaux qui tentent de faire le lien entre les langages de description de matériel et les outils de synthèse logique asynchrones tel le travail réalisé par Blunno (langage Verilog, [BLU 00]).

Dans ce contexte, le travail de cette présente thèse consiste à concevoir une méthodologie de synthèse basée sur un langage de haut niveau et de développer un outil de CAO traduisant cette méthodologie. Cet outil s'intègre dans le projet TAST acronyme de « *Tima Asynchronous Synthesis Tools* », qui est un environnement de conception dédié à la synthèse de circuits asynchrones. Ce projet est développé au sein du groupe de recherche CIS (*Concurrent Integrated Systems*) du Professeur Marc Renaudin au laboratoire TIMA.

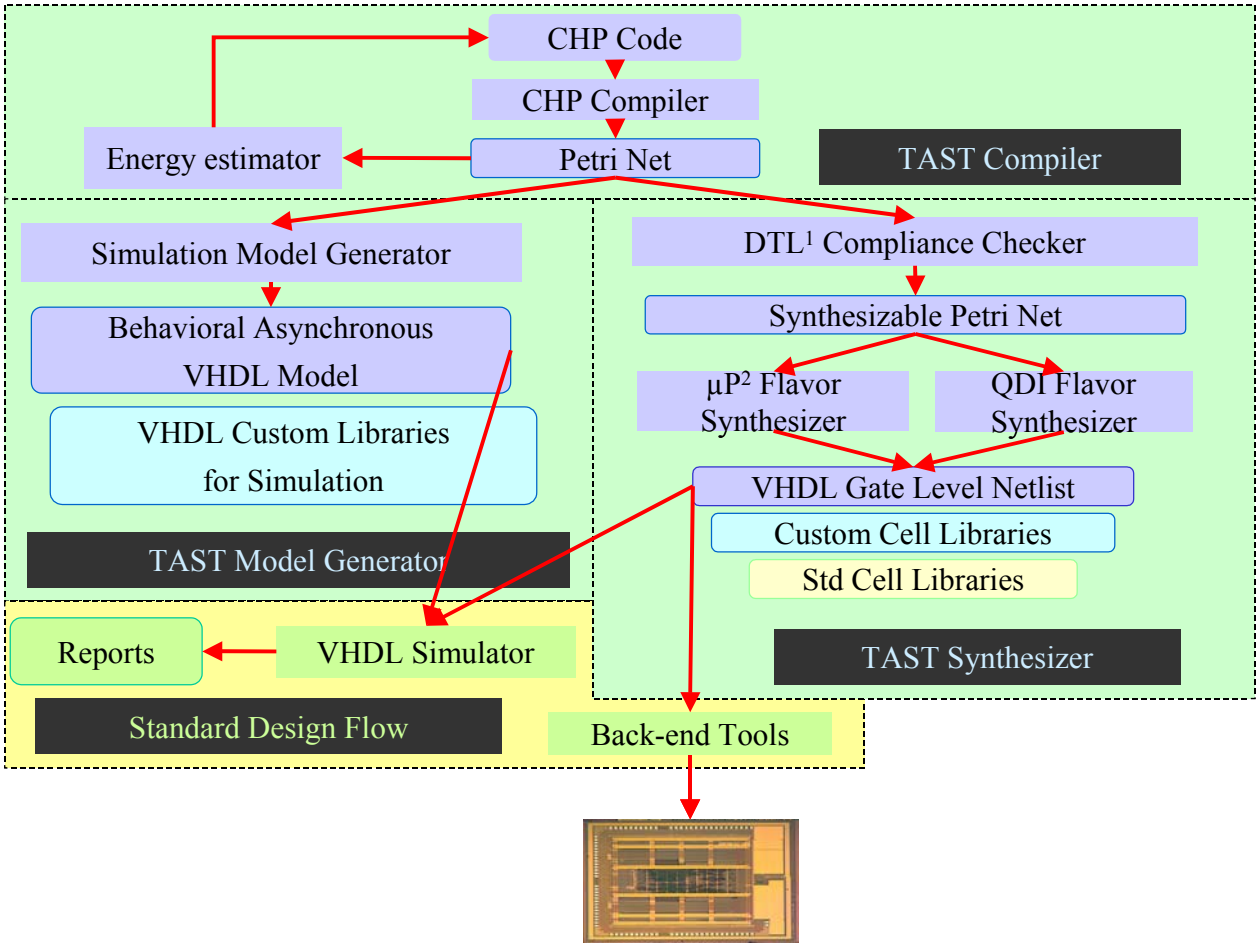


Figure I.1 Flot de TAST

L'outil TAST considère une approche multi-sources (perspective de plusieurs langages), multi-cibles et est construit autour d'un format intermédiaire basé sur une combinaison réseaux de Petri/graphes de flot de données. Il considère en entrée des descriptions de circuits asynchrones en langage haut niveau CHP (*Communicating Hardware Processes*) étendu et permet de générer différents styles de circuits asynchrones micropipeline ou QDI et différents formats de sorties (description comportementale en VHDL, langage C, description matérielle au niveau porte en VHDL).

Le *front-end* de l'outil TAST est un compilateur qui réalise la compilation d'un code source CHP décrivant un système asynchrone pour le transformer en une représentation intermédiaire combinaison de réseaux de Petri et de graphes de flots de données. Cette représentation intermédiaire a pour objectif de découpler le *front-end* (multi-sources) du *back-end* (multi-cibles). Le langage CHP permet de décrire des circuits à base de processus concurrents communicants et l'on considère dans le cadre de cette thèse une version enrichie de celle initialement développée par Alain Martin à Caltech. Le réseau de Petri offre une structure de choix mixte (séquentielle et parallèle) et le graphe de flot de données exprime les dépendances de données.

Le *back-end* de l'outil TAST considère à son entrée le format intermédiaire généré par le *front-end* et engendre plusieurs modèles de sortie visant soit la synthèse soit la simulation. La synthèse est basée sur une spécification dite DTL (*Data Transfer Level*) qui représente un ensemble de règles garantissant la « synthétisabilité » des circuits asynchrones décrit à l'entrée de l'outil. L'outil TAST permet de cibler deux types de circuits asynchrones, QDI (*Quasi Delay Insensitive*) ou Micropipeline. La synthèse des circuits micropipeline sépare la partie opérative du circuit de sa partie contrôle : le contrôle est synthétisé et donne un circuit asynchrone logique niveau porte et le flot de donnée est implémenté en VHDL-RTL pour ensuite être synthétisé par un outil commercial synchrone. De cette manière on tire profit des outils de synthèse commerciaux. La synthèse ciblant les circuits QDI génère quand à elle une *netlist* VHDL autant pour le contrôle que pour le flot de données. Cela est rendu possible par le fait que les canaux de communication utilisent un codage insensible aux délais (multi-rails). Par ailleurs, en vue de la simulation le *back-end* de l'outil TAST génère également un modèle comportemental équivalent en VHDL afin de pouvoir vérifier par simulation la correction fonctionnelle du circuit à l'aide d'outils industriels standards. Un autre modèle équivalent en langage C est généré pour une validation accélérée sous Unix/Linux de la spécification d'entrée CHP.

Plan du manuscrit

Le chapitre premier se veut un état de l'art des méthodologies et outils de synthèse asynchrones. Il commence par tenter d'exprimer la motivation de l'intérêt porté à la logique asynchrone en présentant ses avantages et ses inconvénients vis-à-vis de la logique synchrone. Un certain nombre de concepts de base sont ensuite déclinés pour permettre de comprendre le point essentiel du mode de fonctionnement asynchrone basé sur le contrôle local : un opérateur asynchrone est assimilé à une cellule réalisant une fonction et communiquant avec son environnement au travers de canaux de communication. Le point clé est que ces canaux permettent d'échanger aussi bien des données que des informations de synchronisation grâce au codage des données et au protocole de communication de type « poignée de main » choisis. L'implémentation de ce protocole nécessite l'introduction de la notion de porte de Muller. Par ailleurs le mécanisme de communication asynchrone fait que toute transition d'un signal peut être interprétée comme un événement porteur d'information. C'est pourquoi il est

primordial de concevoir une logique exempte d'aléas Après cela, nous présentons une classification des circuits asynchrones basée sur les modèles qui régissent le comportement des délais des circuits et des environnements. Enfin nous faisons un état de l'art des méthodologies de spécification de circuits asynchrones et des outils de synthèse correspondants. A ce niveau, deux grands ensembles se dégagent, à savoir les méthodologies de spécification basées sur les graphes et leur synthèse logique, et les méthodologies de spécification basées sur les langages de haut niveau et leur synthèse dirigée par la syntaxe ou par compilations successives.

Le chapitre second introduit la spécification des circuits asynchrones que nous synthétisons, le format intermédiaire qui permet de découpler le front-end du back-end de l'outil, et la spécification DTL définie pour identifier les circuits synthétisables en amont de la procédure de synthèse. Les circuits à l'entrée de l'outil sont donc décrits en langage CHP. C'est un langage de description de haut niveau basé sur les processus communicants. La représentation intermédiaire est basée sur une combinaison de réseaux de Petri exprimant le contrôle du circuit et les graphes de flot de données exprimant les dépendances des données. La spécification DTL est essentiellement basée sur l'identification des éléments de mémoire des circuits [DIN02]. Si la description CHP du circuit à l'entrée de l'outil n'est pas conforme à la spécification DTL, le circuit ne peut être synthétisé. Dans ce cas la description est transformée pour la rendre conforme DTL.

Dans le chapitre trois nous nous projetons sur l'architecture des circuits que nous désirons générer et nous présentons les modèles de circuits cibles après avoir expliqué la technique du pipeline asynchrone et ses caractéristiques essentielles (capacité mémoire, débit, latence et temps de cycle). Le modèle de pipeline « données groupées » ou micropipeline que nous traitons dans le cadre de cette thèse n'est pas le plus robuste mais présente l'avantage d'être moins gros en surface, de consommer moins que les autres modèles, et de tirer profit des techniques de conception synchrone (le flot de donnée est synthétisable par les outils commerciaux synchrones). Puis nous déclinons les structures linéaires et non linéaires des circuits cibles en travaillant sur quatre protocoles de communication : séquentiel, WCHB, PCHB et PCFB.

Le chapitre quatre brosse l'ensemble des opérations préalables à la synthèse. Nous expliquons le passage d'une description en langage de haut niveau CHP à la combinaison réseau de Petri / Graphe de flot de données (*Petri Net - Data Flow Graph* ou PN-DFG) traduisant la représentation intermédiaire. Ensuite nous vérifions la conformité de l'ensemble des réseaux de Petri à la spécification DTL. Les réseaux de Petri sont rendus exempts des structures de choix ou sélection pour faciliter la suite de la procédure de synthèse.

Dans le chapitre cinq nous générons des équations dites de dépendances à partir des réseaux de Petri. Elles décrivent les sorties du circuit en fonction des entrées de celui-ci. Le point clé est que ces équations sont d'abord indépendantes du protocole de communication et de la technologie cible. A ce niveau nous scindons la synthèse en deux parties contrôle et chemin de données. Pour la partie contrôle nous introduisons la synthèse de bas niveau en considérant le protocole de communication et le modèle de circuit cible choisis. Nous obtenons une *netlist* de portes décrivant le circuit sur la base d'une bibliothèque de cellules génériques. Cette *netlist* est optimisée avant d'être « projetée » (ou « *mappée* ») sur la base d'une bibliothèque de cellules standards. Les techniques d'optimisation, de décomposition et de projection technologique doivent veiller à ce que le circuit reste à chaque étape exempt d'aléa et à assurer le respect des fourches isochrones.

CHAPITRE I

1. Etat de l'Art sur les Circuits Asynchrones et leurs Synthèses

1.1 Introduction

Deux hypothèses fondamentales facilitent grandement la conception des circuits intégrés logiques synchrones. La première concerne la binarisation des signaux manipulés qui permet une implémentation électrique simple et offre un cadre de conception maîtrisé grâce à l'Algèbre de Boole. La seconde hypothèse a trait à la discrétisation du temps et permet quant à elle de s'affranchir des problèmes de rétro-actions et/ou de boucles combinatoires ainsi que des fluctuations électriques transitoires.

Dans la classe des circuits asynchrones on conserve la politique de codage discret des signaux (binaire la plupart du temps), mais l'on ne se soucie plus de la discrétisation temporelle. En effet, ces circuits ne sont pas commandés par un signal d'horloge périodique. Ils définissent une classe de circuits nettement plus large car leur contrôle peut être assuré par tout autre moyen que celui d'une horloge globale (comme c'est le cas en synchrone). En l'occurrence, le transfert d'information est géré localement par une signalisation bidirectionnelle adéquate en aval et en amont de l'opérateur local concerné ce qui garantit la synchronisation et la causalité des événements au niveau local et la correction fonctionnelle du système dans son ensemble.

Cette caractéristique des circuits asynchrones qui consiste à ne plus considérer l'horloge constitue un avantage de plus en plus pertinent. A titre illustratif, les processeurs modernes ont un chemin critique de moins de 1 ns. Un déphasage d'horloge de plusieurs centaines de pico-secondes est inacceptable puisqu'ils représenterait quelques dizaines de pourcent du temps critique.

D'autres conséquences heureuses inhérentes à l'absence de la contrainte d'horloge concourent à définir l'intérêt grandissant porté aux circuits asynchrones.

- Ignorer le temps de propagation rend possible la conception et l'utilisation des opérateurs dont le temps de calcul est variable. Cela libère le concepteur de l'approche « pire cas » des circuits synchrones. Celle-ci consiste à toujours dimensionner le circuit par rapport au temps de propagation le plus long, et permet ainsi un *calcul en temps moyen*.

- Le contrôle local fait que dans la technique du pipeline, qui s'applique également au cas asynchrone, le nombre de données présentes dans le pipeline n'est plus imposé. En effet, les données progressent dans le pipeline aussi longtemps qu'elles ne rencontrent pas de ressources occupées et ceci indépendamment les unes des autres. En synchrone l'horloge impose une synchronisation relative forte des données entre elles.

- En synchrone, le traitement d'événements non déterministes, tels que l'apparition d'un signal d'interruption dans un microprocesseur (signal asynchrone) peut faire apparaître, à cause de l'échantillonnage de ce signal, un état métastable dont la durée est indéterminée, non bornée [KLE 87]. Or les circuits synchrones imposent à tous les signaux un temps d'établissement borné compatible avec la période de l'horloge et ne permettent donc pas d'assurer un comportement correct dans 100% des cas. En asynchrone par contre, ce problème est annihilé puisque l'on peut attendre autant que nécessaire la fin de l'état métastable. En ce sens, les circuits asynchrones sont un support fiable pour les traitements non déterministes.

- La modularité des circuits et systèmes asynchrones, possible grâce la localité du contrôle, est quasi-parfaite. Cela rend facile la construction d'une fonction complexe en connectant des modules préexistants [ROB 97a][ROB 97b][REN 00a]. Cette propriété est particulièrement intéressante lorsqu'on souhaite favoriser la réutilisation de blocs dans une entreprise ou plus généralement l'échange de propriétés intellectuelles (IPs).

- La synchronisation locale atemporelle des circuits asynchrones fait que le contrôle est distribué dans toute la structure. Par conséquent, les problèmes de pics de consommation des circuits synchrones sont inexistants en asynchrone puisque l'activité électrique est mieux répartie dans le temps que celle d'un circuit synchrone.

- Le terme « asynchrone » n'est pas synonyme de « faible consommation » mais plusieurs propriétés des circuits asynchrones peuvent servir la réduction de la consommation :

  - l'énergie dissipée par le système de distribution des horloges n'existe pas dans les circuits asynchrones ;
  - le mode de fonctionnement asynchrone offre de façon naturelle et implicite une mise en veille (tout opérateur non sollicité est au repos) à tous les niveaux de granularité ;
  - l'activité électrique est mieux répartie dans le temps que celle d'un circuit synchrone grâce à la synchronisation locale atemporelle des circuits asynchrones ;
  - réduire la tension d'alimentation des circuits est souvent utilisée pour limiter la puissance consommée (la puissance varie avec le carré de la tension). L'aptitude des circuits asynchrones à être fonctionnels indépendamment des temps de traversée des opérateurs élémentaires, permet de réduire la tension d'alimentation avec un matériel minimum.

- Une conséquence de cette propriété du courant consommé est que la puissance des ondes électromagnétiques émises par les circuits asynchrones est plus faible que celle émise par les circuits synchrones. Cette propriété de faible bruit est à l'origine de la commercialisation par Philips du premier circuit asynchrone, un processeur 8 bit compatible 80c51 [GAG 98].

- Les circuits asynchrones se prêtent facilement aux différentes migrations technologiques (changements de styles de conception, circuits semi-dédiés, dédiés, ou encore changements d'architectures pour une fonction donnée) et les rendent aussi plus « souples ».

Toutefois, l'ensemble de ces avantages et de ces gains potentiels apportés ne signifie pas que la conception des circuits asynchrones est exempte de contraintes. Le problème majeur est lié au phénomène d'aléas [UNG 69]. En logique synchrone, le fait que le temps soit discret permet d'ignorer les aléas liés aux phénomènes transitoires, alors qu'en asynchrone, l'absence d'horloge globale fait qu'un circuit peut répondre aux transitions de ses entrées à n'importe quel instant. Toute transition d'un signal est susceptible d'être interprétée comme un événement porteur d'information. En conséquence, tous les types d'aléas sont susceptibles d'engendrer un mauvais fonctionnement du circuit et les problèmes inhérents possibles sont :

- les méthodes de conception de circuits asynchrones doivent garantir une implémentation sans aléa et assurer la correction du fonctionnement global ;

- on restreint l'environnement à un mode de fonctionnement fondamental (un seul changement des entrées du circuit à la fois) parce que les techniques éliminant les aléas pour

le mode fondamental sont bien connues et plus simples que celle pour le mode où plusieurs changements sont permis [UNG 69] ;

- dans un système, la communication avec les circuits synchrones nécessite l'implémentation d'interfaces spécifiques ;

- les performances des circuits asynchrones peuvent être affaiblies en pratique lorsque l'on rajoute des éléments de retard souvent utilisés pour s'affranchir des aléas. Cette politique garantit la correction de fonctionnement mais diminue la performance potentielle du circuit asynchrone ;

- les outils de CAO synchrones largement répandus et utilisés ne sont pas appropriés pour les circuits asynchrones. Ce point constitue l'un des freins majeurs à la démocratisation de la conception asynchrone.

Globalement, les circuits asynchrones constituent une voie qu'il est important d'explorer pour apporter des solutions aux problèmes que posent déjà aujourd'hui et que posera dans le futur l'intégration sur silicium d'applications de plus en plus complexes. Les potentiels des technologies silicium ne cessent de croître et la gestion de la complexité devient de plus en plus aiguë (« *Design Gap* »), tant au niveau de la spécification, de la description et la validation des systèmes, que de leur implémentation matérielle. Les enjeux se situent donc aussi bien au niveau des méthodes de conception et des outils, que des techniques de réalisation.

Intéressons nous à présent aux concepts de base permettant de mieux cerner les enjeux et notamment les principales méthodologies de conception de circuits asynchrones.

1.2 Principes de fonctionnement

1.2.1 Mode de fonctionnement asynchrone

Le terme « Asynchrone » signifie qu'il n'existe pas de relation temporelle a priori entre les objets qu'il qualifie. Dans le contexte de la conception de systèmes intégrés, ces objets sont des événements au sens large (contrôle ou données) implémentés par des signaux électriques.

Ainsi, un signal d'interruption appliqué à un microprocesseur est qualifié d'asynchrone par rapport au fonctionnement du microprocesseur et indépendamment du caractère synchrone ou asynchrone de ce dernier.

Par ailleurs lorsque l'on parle de circuits asynchrones, on parle le plus souvent de circuits qui savent gérer des signaux qui sont asynchrones entre eux, mais dont le comportement est parfaitement déterminé. La spécification d'un circuit asynchrone est telle qu'il n'existe pas de relation de causalité a priori entre les évènements, ils sont asynchrones. Cependant, il est garanti qu'il y aura de façon certaine un événement sur l'un et l'autre de ces signaux. Dans ces conditions, il y a indétermination sur les instants d'occurrence des évènements, mais le fait que les évènements aient lieu est absolument certain et déterminé.

Ainsi, les systèmes dits « asynchrones » fonctionnent avec la seule certitude de l'occurrence des événements, sans connaissance des instants ni de l'ordre des occurrences. Le fonctionnement est similaire à celui des systèmes flot de données. Il suffit de décrire

l'enchaînement des événements et des opérations sous la forme d'un graphe de dépendances (réseau de Pétri par exemple). L'évolution globale du système est garantie par l'évolution conjointe et éventuellement concurrente des différents éléments qui le composent. Chaque élément évolue avec les seules « informations » reçues des éléments avec lesquels il est « connecté ». On entend par "informations reçues" des variables ou des événements de synchronisation. Les règles d'activation sont similaires à celles des réseaux de Petri. Il n'est donc pas nécessaire d'introduire de mécanisme global d'activation du système (l'analogie est aussi très forte avec le modèle des processus séquentiels communicants [HOA 78]).

Et c'est là que se situe la différence avec les systèmes synchrones pour lesquels le signal d'horloge joue le rôle d'un actionneur global. Tous les éléments du système évoluent ensemble lors de l'occurrence d'un événement horloge, l'exécution de tous les éléments est donc synchronisée. Ce mécanisme global d'activation introduit une contrainte qui est d'ordre temporel. Tous les éléments doivent respecter un temps d'exécution maximum fixé par la fréquence des occurrences des événements d'activation (condition de bon fonctionnement). Tous les éléments du système sont donc synchronisés dans le temps.

A l'opposé, les systèmes asynchrones évoluent de façon localement synchronisée et le déclenchement des actions dépend uniquement de la présence des données à traiter. La correction fonctionnelle peut ainsi être indépendante de la durée d'exécution des éléments du système. On parle de circuits et systèmes insensibles aux délais.

1.2.2 Caractéristiques d'un opérateur asynchrone

D'un point de vue externe, un opérateur asynchrone peut être considéré comme une cellule réalisant une certaine fonction et communiquant avec son environnement à travers des canaux de communication banalisés. Le point clé est que ces canaux de communication servent à échanger avec l'environnement aussi bien des données que des informations de synchronisation (parfois les deux simultanément).

L'opérateur peut indifféremment réaliser une fonction au niveau bit, au niveau arithmétique ou même implémenter un algorithme complexe, en utilisant toujours la même sémantique pour les canaux de communication. De même, le protocole d'échange étant respecté, l'opérateur peut être combinatoire ou à mémoire.

Un opérateur asynchrone peut être caractérisé par au moins quatre paramètres fondamentaux.

- **Un temps de latence**, qui correspond à la chaîne combinatoire la plus longue nécessaire à une sortie pour être l'image fonctionnelle d'une entrée. Cette caractéristique ne dépend pas de la mémoire de l'opérateur, car en asynchrone un opérateur de mémorisation inoccupé ou vide se comporte comme une cellule combinatoire qui laisse simplement la donnée le traverser. Le temps de latence peut être variable en fonction des données d'entrée. Ces variations dépendent de l'algorithme et de l'implémentation choisis. Il conviendra donc pour être plus précis de définir pour chaque opérateur un temps de latence minimum, moyen et maximum (pire cas).

- **Le temps de cycle**, qui caractérise le temps minimum qui sépare l'acceptation de deux informations en entrée : c'est la bande passante ou débit de l'opérateur. Le temps de cycle vu de l'entrée du circuit « pipeliné » est au minimum égal au temps de cycle de l'étage d'entrée et au maximum égal au temps de cycle de l'étage le plus lent de l'opérateur.

- **La profondeur du pipeline** de l'opérateur définit le nombre maximum de données ou d'informations que l'opérateur peut mémoriser. Cela correspond à la différence maximale entre l'indice de la donnée présente à la sortie et l'indice de la donnée présente à l'entrée, alors que toutes les ressources de mémorisation sont occupées. C'est identique au cas synchrone à mémoire sauf que dans le cas asynchrone le nombre de données présentes dans le pipeline n'est pas imposé. C'est pourquoi on définit une profondeur de pipeline maximale et qu'on parle de pipeline élastique.

- **Le protocole de communication** utilisé pour échanger des informations avec l'environnement et entre opérateurs est fondamental pour les circuits asynchrones car il permet de connecter toutes les cellules du circuit (i.e. étages du pipeline) entre elles. Un point commun à tous les protocoles d'échange asynchrone utilisés aujourd'hui est qu'ils assurent tous la détection de la présence d'une information en entrée et la génération d'une signalisation indiquant d'une part qu'une information a été consommée en entrée et d'autre part qu'une information est disponible en sortie. Cette signalisation bidirectionnelle permet l'implémentation de circuits insensibles aux délais et fonctionnant suivant le mode « flots de données ».

On remarque d'emblée que les opérateurs asynchrones sont caractérisés par un nombre de paramètres *significatifs* plus important que les opérateurs synchrones. En effet, ces derniers sont essentiellement caractérisés par leur bande passante ou débit et un taux de pipeline.

La conséquence immédiate est que le mode de fonctionnement asynchrone apparaît plus riche, car il va permettre de gérer des finesses d'exécution inaccessibles au mode de fonctionnement synchrone exploitées grâce à la complexité du protocole de communication, et à la puissance de synchronisation qu'il apporte. De la sorte, on peut d'ores et déjà entrevoir que le spectre des architectures sera élargi par la prise en compte de ces nouveaux paramètres.

1.2.3 Le principe fondamental des circuits asynchrones : un contrôle local

Le point fondamental du mode de fonctionnement asynchrone est que le transfert d'information est géré localement par une signalisation adéquate. Les opérateurs connectés se synchronisent en échangeant des informations indépendamment des autres opérateurs auxquels ils ne sont pas connectés. Ce mécanisme garantit la synchronisation et la causalité des événements au niveau local et la correction fonctionnelle du système dans son ensemble.

Le contrôle local doit en conséquence remplir les fonctions suivantes : être à l'écoute des communications entrantes, déclencher le traitement localement si toutes les informations sont disponibles (rendez-vous) et produire des valeurs sur les sorties.

Cependant, un signal d'acquittement est également indispensable afin de permettre aux opérateurs qui se trouvent en amont d'être informés que les données qu'ils émettent sont bien consommées.

Globalement donc, pour permettre un fonctionnement correct indépendamment du temps, le contrôle local doit prendre en charge une signalisation bidirectionnelle. Toute action de communication doit être acquittée par le récepteur afin que l'émetteur puisse émettre à

nouveau. Les communications sont dites à poignée de mains ou de type requête-acquittement (figure 1.1).

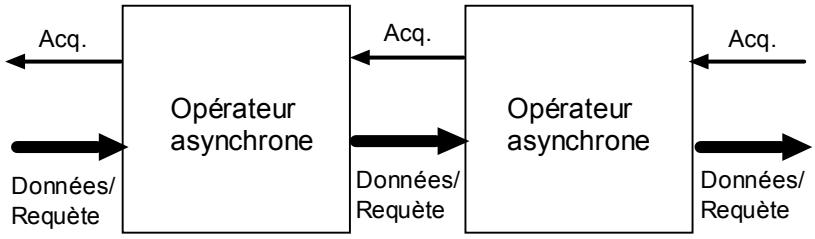


Figure 1.1 : Communication de type poignée de main (i.e. requête/acquittement) entre opérateurs asynchrones pour garantir une synchronisation indépendante du temps.

1.2.3.1 Canaux de communication

Un canal de communication définit un moyen d'échange de données point à point entre modules asynchrones. Il se compose d'un paquet de fils et d'un protocole de communication. Les fils sont l'ensemble des signaux de contrôle nécessaires pour accomplir la communication ainsi que l'ensemble des signaux de données. Le protocole de communication est nécessaire pour gouverner la communication afin de maintenir le fonctionnement correct à l'égard de la spécification souhaitée.

Les échanges de données à travers un canal vont de l'émetteur au récepteur. L'émetteur (ou le récepteur) est défini actif s'il est l'initiateur d'un transfert d'information. Il est passif s'il répond à une demande de transfert. Un émetteur actif initialise le transfert en indiquant que la donnée sur le canal est valide, ce qui est détecté par le récepteur grâce au codage de données adopté. Pour un émetteur passif, le récepteur demande une nouvelle donnée grâce au signal d'acquittement qu'il envoie à l'émetteur. Pour s'affranchir de toute confusion, les canaux de communication tout au long de cette thèse sont considérés reliant des émetteurs actifs à des récepteurs passifs.

Un canal de communication peut aussi être défini de façon à ce qu'il ne joue que le rôle d'élément de synchronisation entre les opérateurs asynchrones, sans contenir aucune donnée. Il est également possible de définir des canaux de communication où les données sont transférées de manière bidirectionnelle [BER 96], [FER 02]. Ce type de canal non abordé dans le cadre de cette thèse est souvent utilisé pour accomplir l'interface avec des mémoires de type RAM, ou ROM.

1.2.3.2 Les protocoles de communication

Pour implémenter une telle gestion des échanges, deux principales classes de protocoles de communication sont utilisés : les protocoles 2 phases (ou NRZ pour Non Retour à Zéro ou encore « *Half-handshake* »), et les protocoles 4 phases (ou RZ pour Retour à Zéro ou encore « *Full-handshake* »).

Le fonctionnement d'un protocole deux phases est décrit figure 1.2 et se déroule en 2 phases comme son nom l'indique :

- ***Phase 1*** *:* c'est la phase active du récepteur qui détecte la présence de nouvelles données, effectue le traitement et génère le signal d'acquittement.
- ***Phase 2*** *:* c'est la phase active de l'émetteur qui détecte le signal d'acquittement et émet les nouvelles données si elles sont disponibles.

Le fonctionnement d'un protocole 4 phases (figure 1.3) se déroule comme suit :

▪ ***Phase 1*** *:* c'est la première phase active du récepteur qui détecte la présence de nouvelles données, effectue le traitement et génère le signal d'acquittement.
• ***Phase 2*** : c'est la première phase active de l'émetteur qui détecte le signal d'acquittement et émet des données invalides (retour à zéro).
• ***Phase 3*** : c'est la deuxième phase active du récepteur qui détecte le passage des données dans l'état invalide et place le signal d'acquittement dans l'état initial (retour à zéro).
• ***Phase 4*** : c'est la deuxième phase active de l'émetteur, qui détecte le retour à zéro de l'acquittement. Il est alors près à émettre de nouvelles données.

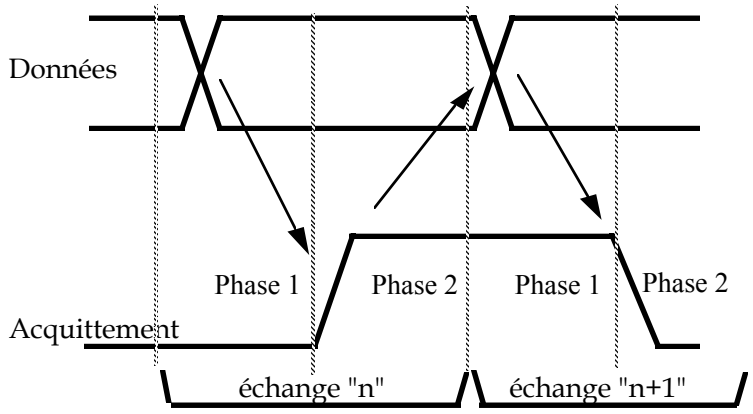


Figure 1.2 : Principe du protocole 2 phases

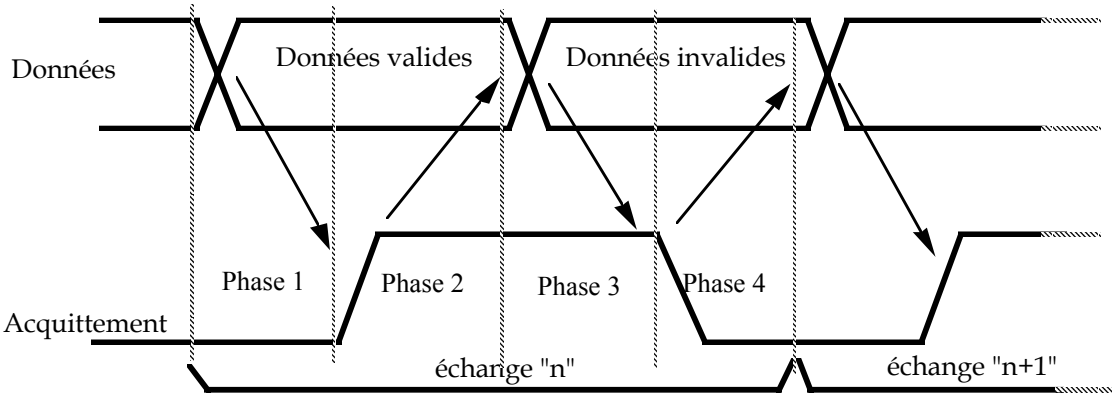


Figure 1.3 : Principe du protocole 4 phases

Il faut noter que dans les deux cas tout changement d'un signal par l'émetteur est acquitté par un changement d'un signal du récepteur et vice-versa. C'est ce qui permet d'assurer l'insensibilité aux temps de traitement. Un changement des données est acquitté par le signal d'acquittement et un changement du signal d'acquittement est acquitté par un changement des données et ainsi de suite.

Il existe bien sûr de multiples variantes de ces protocoles qui se caractérisent par des performances plus ou moins grandes et des implémentations plus ou moins complexes.

Formellement un protocole de communication entre deux opérateurs asynchrones est caractérisé par l'ordonnancement du transfert des informations, par le codage adopté pour les données ainsi que par l'activité des ports. Le nombre de combinaisons possibles est le produit

cartésien de ces options : {2 phases, 4 phases}x{données groupées, double-rails, multi-rails}x{actifs, passifs}.

Il est difficile de dire a priori quel type de protocole deux phases ou quatre phases est le meilleur. Le protocole quatre phases requiert deux fois plus de transitions que le protocole deux phases. Il est a priori plus lent et consomme plus d'énergie. Toutefois, des techniques d'optimisation du pipeline permettent de s'affranchir de la pénalité apparente des phases de retour à zéro.

En terme de vitesse, le protocole quatre phases a ainsi permis jusqu'à aujourd'hui la réalisation de circuits VLSI plus rapides que ceux utilisant le protocole deux phases car ce dernier est pénalisé par une implémentation plus coûteuse.

En ce qui concerne la consommation il faut étudier l'implémentation de ces protocoles. Le protocole deux phases nécessite un matériel plus important que le protocole quatre phases car il doit détecter des transitions et non pas des niveaux. Le nombre plus faible de transitions impliqué dans un protocole deux phases est souvent compensé par la complexité du matériel, et la consommation est comparable à celle des réalisations quatre phases.

En conclusion, le protocole quatre phases est majoritairement utilisé pour implémenter les parties internes d'un circuit intégré. On peut d'ailleurs citer l'expérience de conception de l'Amulet1 [FUR 94]. La première version conçue « en 2 phases », a rapidement été suivie de version « en quatre phases ». Par contre, lorsque les signaux doivent traverser des éléments possédant une latence plus élevée, comme les plots d'un circuit par exemple, le recours à un protocole deux phases est adopté [REN 98].

1.2.3.3 Codage des données

A ce niveau une question légitime à se poser est comment détecter la présence d'une donnée et comment générer un signal qui indique la fin d'un traitement ? La réponse est dans l'adoption d'un codage particulier pour les données. Pourquoi particulier ? Car l'utilisation d'un seul fil par bit de donnée ne permet pas de détecter que la nouvelle donnée prend un état identique à la précédente.

Une première solution consiste à adopter un codage bifilaire ou double rail pour chaque bit de donnée. Cela double donc le nombre de fils par rapport aux réalisations synchrones. Une seconde solution consiste à ajouter un signal de requête aux données, on parle alors de protocole « données groupées ».

Avec deux fils par bit de donnée, quatre états sont utilisables pour exprimer deux valeurs logiques (« 0 », « 1 »). Deux codages sont communément adoptés : l'un utilisant trois états seulement et l'autre utilisant les quatre états (figure 1.4).

Pour le codage trois états, un fil prend la valeur 1 pour coder une donnée à 1 et l'autre fil prend la valeur 1 pour coder la donnée 0. L'état « 11 » est interdit alors que l'état « 00 » représente l'invalidité d'une donnée (utile pour le protocole 4 phases). Ainsi, passer d'une valeur valide à une autre implique de toujours passer par l'état invalide (figure 1.4). Ce codage garantit que le passage d'un état à un autre se fait toujours par changement de l'état d'un seul bit sur les deux, passage détectable sans aléa. Le signal de fin de calcul d'un opérateur peut facilement être généré en détectant qu'un des bits de sortie est passé à 1.

La convention adoptée dans le cas du codage quatre états, est de coder les valeurs 0 et 1 d'un bit avec deux combinaisons. L'une des combinaisons est considérée comme étant de parité impaire et l'autre de parité paire (figure 1.4). Chaque fois qu'une donnée est émise, on change sa parité. Ceci permet de passer d'une valeur logique à une autre sans passer par l'état d'invalidité. En utilisant un code de Gray, le passage d'un état à un autre ne change qu'un seul bit du code [MCA 92]. L'analyse de la parité permet de détecter la présence d'une nouvelle donnée et de générer un signal de fin de calcul. Cette méthode très élégante est cependant d'une implémentation très coûteuse.

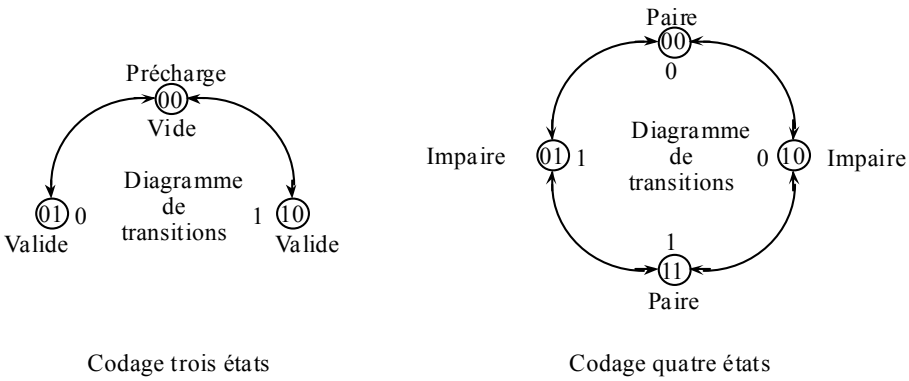


Figure 1.4 : Codage trois et quatre états des données.

Il est clair que le codage trois états est bien adapté au protocole 4 phases alors que le codage quatre états est bien adapté au protocole 2 phases.

L'implémentation du codage quatre états vu qu'elle nécessite la mémorisation de l'état courant, assure la génération du code de Gray et la détection des changements de parité, est plus complexe, plus lente et consomme plus que celle du codage trois états dans le cas général.

Plusieurs logiques ont été proposées pour implémenter le codage 3 états dans les opérateurs de traitement [NIE 94a]. [ELH 95a] propose également une étude approfondie de la « logique cascode différentielle » qui constitue une des implémentations les plus efficaces du codage trois états.

Dans le but de simplifier la détection de présence d'une donnée multi-bits et de limiter le nombre de fils, il a été proposé de séparer l'information de contrôle de l'information de donnée proprement dite. On crée explicitement un signal de contrôle unique, dénommé "requête", qui s'apparente à un signal d'horloge local, typiquement implémenté avec le retard adéquat [FUR 99], [KES 99],[NIEL 99],[TER 99]. Il est utilisé pour déclencher la mémorisation ou le traitement des données qui lui sont associées. L'avantage principal est que les bits de données peuvent être codés à l'aide d'un seul fil. Cette technique est connue sous le nom de « *Bundled Data* » ou données groupées (figure 1.5a) [SUTH 89]. Il est possible de faire de même pour l'acquittement d'une donnée multi-bits. Une fois toutes les données reçues, un signal unique d'acquittement est généré. Dans ces deux cas simplifiés, il existe une hypothèse temporelle qui doit garantir qu'à l'occurrence du signal de requête succède la disponibilité des données, et ceci au niveau du récepteur.

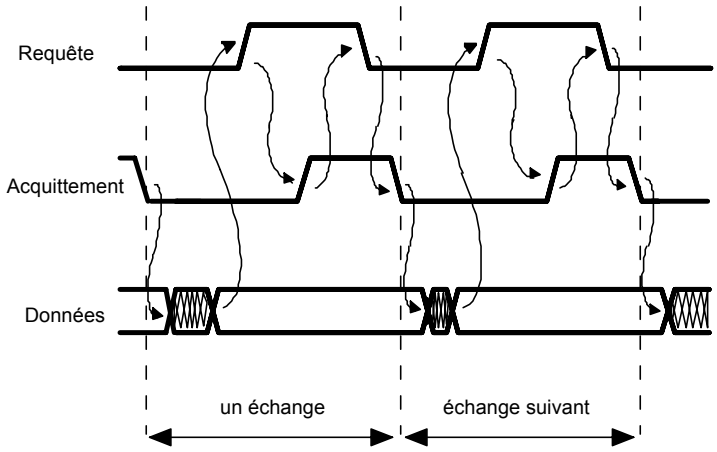


Figure 1.5a : Protocole de communication 4 phases « données groupées » ou « *Bundled Data* ».

1.2.3.4 Implémentation du protocole : la porte de Muller

Pour implémenter les protocoles de communication des circuits asynchrones, les portes logiques élémentaires ne suffisent pas. A cet égard, les portes de Muller proposées dans [MIL 65] sont essentielles. Elles réalisent le rendez-vous entre plusieurs signaux. La sortie d'une porte de Muller copie la valeur de ses entrées lorsque celles-ci sont identiques, autrement elle mémorise l'état précédent. La figure 1.5b représente le symbole d'une porte de Muller, sa spécification ainsi que sa réalisation en différentes formes (à base de portes standard, au niveau transistors).

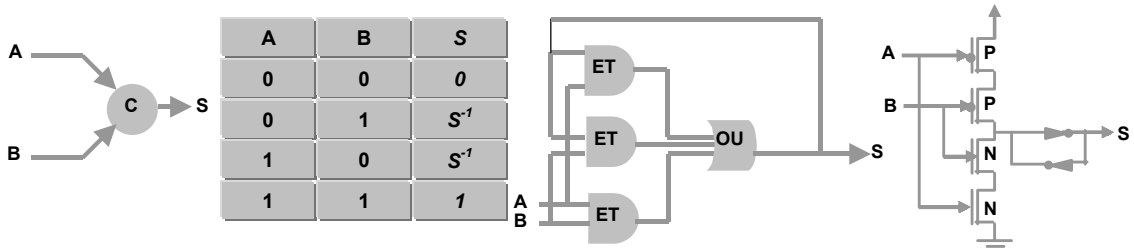


Figure 1.5b : Symbole, spécification, et implémentation de la porte de Muller

Il existe d'autres variantes de la porte de Muller telle que la porte de Muller généralisée (generalized *C element)* [MAR 86]. C'est une porte dans laquelle les signaux qui font monter la sortie à « un » peuvent être distincts de ceux qui la font descendre à « zéro ». La porte de Muller est alors dite dissymétrique et son fonctionnement est explicité sur un exemple à 3 entrées figure 1.5c.

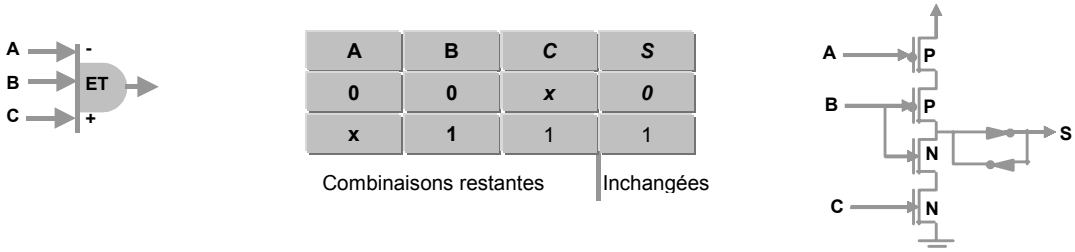


Figure 1.5c : Symbole, spécification, et implémentation de la porte de Muller dissymétrique [RIGA 02]

1.2.3.5 Signaux de fin de calcul

Les protocoles décrits ci-dessus utilisent un signal d'acquittement que chaque opérateur doit générer. Ce dernier représente une complexité additionnelle nécessaire à la réalisation d'un circuit asynchrone (on ne considère pas ici les nouveaux travaux qui consistent à ne pas implémenter certains signaux d'acquittement lorsque des connaissances temporelles sont connues à priori; cette technique est connue sous le nom de « relative timing » [STE 99], [ROT 99]).

Cette complexité est d'ailleurs à comparer avec la complexité induite par l'utilisation d'une horloge. De nombreuses méthodes existent et ont été étudiées pour générer un signal de fin de calcul. Dans les protocoles illustrés ci-après, l'information de donnée valide/invalide peut indifféremment être portée par un signal de requête ou un codage des données.

La première solution suggérée par Chuck Seitz pour la conception du DDM1, premier calculateur *data-flow* [DAV 78], est basée sur l'utilisation d'horloges locales. L'arrivée d'une requête déclenche l'horloge interne du module et après un nombre de cycles déterminés, le calcul est terminé, l'horloge interne est stoppée et un signal d'acquittement généré. Cette technique est adaptée à la conception de modules de tailles significatives, mais très coûteuse et lente pour des modules de petites tailles.

La deuxième solution largement utilisée aujourd'hui pour la conception de circuits intégrés est d'avoir recours à « des hypothèses temporelles » ou « modèles de délais ». On adopte ici une mesure classique de la chaîne critique pour évaluer les temps de calcul de l'opérateur. Ce temps est utilisé pour dimensionner le retard à insérer entre la requête et l'acquittement (figure 1.6). Il faut remarquer que la même technique est utilisée aujourd'hui pour la conception des mémoires (commande des amplificateurs).

La troisième solution consiste à adapter la logique bifilaire pour implémenter un codage des données qui facilite la détection de fin. Les figures 1.7 et 1.8 illustrent le principe de la détection de fin lorsque des codages trois et quatre états respectivement sont utilisés pour coder les données. Les signaux S0 et S1 représentent les deux rails d'une donnée 1 bit en sortie de l'opérateur. Dans le cas du codage trois états il suffit de détecter qu'un des rails est passé à 1 pour signaler la fin du traitement. Lorsque les deux rails sont à zéro, la phase de remise à zéro est terminée (protocole quatre phases). Dans le cas du codage quatre états il suffit de détecter le changement de parité (ou exclusif). Un protocole deux phases est bien adapté car chaque changement de valeur est détecté par le test de parité.

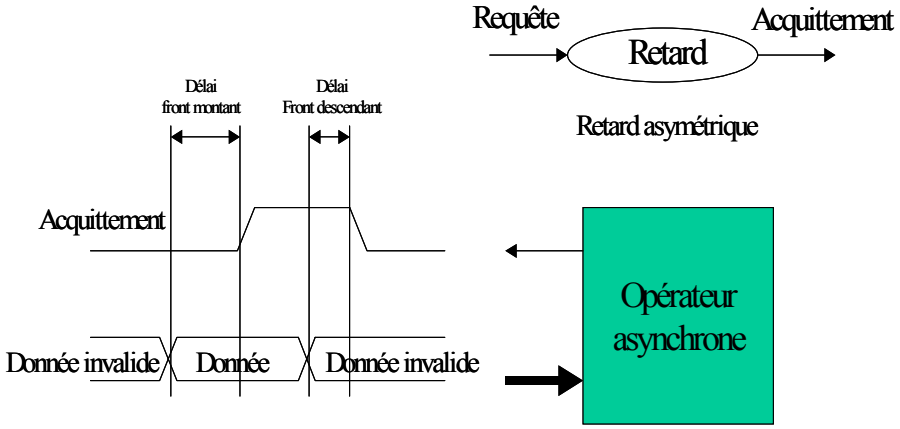


Figure 1.6 : Génération de l'acquittement à l'aide d'un retard.

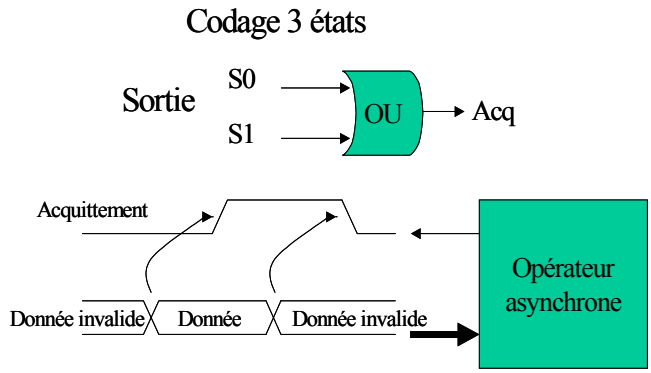


Figure 1.7 : Génération de l'acquittement en exploitant le codage des données 3 états

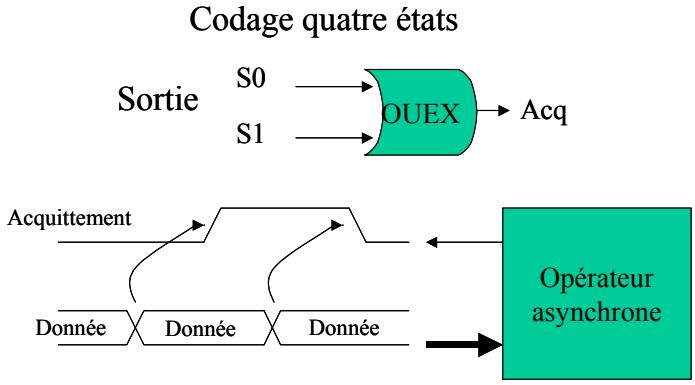


Figure 1.8 : Génération de l'acquittement en exploitant le codage des données 4 états

Par ailleurs, certaines fonctions possèdent des caractéristiques telles que certains signaux de sortie peuvent renseigner sur la fin du traitement. C'est le cas par exemple de certaines fonctions arithmétiques (addition [REN 94] et division [WIL 91], [REN 96], voir également [ELH 95], [ELH 95a]. Dans ce cas il suffit de tester certains des bits de sortie ce qui réduit la complexité de la détection de fin.

1.2.4 Critères de classification des circuits asynchrones

Nous l'avons vu, l'approche asynchrone permet le fonctionnement correct d'un circuit indépendamment de la distribution des retards dans les portes et les connexions. Pour être indépendant des temps de propagation, la progression des signaux dans les éléments logiques du circuit est contrôlée par les signaux de « poignée de main » (« *handshake signals* »). C'est le prix à payer pour supprimer la synchronisation globale réalisée par un signal d'horloge.

Mais ce n'est pas suffisant, la logique doit être exempte d'aléa (« *Hazard free* »). En effet, dans les circuits synchrones, la discrétisation du temps, implémentée par échantillonnage des signaux par l'horloge, permet d'ignorer tous les phénomènes transitoires qui peuvent survenir dans les parties logiques combinatoires.

A contrariori, dans un circuit insensible aux délais, toute transition d'un signal peut être interprétée comme un événement porteur d'information. La principale difficulté réside donc dans la conception de parties combinatoires ou séquentielles sans aléa.

1.2.4.1 Les aléas

La présence d'aléas dans un circuit peut être détectée à différents niveaux de la conception. Un séquenceur peut par exemple contenir des aléas fonctionnels qui trouvent leur origine dans la spécification de la fonction elle-même. Si ce cap est passé, il peut encore apparaître des aléas au niveau de l'implémentation. Cela signifie que la conception d'un circuit insensible aux délais ne s'arrête pas après sa spécification. Il faut encore s'assurer que les techniques de réalisation choisies sont exemptes d'aléa.

Nous citons ci-après les différents types d'aléas dont doit se prémunir tout circuit asynchrone ne faisant aucune hypothèse temporelle.

- **Les aléas combinatoires fonctionnels.** On appelle aléas combinatoires fonctionnels, les aléas qui peuvent être détectés et supprimés au niveau de la spécification logique de la fonction en étudiant et modifiant celle-ci. Ces aléas ne dépendent pas de l'implémentation et de la distribution des délais.

Si une transition possède un aléa fonctionnel, aucune implémentation de la fonction n'est assurée d'éviter les aléas pendant cette transition indépendamment des délais dans les fils et les portes [EIC 65], [BRE 72]. Dans la suite de cette thèse, les transitions des entrées sont supposées exempts d'aléa fonctionnel.

- **Les aléas séquentiels.** Ce sont des aléas qui trouvent leur origine dans les signaux bouclés. On peut les détecter lors de la spécification d'un problème, par exemple lors de l'étude d'une table de flots [UNG 69]. On cite souvent les aléas essentiels qui peuvent être détectés en faisant changer la même entrée une fois puis trois. Il y a aléa essentiel si l'état final n'est pas le même dans les deux cas.

- **Les aléas combinatoires logiques.** Ce sont des aléas dont l'apparition dépend de la distribution des délais dans la logique. On parle d'aléas combinatoires logiques si l'apparition de l'aléa dépend de l'implémentation de la fonction logique. L'implémentation est donc de toute première importance en asynchrone et nécessite la conception d'outils de synthèse qui intègrent la notion d'aléa. En particulier, l'étape de « *Technology mapping* » est spécifique à la conception de circuits asynchrones. L'implémentation d'une fonction logique à l'aide de portes logiques de base doit être contrôlée pour ne pas générer d'aléas combinatoires logiques.

  Deux principaux types d'aléas existent dans cette classe [UNG 69],[UNG 71] : Les aléas de type SIC (« *Single Input Change* ») qui arrivent au cours de la transition d'une seule entrée et les aléas de type MIC (« *Multiple Input Change* ») qui se produisent lorsque plusieurs entrées changent.

  On montre [WAK 94] que pour une implémentation quelconque sous forme de SOP (Produit de Somme ou "*Sum Of Product*"), aucun aléa ne peut survenir lors des transitions $0\rightarrow0$, $0\rightarrow1$, et $1\rightarrow0$ sur la sortie. Ainsi la synthèse sans aléa de circuits SOP nécessite seulement d'éliminer les aléas statiques « 1 ». Par ailleurs la condition nécessaire et suffisante pour éviter les aléas logiques statiques de type MIC pour les implémentations

SOP à 2 niveaux a été démontrée dans [EIC 65]. Pour les implémentations SOP multi-niveaux, les conditions sont plus complexes et discutées dans [BRE 75].

La suppression d'aléas revient à un problème de couverture du tableau de Karnaugh. Les transitions 1→1 doivent être couvertes. Pour les transitions 1→0 et 0→1, un produit qui intersecte avec la transition doit contenir le monôme de départ et le monôme d'arrivée. Les transitions 0→0 ne génèrent pas d'aléas dans les réalisations SOP.

Ces conditions suffisent pour implémenter un quelconque aléa de type MIC. S'il existe plusieurs aléas de type MIC, le problème de couverture peut très bien ne pas avoir de solution. Pour un ensemble de transitions MIC (fonctions de transitions d'une machine à états) l'existence d'une couverture du tableau de Karnaugh conduisant à une réalisation de type SOP sans aléa n'est pas assurée.

Chacune de ces classes d'aléas (combinatoire fonctionnel, séquentiel, et combinatoire logique) comprend les aléas statiques et les aléas dynamiques.

- **Les aléas statiques** Lorsqu'un signal doit rester constant et qu'il change de niveau deux fois successivement on parle d'aléa statique. On les nomme aussi « *spike* » dans le jargon du concepteur. Toute porte « ET » et « OU » est susceptible de générer ce type d'aléas si les deux entrées changent « simultanément » (figure 1.9).

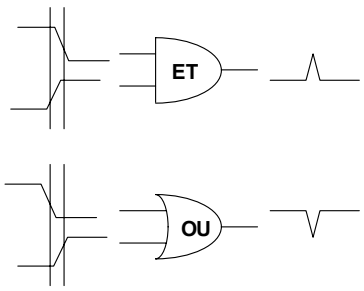


Figure 1.9 : Aléas statiques générés par des portes logiques et / ou.

- **Les aléas dynamiques.** Lorsqu'un signal doit changer de niveau une seule fois et qu'il change de niveau plusieurs fois on qualifie cette situation d'aléa dynamique.

En conclusion, les circuits asynchrones requièrent une conception attentive et fine de toutes les parties du circuit (combinatoires et séquentielles). Au §1.2.5, on voit que l'indépendance aux délais et la contrainte de conception sans aléa, sont plus ou moins respectées en fonction de la classe de circuits asynchrones à laquelle on s'intéresse. La conséquence majeure est que les outils de synthèse logique conventionnels (synchrones) ne peuvent être utilisés car ils ne sont pas conçus pour générer des circuits logiques sans aléa.

Les aléas combinatoires fonctionnels sont difficiles à détecter et à corriger puisqu'ils dépendent uniquement de la spécification du circuit. Comme mentionné, plus haut ces aléas ne sont pas abordés dans la suite. Les aléas de type logique doivent être pris en compte au cours de la réalisation des circuits.

1.2.4.2 Modèles de délais et de circuits

Une façon de distinguer les différents types de circuits asynchrones est de considérer les modèles qui régissent le comportement des délais, des circuits et des environnements.

Les délais sont communément caractérisés par un comportement dit « pur » ou un comportement dit « inertiel » [UNG 69][UNG 71]. Le délai pur ne change pas la forme d'onde des signaux mais les retarde temporellement, alors que le délai inertiel peut éventuellement changer la forme d'onde et possède une période de seuil D qui élimine les impulsions de durée inférieure à D. Une autre façon de caractériser le délai est de le faire vis à vis du temps : Un délai est dit « fixe » lorsqu'il possède une valeur connue déterminée, il est dit « borné » lorsqu'il possède une valeur située dans un intervalle connu. Enfin, un délai peut être « non borné », et sa valeur est finie mais inconnue.

Le comportement d'un circuit peut être modélisé en analysant les modèles de ses constituants. Le premier modèle de circuit dit « porte simple » associe à chaque porte une valeur de délai correspondant à son temps de traversée. Pour le modèle de « porte complexe » un délai est associé à un sous-ensemble de portes simples. Ainsi, avec ce modèle le temps de traitement d'un réseau de portes est caractérisé par un seul paramètre, sans préciser le comportement fin interne. Les fils qui relient les portes sont de la même manière caractérisés par des délais. Ainsi, un modèle de circuit est défini par le modèle des connexions (fils) et le modèle des composants élémentaires (portes).

1.2.4.3 Modèles d'environnement

Il est aussi très important de caractériser le comportement de l'environnement dans lequel s'insère le circuit. Le circuit et son environnement forment un système fermé, dit « circuit complet » [MIL 65]. Selon les terminologies utilisées par Brzozowski [BRZ 89], les modèles d'environnement sont classifiés de la manière suivante :

- **Mode entrée-sortie** : dans ce cas, l'environnement interagit avec le circuit sans contrainte de délais. Il est possible pour l'environnement de changer les entrées sans exiger que le circuit soit dans un état stable.

- **Mode fondamental** : dans ce cas, il existe des contraintes temporelles pour que l'environnement et le circuit interagissent. Précisément, ce modèle impose à l'environnement d'attendre la stabilisation du circuit ainsi que la production d'une sortie, avant de répondre à celui-ci par le changement *d'une seule de ses entrées* (temps de « maintien » d'une flip-flop, temps logique des langages synchrones). Par ailleurs, si l'environnement ne connaît pas l'état stable du circuit, il doit respecter le délai le plus long pour stabiliser le circuit. En conséquence, il est nécessaire d'utiliser des délais bornés pour les fils et les portes du circuit. Les langages ESTEREL, LUSTRE, SIGNAL [IEE 91] en informatique, et les circuits synchrones en électronique utilisent le modèle d'environnement en mode fondamental.

- **Mode rafale** : C'est une extension du mode fondamental dans le sens où dans les mêmes conditions (stabilité du circuit avec production d'une sortie), l'environnement peut procéder au changement de plusieurs entrées (et non plus d'une seule) dans un ordre quelconque.

1.2.5 Classification des circuits asynchrones

Certaines réalisations asynchrones parviennent à relâcher la contrainte de correction fonctionnelle indépendamment des délais en introduisant des hypothèses temporelles qui conduisent à des réalisations et une conception plus simples.

Nous présentons dans ce qui suit la terminologie communément adoptée pour classifier les circuits asynchrones qui se caractérisent par des hypothèses temporelles plus ou moins fortes. (figure 1.10). Nous commençons par la classe des circuits asynchrones dont le fonctionnement respecte fondamentalement la notion d'asynchronisme telle que définie plus haut. Puis, viennent les classes dérivées qui introduisent des hypothèses temporelles de plus en plus fortes et qui se situent entre le mode de fonctionnement asynchrone pur et le mode de fonctionnement synchrone.

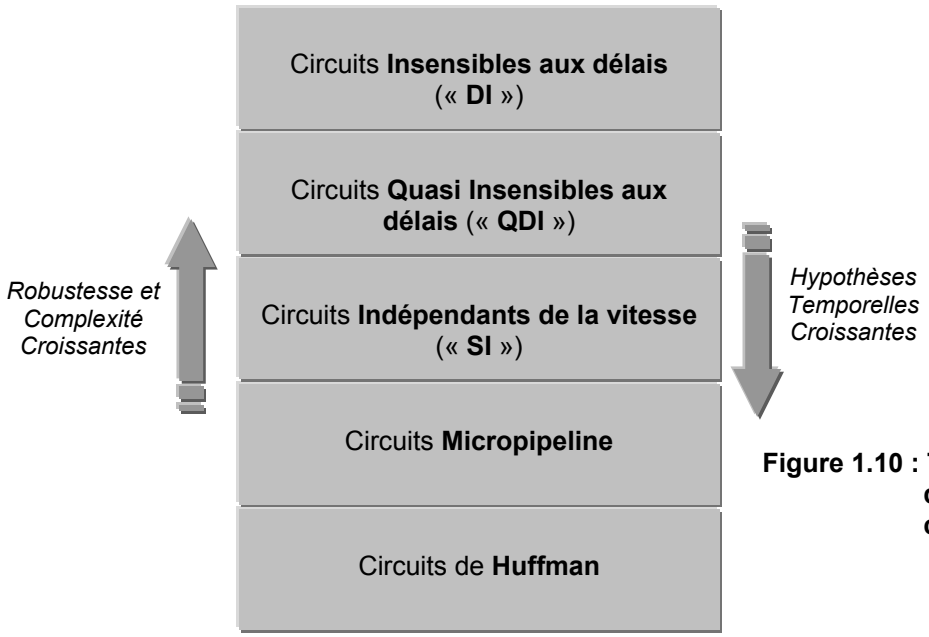


Figure 1.10 : Terminologie des différentes classes de circuits asynchrones

1.2.5.1 Les circuits insensibles aux délais (« DI » pour « Delay Insensitive »)

Cette classe de circuits basée sur les travaux de [WES 67] et re-formalisée par [UDD 86] utilise un mode de fonctionnement purement asynchrone. Aucune hypothèse temporelle n'est introduite. Autrement dit, les circuits de cette classe sont fonctionnellement corrects indépendamment des délais introduits par les fils ou les éléments logiques (quelle que soit leur complexité). D'un point de vue plus théorique cela signifie qu'ils sont basés sur un modèle de délai pour les fils et les éléments qui est « non-borné ».

Ainsi, on suppose qu'un circuit répondra toujours correctement à une sollicitation externe après un temps de calcul inconnu. Ceci impose donc au récepteur d'un signal de toujours informer l'expéditeur que l'information a été reçue. Les circuits récepteurs doivent donc être capables de détecter la réception d'une donnée et/ou la fin de son traitement. Les circuits émetteurs quant à eux doivent attendre un compte rendu avant d'émettre une nouvelle donnée.

Soulignons à ce niveau, les fortes contraintes de réalisation pratique qu'impose l'utilisation de ce modèle. La seule solution connue consiste à utiliser un modèle de circuit de type « portes complexes » pour les composants élémentaires [EBE 91][JOS 93]. Dans ce cas la construction de ces circuits se fait à partir de composants standards plus complexes que de simples portes logiques, et qui peuvent posséder plusieurs entrées et plusieurs sorties [EBE

91], [HAU 95]. Des hypothèses de délais peuvent exister, mais elles restent internes et locales aux cellules complexes.

1.2.5.2 Les circuits quasi insensibles aux délais (« QDI » pour « *Quasi Delay Insensitive* »)

Cette classe de circuits adopte le même modèle de délais « non borné » pour les connexions mais y ajoute la notion de fourche isochrone (« *isochronic fork* ») [MAR 90][BER 92],.

On appelle fourche un fil qui connecte un expéditeur unique à deux récepteurs. On la qualifie d'isochrone lorsqu'on impose que les délais entre l'expéditeur et les deux récepteurs sont identiques. Moyennant cette hypothèse, un circuit peut se permettre de ne tester qu'une branche d'une fourche en supposant que le signal s'est propagé de la même façon dans l'autre branche. La conséquence heureuse est que l'on peut de la sorte autoriser l'acquittement d'une seule des branches de la fourche isochrone sans avoir vérifié la propagation du signal dans l'autre branche.

Martin [MAR 93] a montré que l'hypothèse temporelle de « fourche isochrone » est la plus faible à ajouter aux circuits insensibles aux délais pour les rendre réalisables avec des portes à plusieurs entrées et une seule sortie. Ainsi, les circuits quasi insensibles aux délais se caractérisent par l'adoption d'un modèle de délais pour les connexions qui est de type « non borné » avec en plus l'hypothèse de « fourche isochrone », et un modèle de type « porte simple » pour les composants élémentaires des circuits. Les circuits quasi insensibles aux délais sont donc implémentables avec des cellules standard telles qu'on les utilise pour la conception de circuits synchrones.

En pratique l'hypothèse temporelle de « fourche isochrone » est assez faible, et est facilement remplie par une conception soignée, en particulier au niveau du routage [MAR 90], [VAN 92]. Certains processeurs appartenant à cette catégorie de circuits asynchrones sont les plus performants aujourd'hui. On peut citer le MiniMips [MAR 97], ASPRO [REN 98], MICA [ABR 01], et plus récemment Lutonium [MAR 03].

1.2.5.3 Les circuits indépendants de la vitesse (« SI » pour « *Speed Independent* »)

Les circuits indépendants de la vitesse initialement décrit par [MIL 65] font l'hypothèse que les délais dans les fils sont négligeables tout en conservant le modèle « non borné » pour les délais dans les portes. La fourche isochrone est une façon équivalente de formuler le modèle « indépendant de la vitesse ». Même si pendant longtemps la communauté a tenté de cerner les différences entre ces deux modèles, il y a aujourd'hui un consensus pour considérer qu'en pratique les modèles « QDI » et « SI » sont équivalents. Hauck [HAU 95] montre comment une fourche isochrone peut être représentée par un circuit indépendant de la vitesse (figure 1.11).

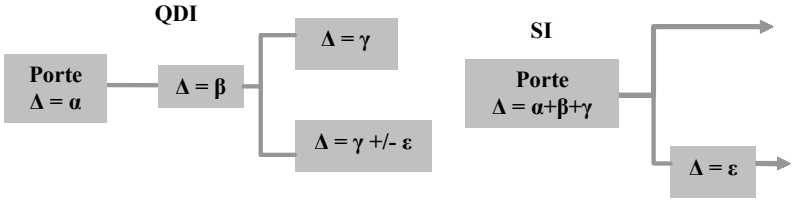


Figure 1.11 : Equivalence entre les modèles "QDI" et "SI"

Pour comparer les deux modèles, on peut dire que le modèle QDI permet d'identifier les fourches isochrones c'est à dire les connexions du circuit qui ne sont pas insensibles aux délais. Le modèle « SI » en revanche ne les identifie pas, il considère toutes les connexions comme potentiellement « sensibles » et toutes les fourches comme isochrones. Identifier les fourches isochrones permet à un outil d'effectuer des vérifications de temps seulement lorsque nécessaire. Le modèle QDI est plus précis ce qui le rend plus puissant.

1.2.5.4 Circuits Micropipelines

La technique de Micropipeline a été introduite par Ivan Sutherland [SUT 89]. Les circuits de cette classe sont composés de *parties contrôles insensibles aux délais* qui commandent des *chemins de données conçus* en utilisant *le modèle de délai borné*.

La structure de base de cette classe de circuits est le contrôle d'une queue de type FIFO (figure 1.12). Elle se compose d'éléments identiques connectés tête-bêche. On rappelle que les opérateurs notés « C » sont des portes de Muller dont la sortie est une copie des niveaux d'entrée lorsqu'ils sont identiques. Cette sortie est mémorisée lorsque les entrées sont différentes.

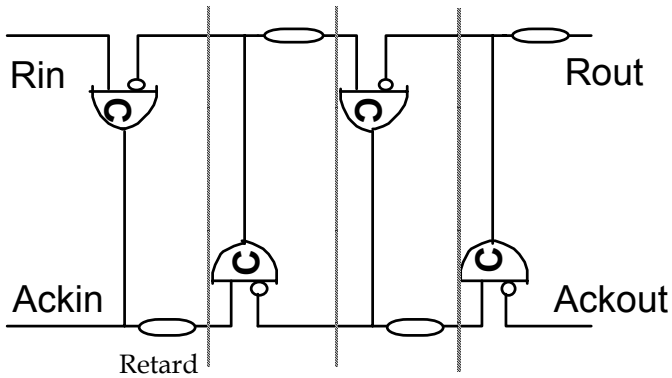


Figure 1.12 : Structure de base des circuits Micropipelinés : Queue de type FIFO

Le circuit réagit à des transitions de signaux et non pas à des états (protocole deux phases). On parle également d'une logique d'événements puisque chaque transition est associée à un événement. Ainsi, si on suppose tous les signaux à zéro initialement, une transition positive sur Rin provoque une transition positive sur Ackin qui se propage également à l'étage suivant. Le deuxième étage produit une transition positive qui d'une part se propage à l'étage suivant mais qui d'autre part revient au premier étage l'autorisant à traiter une transition négative cette fois. Les transitions de signaux se propagent donc dans la structure tant qu'elles ne rencontrent pas une cellule « occupée ». C'est un fonctionnement de type FIFO.

Cette structure de base peut être enrichie d'opérateurs de mémorisation ou pipeline, et d'opérateurs de traitement combinatoires (figure 1.13).

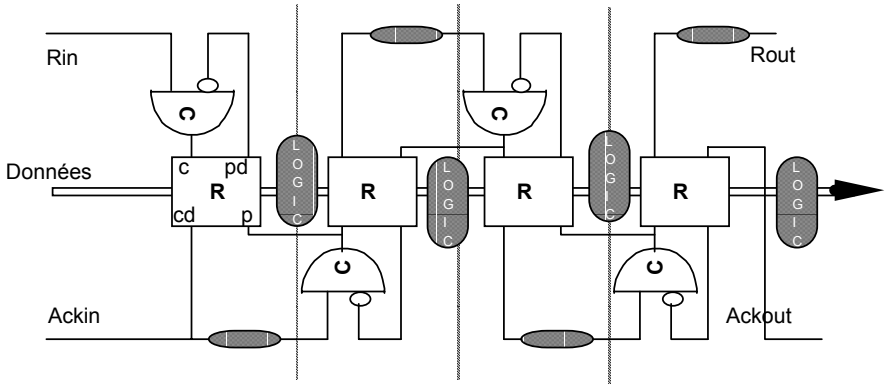


Figure 1.13 : Structure Micropipeline avec traitements et registres

Dans cette structure les opérateurs dénotés « R » sont des registres qui capturent la donnée entrante sur l'occurrence du signal C. Ils produisent le signal « Cd » lorsque la donnée est mémorisée. Durant cette phase la donnée précédemment mémorisée dans le registre est maintenue en sortie. Lorsque P est actif, le registre laisse passer la donnée d'entrée à la sortie. Le signal « Pd » indique que le registre est bien transparent. La structure de la figure 1.13 dépouillée de la logique réalise une FIFO (initialement tous les registres laissent passer les données). Avec la logique combinatoire, la structure est celle d'un circuit asynchrone « pipeliné ».

Le protocole de communication utilisé ici est de type « *bundled data* » (données groupées). Lors d'une occurrence de Rin les données d'entrée sont stockées dans le premier étage. Rin est propagé vers l'étage suivant qui stocke le résultat transmis par la logique du premier étage. La propagation de Rin est retardée de façon à s'assurer que la logique combinatoire a bien convergé avant la capture du résultat dans le deuxième étage. La capture du deuxième étage étant effectuée, le premier registre est rendu passant ce qui permet le traitement de la donnée présente dans le premier étage et autorise la prise en compte d'un nouvel événement sur Rin du premier étage.

La motivation première pour le développement de cette classe de circuits était de permettre un pipeline élastique. En effet, le nombre de données présent dans le circuit peut être variable, les données progressant dans le circuit aussi loin que possible en fonction du nombre d'étages disponibles ou vides. Cependant, ce type de circuits révèle un certain nombre d'inconvénients.

Tout d'abord, il faut remarquer que les problèmes d'aléas ont été écartés en ajoutant des retards sur les signaux de contrôle. Cela permet en fait de se ramener à un fonctionnement en temps discret dans lequel on autorise la mémorisation de données seulement lorsqu'elles sont stables (à la sortie des parties combinatoires). Les délais étant de durée fixe dans la proposition initiale de Sutherland, ce type de circuit effectue les traitements en un temps pire cas. On ne peut donc pas tirer parti de la variation dynamique de la chaîne critique des opérateurs de traitement.

Il est possible cependant de s'affranchir de cette contrainte en utilisant des registres et des opérateurs combinatoires capables de générer leur propre signal de fin de calcul sans avoir recours à des délais fixes. On obtient alors des structures du type de celle décrite figure 1.14 dans laquelle les délais fixes sont remplacés par des délais variables implémentés dans les registres et la logique combinatoire. Ici, le temps de calcul peut varier en fonction des données. On sort alors du cadre des circuits utilisant le modèle de délai borné et le circuit peut être rendu indépendant de la vitesse puisque la seule hypothèse temporelle consiste à assurer

qu'à l'occurrence du signal de contrôle, Rin, succède les données. Des circuits SI peuvent également être obtenus en détectant localement la fin de calcul logique en remplaçant la logique combinatoire par de la logique à précharge [FUR 96], ou en supprimant les latchs explicites par utilisation de la logique différentielle à précharge [REN 96]. Des bascules de mémorisation sur double front peuvent être utilisées dans le cas du protocole deux phases [YUN 96]. Enfin des protocoles quatre phases optimisés permettent d'obtenir un taux de remplissage important de la structure afin d'augmenter les performances [FUR 96b]

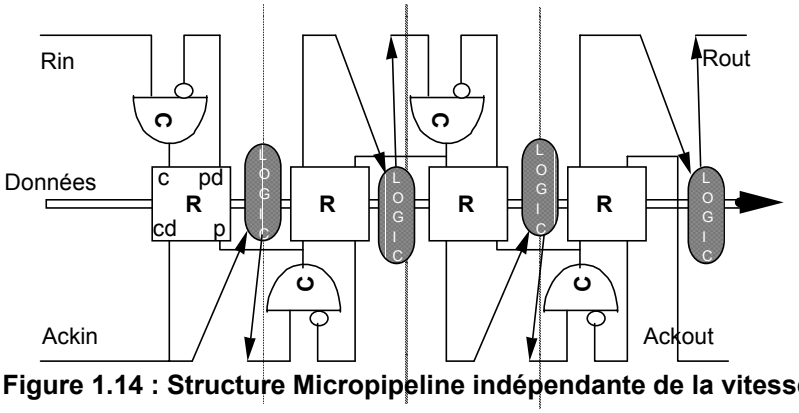


Figure 1.14 : Structure Micropipeline indépendante de la vitesse

1.2.5.5 Circuits de Huffman

Les circuits de cette classe utilisent un modèle de délai identique aux circuits synchrones. Ils supposent que les délais dans l'ensemble des éléments et des connexions du circuit sont bornés ou même de valeurs connues. Les hypothèses temporelles sont donc du même ordre que celles utilisées pour la conception de circuits synchrones. Leur conception repose sur l'analyse des délais dans tous les chemins et boucles mais aussi sur des hypothèses de délais relatives aux signaux issus de l'environnement (mode fondamental).

Huffman a le premier proposé la conception de machines à états composées de boucles combinatoires. Les boucles réalisent la mémorisation des états (figure 1.15). Pour être correct le circuit comporte des délais dans les boucles afin respecter les hypothèses temporelles.

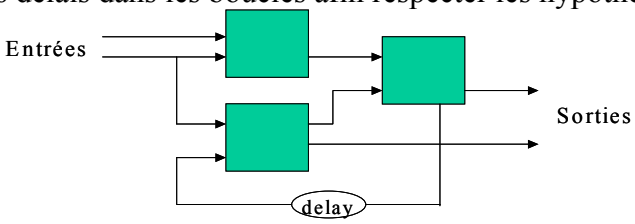


Figure 1.15 : Synoptique des machines à états de Huffman.

Des travaux plus récents ont porté sur la conception de machines à états du même type à partir d'une spécification appelée « *Burst mode* » (mode rafale) [DAV 93], [NOW 95].

1.3 Méthodologies de spécification et outils de synthèse des circuits asynchrones

Il est nécessaire de définir des méthodologies et des outils de conception dédiés précisément aux circuits asynchrones. En effet, on ne peut se contenter des instruments de conception pour les circuits synchrones (largement connus et utilisés) pour deux raisons principales :

- Les langages de description de matériel tel que VHDL et Verilog utilisés dans la conception synchrone, sont inadaptés aux exigences de l'asynchrone. En particulier, ces langages ne sont pas suffisamment flexibles pour modéliser la concurrence et la séquentialité des blocs fonctionnels asynchrones. Par ailleurs, ils ne permettent pas de décrire les communications par canaux interposés.

- La seconde raison est liée au problème majeur inhérent à la conception asynchrone : la façon de traiter les aléas. Ce problème nécessite d'adopter une synthèse exempte d'aléas puisque les outils de CAO standards ne gèrent pas ce type de contrainte.

  En effet, en synchrone il suffit de garantir la stabilité des signaux aux fronts montants de l'horloge, alors qu'en asynchrone, les signaux n'étant pas échantillonnés, tout aléa est potentiellement la source d'une exécution erronée.

  La première solution proposée fut d'utiliser des délais inertiels pour filtrer les « *glitchs* » ou aléas indésirables. Il fut également proposé de retarder certains signaux par rapport à d'autres, pour éliminer les aléas dans les circuits adoptant le modèle de délai « borné ». Aujourd'hui, des méthodes de synthèse existent pour éliminer les aléas.

  Les classes d'aléas combinatoires que les outils de synthèse savent éliminer sont les aléas qui apparaissent lorsqu'une seule entrée change (SIC hazards : *Single Input Change hazards*) et les aléas qui apparaissent lorsque plusieurs entrées changent (MIC hazards : *Multiple Input Change hazards*). Dans les deux cas on considère un modèle de délais "non borné" pour les portes et les connexions. Les techniques existantes classiques permettent d'éliminer les aléas combinatoires SIC et MIC dans des circuits à deux niveaux (AND-OR) [UNG 69], [NOW 95], [THE 96].

  Comme nous l'avons mentionné précédemment, il existe aussi des algorithmes de synthèse de circuits multi-niveaux sans aléa. Une première méthode propose de partir d'un circuit sans aléa à deux niveaux, et d'appliquer des transformations pour passer à un circuit multi-niveaux sans introduire d'aléa [UNG 69], [KUN 92]. Il a par ailleurs été proposé de synthétiser directement un circuit multi-niveaux sans aléa en utilisant des diagrammes de décision binaires (BDD : *Binary Decision Diagrams*) [LIN 94].

  D'autres algorithmes ont été développés pour réaliser des circuits multi-niveaux sans aléa en utilisant des bibliothèques de cellules standard (« *technology mapping* ») [SIE 93], [KUD 96].

1.3.1 Méthodologies de spécification et de synthèse logique

On peut distinguer deux orientations majeures pour les méthodologies de spécification et de représentation des circuits asynchrones. La première concerne les méthodes basées sur un langage de description de haut niveau tel que VHDL, CHP, Occam, Tangram, Balsa. La seconde regroupe les méthodes basées sur une description du comportement des signaux du circuit en graphes telle que les réseaux de Petri (*Petri Net* ou PN), les graphes d'états, les STG (*Signal Transition Graph*).

Les méthodes basées sur un langage de description de haut niveau sont expressives et adéquates pour décrire des systèmes complexes avec une structure hiérarchique et modulaire.

En ce sens, elles facilitent la tâche du concepteur, toutefois le circuit synthétisé à partir de ces spécifications n'est pas optimal.

Les méthodes basées sur un graphe sont quand à elles préférées pour l'analyse temporelle et la synthèse. L'inconvénient de ce type de représentation reste à l'évidence le caractère ardu et pénible de la spécification que doit décrire le concepteur.

On peut dès lors imaginer qu'une combinaison des deux approches pourrait permettre d'englober les deux avantages. En amont une spécification en langage de haut niveau offrirait au concepteur la possibilité de décrire aisément des systèmes complexes, et plus en aval une représentation en graphes permettrait d'analyser temporellement ces systèmes, de les synthétiser et d'optimiser les circuits générés.

Par ailleurs, les méthodologies de synthèse des circuits asynchrones semblent intimement liées à leur spécification.

En effet, globalement les circuits décrits en langages de haut niveau sont soumis à une synthèse dirigée par la syntaxe. Ce type de synthèse est transparent : à une construction du langage au niveau de la spécification correspond, de façon directe, un composant « poignée de main » (« *Handshake component* ») à tel point que pour un concepteur expérimenté, il est relativement aisé d'envisager l'architecture du circuit qui correspond à la description initiale.

D'un autre côté, les circuits spécifiés par la méthode des graphes suivent la méthode de synthèse logique dans laquelle ces graphes subissent des vérifications et des transformations pour répondre à des exigences précises qui permettent d'affirmer que le circuit qu'ils représentent est synthétisable. Ensuite des équations logiques sont générées suivies des phases d'optimisations et de projections technologiques pour enfin donner le circuit final.

1.3.1.1 Spécification basée sur les graphes et synthèse « logique »

Les méthodes de spécification basées sur les graphes spécifient le comportement des circuits asynchrones avec un niveau d'abstraction faible (fréquemment au niveau signal). Les graphes utilisés pour ces approches incluent les réseaux de Petri [PET 62], les graphes de transitions de signaux (« STG » pour « *Signal Transition Graph*») [CHU 87], les diagrammes de changements [VAR 90], les machines à états asynchrones [YUN 92][DAV 93][HUF 54], et les graphes d'états [MUL 59].

De nombreuses méthodes de spécification des comportements des circuits asynchrones utilisent le réseau de Petri car il permet une représentation graphique qui peut décrire des événements concurrents autant que séquentiels. A titre d'exemple, les réseaux M (*M-Nets*) proposés par Seitz [SEI 70], les réseaux I (*I-Nets*) définis par Molnar [MOL 85] et les graphes de signaux développés par Yakovlev [ROS 85] sont une classe restreinte de réseaux de Petri (graphes marqués). Ces modèles représentent la concurrence entre des événements, mais ils ne peuvent pas décrire le comportement conditionnel, tel que un choix entre des entrées.

Un autre modèle important aujourd'hui largement étudié et utilisé par la communauté asynchrone, est le graphe de transitions de signaux (STG). Le STG permet de modéliser la concurrence et adopte une sémantique limitée pour les choix sur les entrées. Plusieurs méthodologies de conception de circuits asynchrones existantes sont basées sur ce type de graphes [CHU 87][COR 02][MYE 92][YKM 94][SEN 92].

La spécification basée sur les machines à états asynchrones de type Mealy (ASM ou «*Asynchronous State Machine* ») est également connue [NOW 93]. L'ASM est en fait une

machine de Huffman dans laquelle à chaque état, elle peut recevoir des entrées, générer des sorties et puis changer son état. Sa structure est alors similaire à la machine à états synchrones mais sans l'élément de stockage piloté par une horloge.

Du point de vue de la synthèse, les circuits spécifiés par la méthode des graphes suivent la méthode de synthèse logique et leur comportement est généralement décrit par un graphe d'états [CHU 87] énumérant exhaustivement les états atteignables. Le graphe d'états représente le fonctionnement désiré de l'implémentation et peut être mappé sur du matériel N'étant pas considérée dans cette approche de synthèse, la partie opérative (les chemins de données) est généralement générée manuellement (pour les chemins de données QDI ou SI) ou en utilisant les outils de CAO existants (pour les chemins de données avec des retards bornés).

Notons toutefois, que les graphes représentatifs des circuits sont soumis à des critères pour déduire leur caractère synthétisable ou non. A titre illustratif, les STG ne peuvent pas être directement synthétisés sans respecter un certain nombre de contraintes telles que la complétude du codage des états CSC (cf. §1.3.2.5).

Aussi, les méthodes basées sur les STG — Chu [CHU 87], Meng [MEN 89][MEN 91], Myers [MYE 92], Cortadela [COR 02] — possèdent un certain nombre d'handicaps telle que la limitation dans la description de l'opérateur de choix entre des entrées. Une autre restriction est qu'aucun signal ne peut avoir plus d'une transition montante et d'une transition descendante dans un réseau. Ces contraintes rendent le STG peu utilisable lors de description de systèmes complexes. Par ailleurs, ces méthodes ne comprennent pas de techniques systématiques pour ajouter des variables d'état afin de remplir la condition CSC.

Finalement et essentiellement, ces méthodes, par exemple celles de Chu et Meng, conservent le problème des aléas au niveau des portes. Elles génèrent des circuits « *customs* » en utilisant des portes complexes. Ce modèle de portes complexes fait l'hypothèse que la logique combinatoire est constituée de blocs monolithiques sans délai interne. Cependant, le réseau de portes obtenu peut en fait avoir des aléas.

Une autre méthode proposée par Beerel et Meng [BEE 92] consiste à synthétiser directement des circuits SI à partir des graphes d'états. Cette méthode évite les conditions syntaxiques des STG. Bien que la méthode produise des implémentations sans aléa au niveau des portes, la taille des portes générées est importante. Les techniques de décomposition actuelles pour décomposer ces portes en des portes de taille réaliste peuvent introduire des aléas.

En conclusion, même si elles produisent généralement des circuits efficaces et rapides, les méthodes de synthèse basées sur les graphes exigent souvent l'exploration complète de l'espace d'états pour trouver tous les états accessibles. Par conséquent, le nombre d'états accessibles de l'espace d'états explose rapidement quand la complexité et la taille de la spécification augmentent. De plus, comme les spécifications avec ces approches sont en général au niveau des signaux, l'écriture est ardue et est sujette à des erreurs surtout si la taille du circuit à concevoir est conséquente. Même si les problèmes d'affectation des états et de description des choix des entrées sont maîtrisés, le problème de la production des circuits sans aléa reste toujours un obstacle majeur à l'utilisation pratique de ces méthodes.

1.3.1.2 Spécification basée sur les langages de haut niveau

Les langages qui sont employés pour spécifier des circuits asynchrones incluent CHP [MAR 90], Occam [MAY 90][BRU 91], Tangram [BER 91][BER 93], Balsa [BAR 97] VHDL [ZHE 98] et Verilog [BLU 00]. Les méthodes de spécification des circuits asynchrones basées sur les langages présentent des avantages très importants.

Depuis une première spécification de haut niveau jusqu'à un niveau de description structurelle à grain fin, le circuit peut être décrit en utilisant un même langage. Ceci offre une continuité sémantique entre tous les niveaux de description, y compris les environnements de test, et constitue donc un outil puissant pour faire de l'exploration architecturale. Ainsi, les approches langages, en cachant les aspects liés à l'implémentation, permettent d'étudier des architectures tout en programmant. Ceci est couramment appelé « la programmation VLSI »

Les langages de description de matériel comme VHDL et Verilog -actuellement supportés par les outils commerciaux et largement adoptés par l'industrie- fournissent un niveau d'abstraction élevé et évitent de spécifier le séquencement des transitions de signaux. Néanmoins, ils n'utilisent pas le concept de canal de communication, tel qu'il est défini dans §1.2.3.1. Il faut donc spécifier des paquetages qui permettent au concepteur de définir la communication entre processus concurrents communicants, tel que présenté dans [REN 99][MYE 01].

A l'opposé, les autres langages sont dérivés du langage CSP. Ces langages utilisent le même concept que CSP, à savoir des processus concurrents communicants par passage de message via des canaux. Ceci offre au concepteur la facilité de décrire des blocs fonctionnels asynchrones communiquant concurremment et séquentiellement entre eux. Cependant, même si de nombreux langages basés sur CSP sont largement utilisés pour modéliser des circuits asynchrones, il n'existe pas aujourd'hui de réel consensus sur un unique langage de spécification dédié à leur modélisation. L'université Caltech a proposé le langage CHP. De son côté, Philips a défini le langage Tangram tandis que l'université de Manchester a développé le langage Balsa. Ces langages et leur méthode de conception sont abordés par la suite.

1.3.1.2.1 *Synthèse dirigée par la syntaxe*

Les méthodes de synthèse dirigées par la syntaxe sont utilisées pour traduire directement les structures du langage en blocs de circuit. Des exemples de ces méthodes sont proposées par van Berkel [BER 98], Brunvand [BRU 89], Ebergen [EBE 91], et Burns [BUR 88].

Dans la méthode de Brunvand, le langage OCCAM est utilisé pour décrire un système concurrent. Cette spécification est d'abord compilée en un circuit non optimisé en utilisant la synthèse dirigée par la syntaxe. Ce circuit est ensuite amélioré par les raffinements locaux automatisés similaires à l'optimisation «peephole» utilisée dans des compilateurs logiciels. Cette méthode génère des circuits dont le contrôle est DI et le chemin de données est de type «données groupées » en utilisant le protocole de communication deux phases.

Une approche similaire a été développée par Ebergen pour la conception de circuits DI purs. Les spécifications sont transcrites en des programmes dits « commandes », qui décrivent les séquencements des événements possibles. Ces commandes sont écrites dans un langage basé sur la théorie des traces. Une commande est raffinée à travers des décompositions en un réseau de composants DI. Comme dans la méthode de Brunvand, la communication dans cette approche est en deux phases.

En plus de leur simplicité et leur transparence, l'avantage de ces approches est leur capacité à décrire élégamment à haut niveau des systèmes concurrents, complexes et hiérarchiques et

donc d'éviter le problème d'explosion d'états en «traduisant » des structures de langage directement en des blocs de circuits fixes. Ces méthodes permettent d'une part la vérification formelle et d'autre part la synthèse des circuits possédant des comportements de sortie non déterministes en utilisant des arbitres et des synchroniseurs [BUR 88b].

Cependant, les circuits générés par ces méthodes peuvent être inefficaces et redondants puisque les optimisations sont difficiles à appliquer avec une synthèse dirigée par la syntaxe. En fait, comme ces approches reposent sur des transformations locales, elles manquent de flexibilité en vue d'une optimisation globale. De plus, il faut que le concepteur ait une bonne connaissance de la conception de circuits asynchrones et ceci même s'il est bon concepteur dans le domaine synchrone.

Finalement, il est important de noter que la synthèse dirigée par la syntaxe préserve la correction par construction, mais n'assure pas que le circuit global soit correct. Il est possible que le circuit présente un blocage (« *deadlock* ») par exemple.

1.3.1.2.2 *Synthèse par compilation successives*

Les circuits spécifiés par un langage sont également synthétisés par une autre approche proposée par Alain Martin dans [MAR 90b]. Cette approche traduit un programme de spécification en circuit par une série de transformations tout en préservant la sémantique. Cependant, elle a besoin de beaucoup d'interventions humaines pour être efficace. De plus, son automatisation, faite par Burns [BUR 88c] est très complexe. Cette méthode est détaillée en §1.3.2.3.

1.3.2 Méthodologies et outils actuels de synthèse de circuits asynchrones

Il existe aujourd'hui quelques outils de conception dévolus à la synthèse de circuits asynchrones mais aucun n'est commercialisé. Philips est le seul industriel à posséder un environnement de conception complet [VAN 93] pour les circuits asynchrones, mais il reste d'un usage privé. D'autres industriels ont développé des outils pour répondre à des développements ponctuels dans le domaine [MAR 94], [CHU 93]. Les laboratoires universitaires possèdent des outils de haut niveau mais qui souffrent d'une faible compatibilité avec les environnements commerciaux et qui trop souvent ne couvrent pas l'ensemble du cycle de conception. Leur usage permet cependant à la communauté universitaire de concevoir des circuits fonctionnels. Nous présentons dans ce qui suit quelques uns de ces outils commerciaux et universitaires.

1.3.2.1 Méthode Tangram (basée langage, dirigée syntaxe)

Tangram est un système propriétaire de Phillips [BER 88][BER 93] comprenant un langage de description de haut niveau (Tangram) et un compilateur associé qui permet de traduire les structures de langage en des structures de composants « poignée de main » (figure 1.16). Cet outil est bien connu dans la communauté asynchrone [BER 94][BER 95][GAG 98][KES 00][KES 00b]

La méthode de spécification de Tangram est basée sur le langage de haut niveau Tangram dont la syntaxe ressemble aux langages de programmation traditionnels tels que C, Pascal. Ce langage utilise les concepts de CSP et modélise des processus concurrents communicants par passage de messages synchrones via des canaux de communication point à point.

La méthode de synthèse de Tangram est dirigée par la syntaxe et utilise un format intermédiaire basé sur les circuits de type « poignée de main ». Cette méthode permet de traduire de façon directe un circuit asynchrone décrit en langage Tangram en une structure de composants «poignée de main».

Le back-end du flot de conception implique de disposer d'une bibliothèque de circuits de type « poignée de main» que le compilateur cible ainsi que de certains outils comme l'analyseur de performance et le simulateur fonctionnel. De nombreuses bibliothèques de circuits existent, ce qui permet les implémentations utilisant différents protocoles (4 phases « données groupées », 4 phases double-rails) et différentes technologies cibles (cellules standard CMOS, FPGA). Les circuits de type «poignée de main» dans les bibliothèques

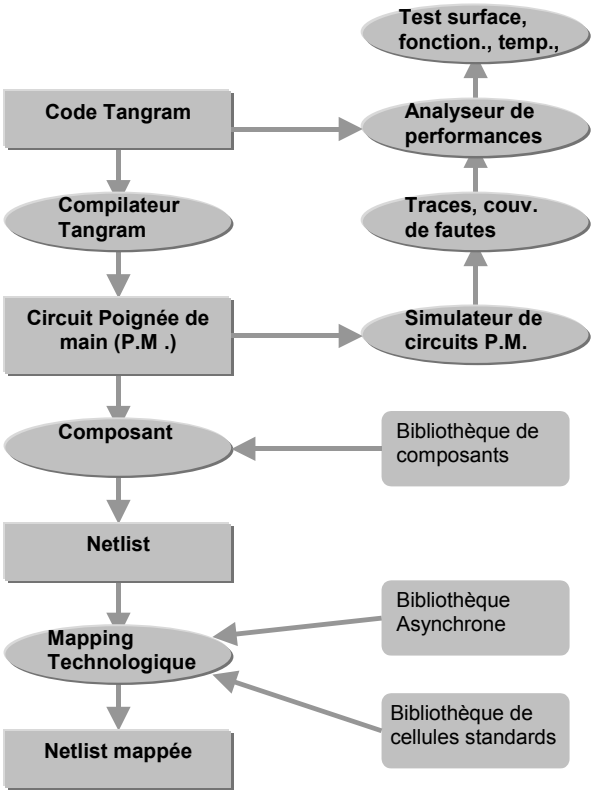


Figure 1.16 : Méthode de Tangram

peuvent être spécifiés et conçus soit manuellement, soit en utilisant les méthodes STG, soit en utilisant les étapes de la méthode de Caltech présentée ultérieurement.

Puisqu'il existe une correspondance un pour un entre une structure de langage et un circuit de type « poignée de main» généré, le processus de compilation de cette méthode est simple et entièrement transparent au concepteur: un changement progressif au niveau spécification a pour résultat un changement prévisible au niveau implémentation de circuit. L'optimisation dans la méthode Tangram consiste à remplacer les structures des composants «poignée de main» par des équivalents plus efficaces.

1.3.2.2 Méthode Balsa (basée langage, dirigée syntaxe)

Balsa est un outil de synthèse tout autant qu'un langage de description développé par l'université de Manchester. Cet outil qui emprunte la fonction de compilation de Tangram, traite des circuits asynchrones spécifiés par le langage de haut niveau Balsa (d'ailleurs largement adapté des langages de haut niveau CSP et OCCAM). Aussi adopte-t-il une

approche de synthèse dirigée par la syntaxe : les structures de langage sont « mappées » directement sur des composants communicants de type « poignée de main». Le flot de conception de cette approche est donné Figure 1.17.

Comme Tangram duquel il s'inspire, le langage Balsa [BAR 97][BAR 00] fait partie de la famille des langages de programmation concurrents. Il se base sur la communication synchronisée des canaux et le style de description parallèle de CSP.

Breeze est le format intermédiaire utilisé principalement dans le flot Balsa. Il agit tel une bibliothèque de « *netlists* » qui définie le réseau de circuits « poignée de main» et renforce ainsi l'indépendance des outils finaux par rapport aux outils frontaux. Dans le système Balsa, la simulation fonctionnelle est effectuée par LARD [END 94] un simulateur développé dans le cadre du projet Amulet. La simulation après synthèse est réalisée grâce à des outils commerciaux.

Cette méthode de conception a été illustrée par la conception du microprocesseur Amulet3i [BAR 00b].

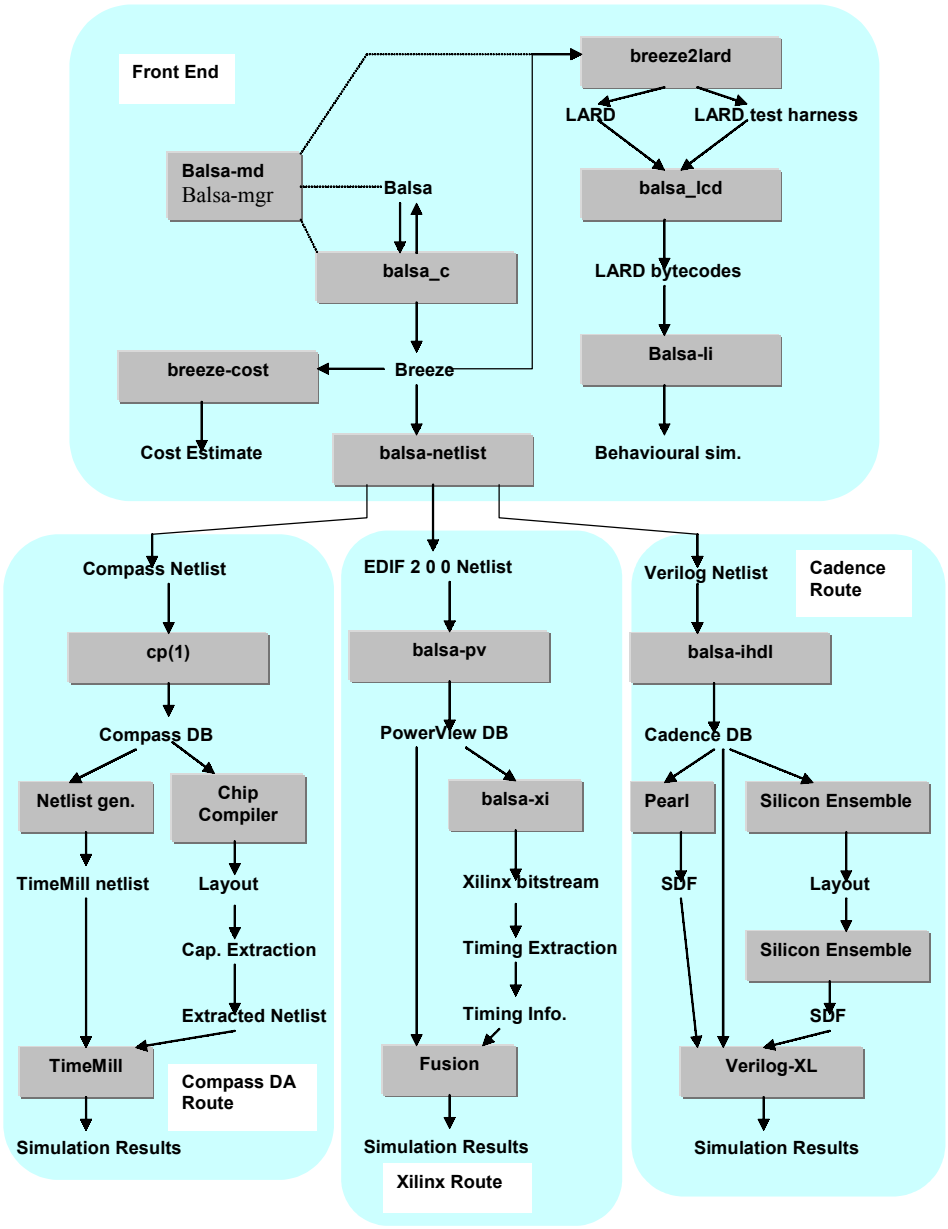


Figure 1.17 : Méthode Balsa

1.3.2.3 Méthode Caltech (basée langage, dirigée par compilation)

Universellement reconnue dans la communauté asynchrone comme étant l'une des voies de développement asynchrone les plus probantes, cette méthode repose aussi sur une description de haut niveau sous forme de processus concurrents communicants. Basée sur le langage CSP [HOA 78], cette modélisation CHP [MAR 90b] garantit depuis le début du développement « top» jusqu'à la fin « down » la préservation des clauses d'insensibilité aux délais imposées par le modèle : les processus dialoguent entre eux sans jamais faire d'hypothèse concernant la propagation des signaux le long des canaux. Le langage CHP a été principalement développé pour décrire un circuit correct (en termes d'hypothèses temporelles autant que de fonction implémentée) et préserver cette correction tout au long des transformations appliquées. La nécessité de modéliser l'ensemble des contraintes depuis le plus haut niveau à toutes les étapes du développement étant maintenant une certitude, CHP s'est imposé comme le langage idéal : rigoureux et complet.

Le principe de la méthode se présente comme suit (figure 1.18).

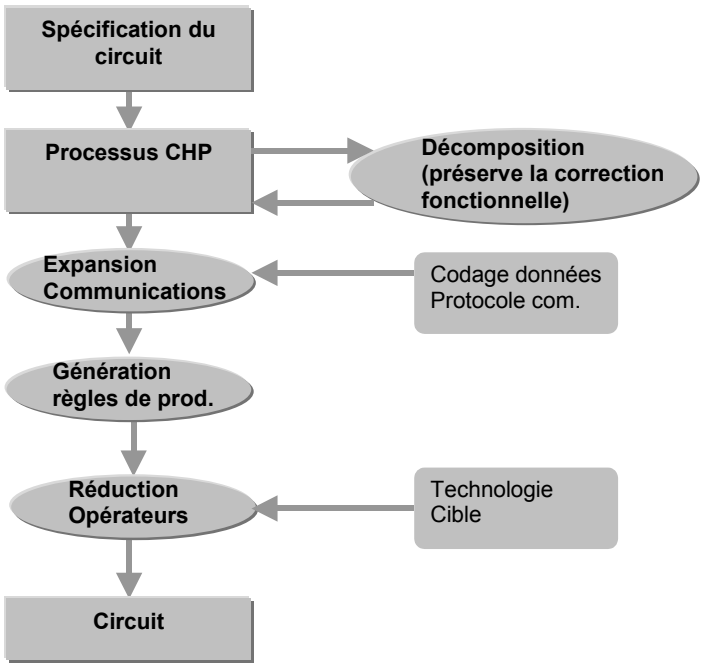


Figure 1.18 : Méthode Caltech

Décomposition des processus. La première étape de la procédure de synthèse est la décomposition des processus. Son but est de convertir chaque processus en des processus élémentaires de faible complexité. Ce faisant, on peut en particulier faire apparaître des processus communs. Réduire la complexité des processus facilite le reste de la procédure de synthèse, alors que faire apparaître les processus communs permettra de réduire la surface du circuit résultant. Cette phase de décomposition doit également permettre de faire apparaître la concurrence inhérente à l'algorithme à implémenter et aussi identifier tous les problèmes d'arbitrage et de synchronisation. Ces derniers feront l'objet d'une attention particulière lors de l'implémentation. Le programme résultant de cette phase de décomposition est une image très précise de l'architecture de la fonction à implémenter.

Expansion des communications (*Handshaking expansion*). Cette étape concerne les canaux de communication. Un canal de communication permet à deux processus d'échanger des données par une connexion point à point (il est également possible de concevoir des schémas de communication multi-points complexes). Afin de maintenir la correction du code initial, les protocoles de communications à travers les canaux doivent respecter certaines règles. Les protocoles deux phases ou quatre phases peuvent être indifféremment utilisés. La règle de base est que pour un canal de communication donné, un des deux ports terminaux est choisi « actif » alors que l'autre est choisi « passif ». Cette règle assure que deux processus connectés par un canal ne sont pas tous deux en attente d'une donnée. Ces situations causeraient bien sûr des blocages.

Une fois que le protocole est choisi et que le type d'action de communication (passif ou actif) est déterminé pour tous les ports de tous les processus, les actions de communication sont explicitées. On obtient alors un code composé d'instructions d'assignation de signaux et d'instructions d'attente ou de test sur des signaux. Ces instructions associées aux opérateurs « ; » constituent en fait le contrôle du circuit. Ce contrôle définit toute la synchronisation entre les processus du circuit.

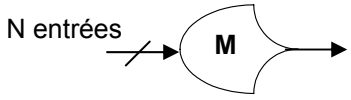
Règles de production (*production rules*). Nous disposons en entrée de cette phase d'un programme dont toutes les instructions sont implémentables à l'exception des directives de séquentialité « ; ». Cette étape consiste à supprimer les « ; » qui sont remplacés par un ensemble de règles de production.

A partir de ces règles de production il est possible de construire pour chaque signal, une cellule CMOS à une sortie et plusieurs entrées. La connexion de toutes ces cellules entre elles implémente le circuit. C'est au cours de cette phase qu'il faut se préoccuper de la localité des fourches isochrones. Il est possible d'appliquer des transformations sur le circuit de façon à déplacer les fourches isochrones. Ces transformations préservent bien sûr la correction fonctionnelle du circuit.

La méthode de synthèse de circuits asynchrones développée par Alain Martin à Caltech est certainement l'une des plus performantes à l'heure actuelle. Des circuits de complexités significatives ont d'ailleurs été conçus dont un microprocesseur 16 bit [MAR 89], le MIPS [MAR 97], et plus récemment le Lutonium [MAR 03]. Il a été démontré que la méthode pouvait également être appliquée à la conception de circuits en AsGa [TIE 94a], [TIE 94c] démontrant l'indépendance de l'approche vis-à-vis de la technologie.

1.3.2.4 Méthode NCL

«*Null Convention Logic*» [FAN 96][LIG 00] est une logique propriétaire proposée et brevetée par Theseus Logic. Elle est basée sur le codage 3 états et l'utilisation de portes à seuil («*threeshold gates*»), représentées Figure 1.20, qui sont en fait des portes de Muller généralisées. La « valeur ajoutée » se situe au niveau de l'optimisation logique et des possibilités de projection technologique supérieures qu'offrent ces portes généralisées.



Entrées	*Sortie*
N entrées valides	*1*
Toutes entrés nulles	*0*

Figure 1.20 : Fonctionnement d'une porte à seuil (M=<N)

La philosophie des circuits générés par la méthode NCL repose sur le motif de base représentée Figure 1.21. Il s'agit de séparer fortement la partie combinatoire (proche du circuit synchrone équivalent) et la partie acquittement/gestion de la validité des sorties (spécifique à l'asynchrone). Ainsi, l'apposition d'une «barrière de registre» en sortie de la partie combinatoire permet d'éviter de propager des données non valides. La génération de l'acquittement se fait dans des blocs logiques distincts dont la synthèse n'est pas automatisée.

Avec l'intention d'adapter la description de circuits asynchrones aux langages et outils existants, la description du circuit n'est pas basée sur un nouveau langage, pour lequel il serait

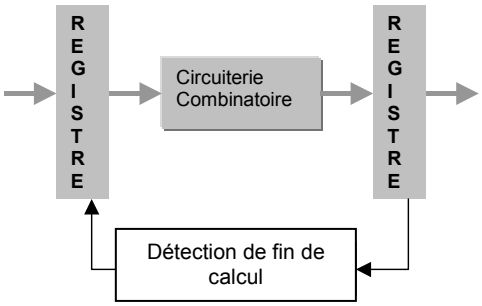


Figure 1.21 : Motif de base de circuit généré par méthode NCL

nécessaire de former des concepteurs, mais en VHDL (pour lequel il sera possible de tirer profit des compétences et outils existants déjà sur le marché). II faut néanmoins ajouter quelques directives spécifiques à l'approche Theseus pour modéliser des circuits asynchrones (complétion des données, codage).

La partie « circuiterie combinatoire » est décrite de manière classique (ou presque). En effet, la seule distinction repose dans les nouveaux types correspondant au codage de données (3 états) et les nouvelles définitions de portes (correspondant aux portes classiques « AND », « OR», etc.). C'est par la suite que le logiciel optimisera de manière automatique la fonction désirée tout en respectant les critères de circuits asynchrones (absence d'aléas, contraintes QDI, etc.).

Toutefois, le code VHDL tel qu'il est à ce stade ne peut être simulé sans introduire la notion de « rendez-vous ». Pour pallier à ce problème, une nouvelle procédure, appelée «*hysteresis*», dont la propriété est d'informer le simulateur sur les conditions à remplir pour produire chaque sortie, est proposée. Les registres sont explicitement spécifiés dans le code afin de réaliser des étages de pipeline.

Enfin, la génération des signaux d'acquittement - élément crucial des circuits asynchrones - doit se faire à la main, c'est-à-dire en instanciant les fonctions logiques dans le code VHDL sans bénéficier de garanties que les conditions d'acquittement sont correctement gérées. Ce point faible est crucial dans la mesure où la grande majorité des méthodologies de synthèse des circuits asynchrones gère d'une manière ou d'une autre les signaux d'acquittement selon des règles bien définie. Souvent cela représente plus de travail que la partie combinatoire elle-même.

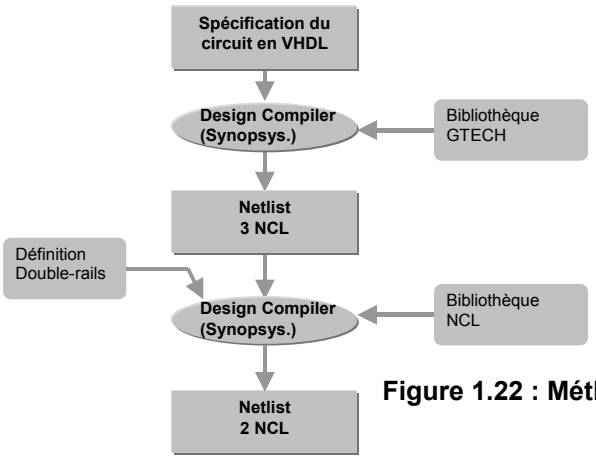


Figure 1.22 : Méthode NCL

Dans la méthode NCL, la conception d'un circuit asynchrone décrit en VHDL est synthétisée par l'outil de synthèse commercial Synopsys comme montrée Figure 1.22. La synthèse [LIG 00] est réalisée en deux étapes :

- La description VHDL est synthétisée et optimisée par l'outil de synthèse commercial « Design Compiler ». Le résultat de cette étape est un réseau de portes de base (« AND », «OR», « INV ») qui décrit le flot de données du circuit.
- La deuxième étape consiste à coder tous les signaux et les portes en double-rails en utilisant toujours un outil de synthèse commercial. Cette étape effectue une optimisation de type ASIC et puis une projection technologique à l'aide d'une bibliothèque de cellules de portes à seuil.

Une fonction principale, pour laquelle Theseus pourrait réellement apporter une nouveauté sur le marché, concerne la vérification des fourches isochrones [KON 02]. Il s'agit de détecter les nœuds susceptibles de constituer des fourches isochrones. La tâche est bien connue comme étant complexe et donc nécessitant beaucoup de ressources de calcul, si bien que peu de techniques permettent à ce jour de donner rapidement la liste de toutes les fourches isochrones d'un circuit, à fortiori si ce dernier se révèle complexe. Cependant, un certain nombre de restrictions rendent cette vérification plutôt incomplète. Tout d'abord, cette analyse n'est possible que sur les parties combinatoires et non les acquittements. Cette restriction est assez inattendue à ce point car la méthode est sensiblement la même concernant les deux cas. En effet, les contraintes QDI sont cruciales autant dans la partie combinatoire que dans la partie acquittement. Il faut sans doute supposer que, comme pour la synthèse, le modèle de Theseus garantit des arbres d'acquittement simplistes et QDI par construction.

Cependant avec Tangram c'est la seule méthode de conception de circuits asynchrones utilisée et développée aujourd'hui par une société privée même si elle possède un certain nombre de limitations.

- Le style de conception proposé est très typé. Il apparaît que non seulement cette description générique n'est pas universelle, mais de plus, elle est limitée à plus d'un point de vue. En effet, si la distinction opérée entre logique directe et logique d'acquittement semble permettre une relative simplification, il ne s'agit pas moins d'une très importante restriction: il n'est dès lors plus possible de générer des acquittements partiels (n'acquitter que certaines branches du circuit), ni de procéder à toutes les combinaisons complexes où l'acquittement dépend du chemin suivi par les données dans la partie logique combinatoire. Les implémentations de type «lecture conditionnelle» (comme la fonction de multiplexage) ne sont pas synthétisables par la méthode NCL.

- Comme on l'a vu, la méthode NCL n'est pas basée - contrairement à Tangram, Balsa et Caltech - sur la description de communication mais bien sur une forme de communication simplifiée dans laquelle le concepteur a pour charge d'implémenter « manuellement » les acquittements selon le modèle souhaité. Ce qui pourrait à première vue sembler une liberté de conception se révèle surtout un manque de flexibilité et de robustesse vis-à-vis de la spécification « haut niveau ». En effet, il n'est nulle part possible de vérifier si la génération des signaux d'acquittement est correcte ni qu'elle réalise le schéma de communication souhaité. D'autant plus que la valeur ajoutée de la génération manuelle des acquittements est nulle. En effet, on ne fait que réaliser un protocole dont la description serait relativement aisée avec un modèle haut niveau.

1.3.2.5 Méthode Petrify (basée graphes (STG), synthèse « logique »)

Cette méthode proposée par Cortadella [COR 02] traite la conception des circuits de type « SI » (indépendant de la vitesse) spécifiés par un graphe de transitions de signaux (STG). La Figure 1.23 représente le flot de conception proposé.

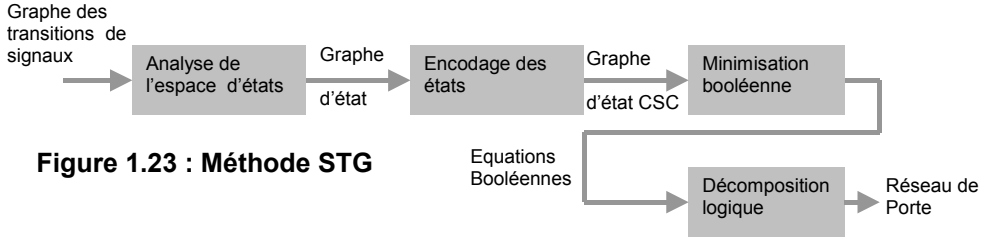


Figure 1.23 : Méthode STG

Basée sur le réseau de Petri [PET 62][YAK 00] le modèle STG [CHU87] servant à spécifier le circuit asynchrone est une re-formalisation du diagramme temporel (un diagramme temporel spécifie les relations de causalité entre les événements des signaux). Il permet de modéliser la concurrence et une version limitée de choix entre des entrées. Un STG, dont un exemple est donné Figure 1.24, est en fait un réseau de Petri limité par les caractéristiques suivantes :

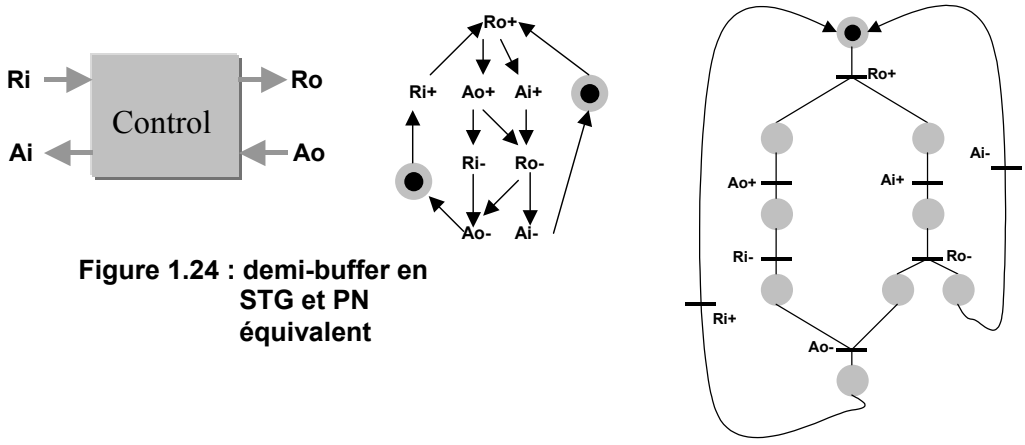


Figure 1.24 : demi-buffer en STG et PN équivalent

- Liberté de choix : la sélection entre des alternatives doit être seulement contrôlée par les entrées mutuellement exclusives.
- 1-borné : il n'existe jamais plus de deux jetons dans une place
- Vivacité : STG sans blocage

Dans cette méthode, un circuit asynchrone SI est décrit par un STG si ce dernier possède les caractéristiques suivantes

- Cohérence : les transitions d'un signal doivent strictement alterner entre la montée et la descente dans n'importe quelle exécution du STG
- Persistance : si une transition de signal est autorisée, elle doit avoir lieu, c'est-à-dire qu'elle ne peut pas être désactivée par une autre transition. La persistance des signaux internes et des signaux de sortie doit être garantie par le STG, tandis que celle des signaux d'entrées est assurée par l'environnement.

La spécification STG décrit des relations de causalité entre les événements des transitions de signaux. Pour calculer la fonction de chaque sortie du circuit - la tâche de synthèse de circuit - l'espace d'états atteignables [MUL 59] doit être produit. Le nombre d'états de l'espace d'états augmente exponentiellement avec le nombre de signaux du STG. C'est le principal point faible de cette méthode qui en pratique limite le nombre de signaux à une vingtaine. En utilisant la théorie des régions [KIS 94] sur cet espace d'états, les équations logiques booléennes des signaux internes et des signaux de sortie sont calculées.

Un problème d'ambiguïté peut se poser lors du calcul des fonctions de signaux : il peut exister des états différents ayant le même codage. En fait, ce phénomène apparaît quand les seuls signaux ne suffisent pas pour identifier tous les états. Dans ce cas, le système viole la propriété CSC (« Complete State Coding »). Il est difficile pour l'outil de synthèse de garantir la propriété CSC. La solution consiste à ajouter des variables d'état supplémentaires de façon à ce que les différentes combinaisons de codage correspondent aux différents états.

L'implémentation matérielle du circuit est ensuite déduite en respectant des contraintes technologiques. Dans le style semi-custom, les circuits doivent être construits avec les portes d'une bibliothèque spécifique de cellules, tandis que dans le style custom, les circuits sont composés des portes complexes et la complexité de ces portes est déterminée par des contraintes telles que l'entrance maximale. Chaque équation booléenne générée après optimisation doit être implémentée par un ensemble des portes. Il faut alors résoudre le problème de la décomposition logique tout en préservant le fonctionnement correct du circuit (n'introduisant pas des comportements non désirés dans le système). C'est une tâche très difficile à réaliser dans les outils de synthèse des circuits asynchrones. Cela demande beaucoup de ressources en termes de temps de calcul.

En résumé, la conception de circuits asynchrones SI avec la méthode basée sur les STG implique les étapes suivantes.

1. Décrire le comportement souhaité du circuit et de son environnement par un STG

2. Vérifier si cette représentation satisfait les caractéristiques d'un STG décrivant des circuits SI (la liberté de choix, 1-borné, la vivacité, la cohérence et la persistance)

3. Vérifier si ce STG est synthétisable (satisfait la condition CSC).

4. Choisir un modèle d'implémentation et calculer les équations booléennes pour les fonctions de charge et de décharge. A partir de ces équations, l'implémentation du circuit est effectuée en utilisant des portes atomiques telles que les portes complexes AO ou des portes de base plus simple.

5. Modifier cette implémentation pour que le circuit puisse être forcé dans l'état initial souhaité en utilisant le signal de reset ou les signaux d'initialisation.

Cette méthode est automatisée par l'outil de synthèse Petrify [COR 97]. Cet outil prend en entrée un STG décrivant le circuit, qui peut être saisi sous une forme texte ou graphique. La

description de circuits asynchrones sous forme d'un STG est une tâche ardue et sujette aux erreurs, notamment pour les concepteurs synchrones. La synthèse d'un tel STG nécessite l'exploration de tous les états possibles, ce qui limite la complexité des problèmes pouvant être réalisés.

1.3.2.6 Méthode Minimalist (basée graphes (ASM), « logique »)

Cette méthode, proposée par Nowick [NOW 93] traite la conception de contrôleurs asynchrones fonctionnant en mode rafale. La spécification utilisée dans cette méthode est une machine à états de type Mealy. Du point de vue du calcul, cette spécification se base sur les états. A chaque état, la machine peut recevoir des entrées, génère des sorties et avance jusqu'à l'état suivant (une telle spécification peut être également décrite en un tableau de flots [UNG 69]. A titre d'exemple, la Figure 1.25 représente un exemple de la spécification de contrôleurs fonctionnant en mode rafale. Cette spécification est plus concise que le STG, notamment quand la concurrence du système est importante.

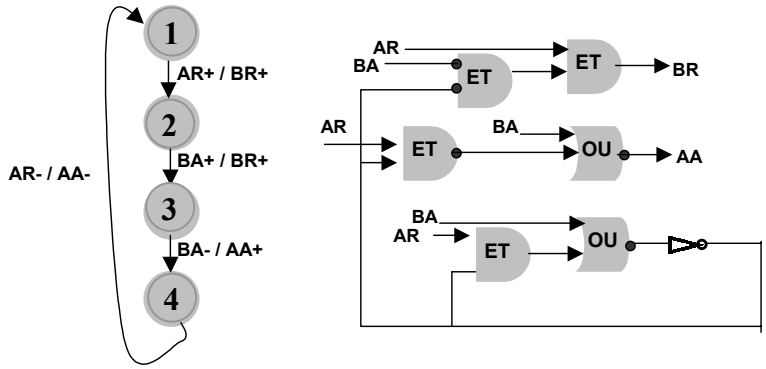


Figure 1.25 : Spécification en mode rafale et son implémentation

Des contraintes sur la spécification sont définies pour que le contrôleur fonctionne correctement. D'abord, à chaque état donné, aucun changement des entrées ne peut être couvert par d'autres changements des entrées puisque dans ce cas le comportement du contrôleur pourrait être ambigu. La deuxième restriction est que chaque état a un point d'entrée unique, ce qui simplifie la minimisation et garantit une implémentation sans aléa. Une autre restriction sur la spécification est qu'elle ne permet pas un changement d'état sans un changement sur des entrées. Cela signifie que le système reste dans un état stable si aucune entrée ne change. Ces restrictions distinguent cette méthode de spécification de la spécification pilotée par des événements, telle que celle développée par Davis [DAV 93 b].

Nowick a également proposé une méthode de synthèse, à partir de cette spécification en mode rafale, basée sur une implémentation de type machine à états synchronisée localement (*locally locked state machine*). Intrinsèquement, cette machine est une machine de Huffman ajoutant une unité d'auto-synchronisation, qui fonctionne comme une horloge locale sur des verrous. Cette machine est appelée machine auto-synchronisée (*self-synchronized)* (figure 1.26). Les caractéristiques suivantes font que la machine proposée par Nowick est différente des machines de Huffman.

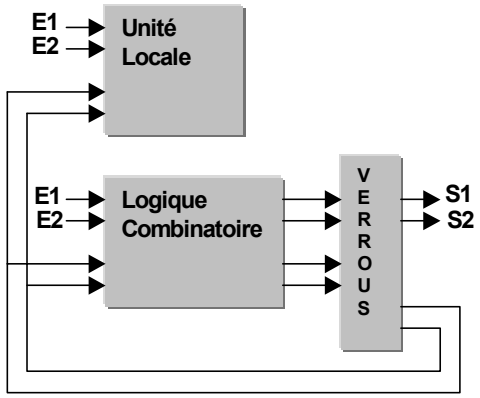


Figure 1.26 : Schéma de la machine auto-synchronisée

- L'horloge est générée de manière sélective certaines transitions ne nécessitent pas d'horloge.
- L'unité d'horloge n'utilise pas de modèle de délais «inertiel » pour éliminer les aléas.
- La logique combinatoire sans aléa est requise pour obtenir une certaine transition.

La méthode de synthèse se compose des étapes suivantes. La première étape est de générer, à partir de la spécification en mode rafale, un tableau de flots pour les sorties et les prochains états. Les états dans ce tableau sont ensuite minimisés et affectés par des codes d'état uniques. La dernière étape consiste à générer les fonctions booléennes pour l'horloge, chaque sortie et chaque variable d'état. Ces fonctions booléennes subissent une minimisation logique, ce qui permet une implémentation sans aléa.

Un outil de synthèse, appelé Minimalist [FUH 99] a été développé pour prototyper cette méthode. Grâce à cet outil de synthèse, les contrôleurs asynchrones sont générés en utilisant des éléments C généralisés.

Bien que cette méthode ait été appliquée dans la conception de contrôleur STRiP [DEA 92], il reste encore des problèmes à traiter :

- Le fonctionnement des circuits générés est limité par le mode rafale
- La spécification en mode rafale est pratique pour décrire les systèmes dont la taille est petite ou moyenne. Il est difficile de décrire un grand système concurrent par cette méthode de spécification.
- Le fonctionnement sophistiqué et complexe des circuits générés nécessite un test. Tester une telle implémentation consiste à tester la logique combinatoire autant que les délais dans les différents chemins afin de s'assurer que les contraintes temporelles sont respectées. Ce type de test n'est pas facile à réaliser.

1.4 Conclusion

La conception asynchrone se propose donc de pallier à un certain nombre de failles de la conception synchrone essentiellement dues aux problèmes liés à l'horloge. Nous avons souligné les différents avantages que recèle potentiellement l'implémentation asynchrone tel que le calcul en temps minimum, le caractère élastique du pipeline, la fiabilité pour les traitements non déterministes, le faible bruit (électromagnétique), la faible consommation, et la migration. L'ensemble de ces avantages découle du caractère local du contrôle des circuits asynchrones.

Les différents types de circuits asynchrones sont classés en fonction du modèle de délais dans les fils et dans les portes, et les différentes méthodologies de conception (spécification et synthèse) de circuits asynchrones sont présentées. Les circuits spécifiés par les méthodes basées sur les langages haut niveau sont synthétisés le plus souvent par l'approche orientée syntaxe. Un cas particulier très connu de méthode basée langage est synthétisé par compilations successives (expansion des communications et ré-ordonnancement, génération des règles de production). Les circuits spécifiés par les méthodes basées sur les graphes sont essentiellement compilés par la synthèse logique (transformation de graphes, équations logiques). Il apparaît qu'aucune méthodologie particulière ne cible tous les problèmes de la conception asynchrone.

Dans TAST, nous avons voulu allier l'avantage offert par une description des circuits en langage haut niveau (conception hiérarchique de circuits complexes) à l'avantage de la spécification des circuits par la méthode des graphes (synthèse facilitée). De la sorte nous spécifions nos circuits en langage de haut niveau dit CHP étendu, puis cette représentation est compilée en graphes (réseaux de Petri et graphes de flots de données) exploités pour la synthèse.

CHAPITRE II

2. Spécification synthétisable de circuits asynchrones

Nous avons distingué dans le chapitre traitant de l'état de l'art, deux méthodologies majeures pour la spécification et la représentation des circuits asynchrones, la première basée sur les graphes et la seconde basée sur les langages de haut niveau.

L'idée de combiner ces deux approches tire sa justification du fait que chacune de ces méthodes lève l'inconvénient de l'autre. En effet, les méthodes basées sur les graphes sont généralement préférées pour l'analyse temporelle et la synthèse, mais le concepteur doit fournir un effort pénible pour décrire une spécification en graphes. A contrariori, les méthodes basées sur les langages de haut niveau sont expressives et adéquates pour décrire des systèmes complexes avec une structure hiérarchique et modulaire sans s'encombrer des détails de l'implémentation (protocole de communication, codage, modèle de circuits cibles). Toutefois le circuit synthétisé à partir de cette spécification n'est pas optimal.

Il reste que les spécifications en langage de haut niveau ne peuvent pas être systématiquement synthétisées car, souvent, les outils de synthèse et les technologies cibles imposent des limitations. C'est pourquoi, il s'agit à ce niveau de définir l'ensemble des spécifications synthétisables en définissant un sous-ensemble du langage de haut niveau.

Au cours de ce chapitre, nous présentons d'abord le CHP (« Communication Hardware Processes »), langage de description de matériel servant à spécifier les circuits asynchrones. Ensuite, nous introduisons les PN-DFG (*Petri Net - Data Flow Graph*) qui permettent de représenter un circuit asynchrone sous forme de graphes. En plus de servir à la vérification formelle de circuits, cette combinaison de graphes permet également de découpler le *front-end* du circuit de son *back end*. Enfin, nous concluons par la proposition d'une spécification synthétisable de circuits asynchrones.

2.1 Le langage CHP (Communicating Hardware Process)

2.1.1 Adéquation de la sémantique du langage CHP avec la synthèse des circuits asynchrones

Un circuit asynchrone est un ensemble de blocs fonctionnels communicant entre eux via des canaux. En conséquence, le formalisme adéquat pour la description de ce genre de circuit doit englober la notion de processus concurrents qui communiquent mutuellement par des passages explicites de messages. Cette approche favorise en effet la transposition de la spécification fonctionnelle tout en gardant une forte lisibilité de la structure matérielle sous-jacente.

Un formalisme existant qui semble répondre à ces exigences est le langage CHP [MAR 90] qui résulte du travail d'Alain Martin. Ce dernier s'est inspiré du langage CSP (*Communicating Sequential Process*) introduit par Hoare [HOA 78] pour décrire des systèmes parallèles sous forme de processus séquentiels communicants ainsi que de commandes gardées de Djikstra [DJI 76].

Toutefois, la description de systèmes complexes à l'aide du CHP d'Alain Martin n'est pas aisée dans le sens où ce langage ne fournit pas les facilités de langages de haut niveau tels que Verilog, VHDL ou encore System C. A titre d'illustration, le type booléen est le seul type de données qu'il propose. L'ensemble des autres types peut être créé par l'association de

variables booléennes. Egalement, l'usage d'opérateurs sur des types autres que booléens fait implicitement référence à des fonctions qui traitent des booléens.

Le recours au langage VHDL par exemple, n'est cependant pas approprié car il est sémantiquement synchrone.

C'est pourquoi, le groupe CIS a étendu le CHP d'Alain Martin pour faciliter la tâche du concepteur, et rendre le langage plus puissant et plus lisible. La valeur ajoutée en terme de « puissance » transparaît par exemple dans l'ajout de types et la possibilité d'effectuer des calculs arithmétiques avec des opérandes de types signés et non signés en multi-rails « 1-parmi-N ». La meilleure lisibilité se traduit entre autre par la modélisation structurelle qui supplante la déclaration à « plat » de tous les processus. Egalement, la modélisation hiérarchique permet de pouvoir gérer la complexité des systèmes VLSI. Cette hiérarchie est simplement basée sur les composants dont l'interconnexion repose uniquement sur les canaux.

2.1.2 Modélisation structurelle du code CHP

Nous avons souligné plus haut que dans le CHP initial d'Alain Martin [MAR 90] l'ensemble des processus était déclaré à plat. La complexité des systèmes VLSI a imposé de définir une approche de modélisation hiérarchique. Cette hiérarchie est seulement basée sur les processus et les composants connectés via des canaux (et non des signaux), ce qui est moins modulaire et moins riche qu'en VHDL où il est nécessaire de déclarer à la fois entité, architecture, composant, processus, et configuration.

A titre indicatif, un programme CHP ressemble à ceci :

```
COMPONENT buffer
PORT (e : IN DR ;
      s : OUT DR)
BEGIN
 PROCESS buffer_proc
 PORT (e : IN DR ;
       s : OUT DR)
 VARIABLE x : DR;
 BEGIN
  [e?x ; s!x ; LOOP];
 END;
END;
```

Ce programme décrit un circuit tampon ou « *buffer* » qui lit en boucle une donnée sur le port d'entrée « e » puis restitue cette donnée le port de sortie « s ».

2.1.2.1 Le composant

Le composant constitue la vue globale d'un circuit asynchrone et sa vue externe déclare éventuellement des ports de communication. Sa partie déclarative peut accessoirement contenir des déclarations de constantes, de canaux de communications permettant de connecter les différents processus et instances de composants, ou encore de composants locaux destinés à être instanciés. Le corps d'un composant comporte des instructions concurrentes telles que des déclarations de processus et des instances de composants.

```
COMPONENT nom_du_composant

PORT      (liste_de_ports) ;
CONSTANT  (liste_de_constantes) ;
CHANNEL   (liste_de_canaux) ;
COMPONENT (liste_de_composants_locaux);

BEGIN

 (instances_des_composants_locaux)
 (déclaration_de_processus)

END;
```

La partie déclarative d'un composant est optionnelle. Si un port est déclaré, il doit obligatoirement contenir le nom du port, sa direction, ainsi que le type de données qui y transitent, et accessoirement le type de codage ainsi que le protocole de communication des données.

Dans les circuits micropipeline de Yvan E. Sutherland [SUT 89], les données sont codées en binaire et on associe au bus de données, un signal de requête et un signal d'acquittement (codage *Bundled Data*). Chaque digit est représenté par sa représentation binaire. Dans les circuits quasi-insensibles aux délais [MAR 90], les données sont codées en 1 parmi N et on associe au bus de données, un signal d'acquittement. Le codage est dit DI (*Delay Insensitive*).

2.1.2.2 Instances de composants locaux

Les ports utilisés dans un processus ou un composant instancié sont soit ceux déclarés au niveau du composant global soit les canaux déclarés dans le composant courant. Dans tous les cas, ils doivent déjà avoir été déclarés avant d'être utilisés et leurs types doivent être les mêmes partout.

2.1.2.3 Processus

Les noms des différents processus doivent être distincts mais un processus peut avoir le même nom que le composant global dans lequel il est défini. Par ailleurs et contrairement à un processus en VHDL, un processus en CHP doit forcément déclarer ses ports de communications pour permettre de déclarer proprement les attributs de canaux (codage, direction, et protocole) et effectuer les vérifications nécessaires. Les variables locales d'un processus sont définies dans sa partie déclarative. Le corps d'un processus contient des instructions séquentielles mais également concurrentes ce qui est prohibé dans le langage VHDL par exemple.

```
PROCESSUS nom_du_processus

PORT      (liste_de_ports) ;
VARIABLE  (liste_de_variables) ;

BEGIN
(instructions_séquentielles_et_concur-
 rentes)
END;
```

2.1.2.3.1 Port de processus

La syntaxe est équivalente à celle de la déclaration d'un port de composant. Les identificateurs des ports du processus doivent avoir déjà été déclarés soit au niveau du composant global soit au niveau de la déclaration des canaux. Aucun port ne peut donc être utilisé dans la liste d'instruction du processus s'il n'a pas été déclaré dans la partie déclarative du processus (alors qu'en VHDL par exemple, un port peut être utilisé dans un processus dès lors qu'il a été déclaré au niveau global). Cette re-déclaration forcée permet à l'utilisateur de savoir quels ports du composant global sont utilisés dans chaque processus.

2.1.2.3.2 Variable de processus

Seul le type SR est interdit pour la déclaration de variables car il ne porte pas d'information. Une variable est déclarée dans un processus. Elle est locale à ce processus, et ne peut en aucun cas être accédée par un autre processus. Par conséquent, deux variables déclarées dans deux processus différents peuvent avoir le même nom. Une variable peut être initialisée.

2.1.2.3.3 Liste d'instructions de processus

L'ensemble des instructions d'un processus est exécuté une et une seule fois du début à la fin. Dans un processus les instructions peuvent être séquentielles « ; » ou parallèles « , ». Les instructions parallèles sont plus prioritaires que les instructions séquentielles mais le concepteur peut avoir recours au « *parenthésage* » pour imposer l'ordre des priorités. La dernière instruction d'un processus doit se terminer par un « ; ». Les structures de contrôle en CHP étendu sont explicités en §2.1.2.4. Les actions de communications sont explicitées en §2.1.3.4.3.

2.1.2.4 Structures de contrôle

En CHP, la modélisation du contrôle (instruction conditionnelle, boucle, sélection) emprunte la syntaxe des commandes gardées de Djikstra [DJI 76].

Le contrôle est représenté par des commandes gardées et s'exprime comme suit « **[<garde> => <liste_d_instructions> ; (LOOP|BREAK)] ;** ». Sémantiquement cette syntaxe signifie que la liste d'instructions ne s'exécute que si la garde correspondante est vraie. La garde étant une expression booléenne, toute expression retournant un booléen peut être utilisée, à l'image de l'opérateur de sonde de canal (ou probe) « # », des opérateurs de comparaison, des opérateurs logiques dont les opérandes sont des booléens, et du mot réservé « **OTHERS** », qui prend en compte tous les cas possibles non explicitement écrits.

Lorsqu'aucune garde n'est explicitée, le booléen considéré est « *TRUE* » et on simplifie l'écriture par « **[< liste_d_instructions > ; (LOOP|BREAK)] ;** »

Avec ou sans garde, deux types de structures de contrôle sont définis en CHP. Les mots clés définissant ces deux structures sont « *BREAK* » et « *LOOP* ».

Dans les instructions sans garde, le mot clé «*BREAK* » renvoie à une structure de contrôle dans laquelle la liste des instructions est systématiquement exécutée, après quoi l'on passe à la suite du programme. Par ailleurs, le mot clé « *LOOP* » exprime une structure de contrôle dans laquelle la liste des instructions est systématiquement exécutée, après quoi le processus continue de boucler ré-exécutant ainsi la même liste d'instructions.

Dans le cas où des gardes existent, la structure est dite de ***sélection déterministe*** (@) ou ***indéterministe*** (@@) et s'écrit comme suit :

```
@{@}[ garde_1  =>     liste_d_instructions_1  ; (BREAK|LOOP)
      garde_2  =>     liste_d_instructions_2  ; (BREAK|LOOP)
     {garde_i  =>     liste_d_instructions_i  ; (BREAK|LOOP)}
    ]
```

Ici, le mot clé « *BREAK* » renvoie à une structure de contrôle dans laquelle le processus est suspendu jusqu'à ce qu'une garde de la structure soit vraie. Dès lors qu'une garde est vraie, la liste des instructions qui lui est associée peut alors être exécutée et on passe à la suite du programme. Par ailleurs, le mot clé « *LOOP* » exprime une structure de contrôle dans laquelle la liste des instructions associées à une garde vraie est exécutée mais le processus continue de boucler tant qu'une garde est vraie exécutant ainsi les instructions correspondantes à cette garde.

Notons que dans la sélection déterministe, les gardes sont mutuellement exclusives. Ainsi une seule garde est vraie à la fois. Si plusieurs gardes son vraies simultanément, l'ambiguïté ne peut être résolue et une erreur est renvoyée. Par ailleurs, il appartient au concepteur de prévoir toutes les gardes possibles.

On peut donc constater que la modélisation structurelle offerte par cette syntaxe est suffisamment riche pour envisager l'écriture de problèmes VLSI complexes. A titre comparatif, les notions VHDL de configuration et de couple entité/architecture ne sont pas disponibles en CHP mais peuvent s'implémenter avec une politique de renommage des composants. Egalement, les possibilités de synchronisation/parallélisme en CHP sont plus riches qu'en VHDL sachant que les processus peuvent inclure des instructions concurrentes. Aussi, notons comme cela a été souligné plus haut, qu'en CHP les ports des processus sont déclarés. Cela assure une meilleure lisibilité structurelle et fonctionnelle de chaque processus.

2.1.3 Eléments de syntaxe

2.1.3.1 Conventions lexicales

La typologie, les commentaires, les caractères permis, les identificateurs, et les mots clés du langage CHP sont déclinés en annexe A1. Notons juste que tous les objets CHP (composants, variables, canaux, …) sont repérés par un unique identificateur obligatoirement différent des mots clés du langage.

2.1.3.2 Type de données

Un entier N est représenté en précision arbitraire par $''d_{L-1}, d_{L-2}, .., d_0''$[B][L]. Cette écriture signifie vecteur de « L » digits représentés dans la base « B ». La longueur L et la base B sont des nombres naturels. Le type de données peut être signé ou non signé.

2.1.3.2.1 Type de données non signé

La valeur d'un entier non signé selon l'écriture définie précédemment est donnée par

$$N = \sum_{i=0}^{L-1} d_i B^i$$

Les types sont présentés selon l'ordre de complexité (valeur simple, vecteur, ou tableau). Deux types de base existent en CHP étendu, un type simple et un type vecteur.

Le type simple « **MR[B]** » est un type de Multi-Rails en base B représentant un nombre entre 0 et (B-1), codé en « 1 parmi n ».

Exemple :

```
VARIABLE  b : MR[B] ;     -- déclaration d'une variable b avec le type MR
          b := "x"[B];    -- affectation de b avec 0 <= x < B
```

Le type vecteur « **MR[B][L]** » représente un vecteur de L éléments de type Multi-Rails en base B. Une variable de ce type correspond à un nombre entre 0 et (B^L- 1).

Exemple :

```
VARIABLE  b : MR[B][3] ;    -- déclaration d'un vecteur de 3 éléments en base B
          b := "x.y.z"[B];  -- affectation du vecteur b avec  0 <= x , y, z< B
```

Plus de détails sur ces types de données non signé ainsi que d'autres types de données basés sur les types non signés précédents ont été définis et sont explicités en annexe. Ils introduisent plus de facilité pour les concepteurs. Singulièrement notons le type « SR » (équivalent à MR[1]) qui ne sert qu'à synchroniser des processus communicants (il n'est pas utilisé pour des variables).

Un troisième type "tableau" (**MR[B][L][D]**) peut être utilisé uniquement dans la déclaration de variables et de constantes. Les ports et les canaux ne peuvent être déclarés en type tableau pour des raisons technologiques. En effet, le protocole de communication implique un seul acquittement par canal de communication. Or si un canal est déclaré en tableau et que ses différents éléments sont connectés à différents ports, quel port devra alors arbitrer l'acquittement et que feront les autres ports en parallèle ? Tous les types de CHP sont représentables en tableau, excepté SR.

2.1.3.2.2 Type de données signé

La valeur d'un entier signé selon l'écriture définie précédemment est donnée par

$$N = \begin{cases} \sum_{i=0}^{L-1} d_i B^i & \text{si } d_{L-1} < B/2 \\ (d_{L-1} - B)\ B^{L-1} \sum_{i=0}^{L-2} d_i B^i & \text{si } d_{L-1} >= B/2 \end{cases}$$

Cette écriture signifie que seul le digit de poids fort (le plus à gauche) porte le signe. Les autres digits se comportent comme des digits non signés.

En outre tous les types de données non signés présentés plus haut (sauf le type SR) possèdent leurs équivalents en type signé. Les deux types de base sont le type simple et le type vecteur.

Le type simple (**SMR[B]**) est un type de Multi-Rails signé en base B. Si B est pair, ce type représente un nombre compris dans l'intervalle [–(B/2), (B/2)-1]. Si B est impair, il représente un nombre compris dans l'intervalle [-(B-1)/2 , (B-1)/2].

Exemple:

```
VARIABLE  b : SMR[4];          -- déclaration d'une variable
          b := "2"[4];         --affectation de b :  b = 2 – 4 = -2
```

Le type vecteur (**SMR[B][L]**) représente un vecteur de L éléments « Multi-Rail » signés en base B.

Exemple :

```
VARIABLE  b : SMR[3][3] ;    -- déclaration de b
          b := "2.2.2"[3];   -- affectation : b=(-1)*3² + 2* 3¹ + 2* 3⁰  = -1
```

Les autres types de données signés définis sont basés sur les types signés de base précédents. Ils introduisent plus de facilité pour les concepteurs. Ils sont reportés en annexe.

2.1.3.3 Conversion de types de données

Les données manipulées en CHP sont typées. Cela signifie que pour chaque donnée que l'on utilise il faut préciser le type de donnée, ce qui permet de connaître l'occupation mémoire de la donnée ainsi que sa représentation. Les deux opérandes d'une opération logique, arithmétique ou encore d'une action de communication doivent être du même type. L'affectation est la seule opération qui permet de convertir les types de données et les rendre compatibles. Le type de données de la partie droite de l'affectation est toujours converti au type de données de la partie gauche. A titre d'illustration si une variable « x » est déclarée avec le type DR et une variable « y » avec le type MR[3] alors l'affectation x :=y appelle automatiquement la procédure de conversion de type et « y » est converti en DR.

2.1.3.4 Opérateurs

Toutes les opérations impliquent une vérification de type, mais tous les types CHP s'expriment en fonction des 2 types de base MR[B][L] (type non signé) et SMR[B][L] (type signé). Une conversion de type est effectuée seulement lorsque les types des opérandes sont structurellement différents. A titre d'exemple les opérations impliquant les types BOOLEAN, BIT et DR ne nécessitent pas de conversion puisque ils sont tous équivalent au type MR[2][1].

On note les opérateurs par ordre de priorité décroissante de haut en bas et de droite à gauche.

()
- (unaire) not abs
*** / mod + - and nand or nor xor xnor sll sla srl sra rotl rotr**
= /= < <= > >=
 # ! ?
:=

2.1.3.4.1 Opérateur de séquentialité « ; »

Sémantiquement, l'écriture « **inst1 ; inst2** » signifie que l'instruction 2 n'est exécutée que lorsque l'instruction « inst1 » a été elle même exécutée. Dès la fin de l'exécution de l'instruction « inst2 » on passe à la suite du programme.

2.1.3.4.2 Opérateur de parallélisme « , »

Sémantiquement, l'écriture « **inst1 , inst2** » signifie que l'instruction « inst1 » s'exécute parallèlement à l'instruction « inst2 ». On ne passe à la suite du programme que lorsque l'exécution des deux instructions est terminée.

2.1.3.4.3 Opérateur d'actions de communication

Les canaux et les ports sont utilisés pour communiquer entre processus et composants. Pour qu'un canal ou un port de type « td » réceptionne ou émette une donnée à travers une variable, il faut que cette variable soit également de type « td » excepté pour le signe. Aucune conversion n'est effectuée à ce niveau vu que seul l'opérateur d'affectation le permet. Les opérateurs assurant les actions de communications sont au nombre de trois : l'opérateur de sonde de canal (probe) « **#** », l'opérateur de réception de données « **?** », et l'opérateur d'émission de données « **!** ».

▪ ***Le probe « # »*** : Le probe permet de vérifier si une donnée est prête sur un port. Cet opérateur retourne un booléen et ne peut être utilisé que pour les ports passifs (un port d'entrée est passif par défaut, un port de sortie doit par contre être explicité passif avant d'être probé). En fait, il n'est utilisé que dans les expressions gardées car seules les gardes retournent un booléen. L'écriture « **nom_de_port#** » teste si une donnée est présente dans le canal relié au port « nom_de_port ». Si le port « probé » est un port d'entrée, on peut également comparer la valeur de cette donnée à une valeur explicite grâce à l'écriture « **nom_de_port# expression** ».

▪ ***Réception de données « ? »*** :
La syntaxe « **nom_de_port? ;** » permet de recevoir une donnée sans la mémoriser et la syntaxe « **nom_de_port?nom_de_variable ;** » permet de lire une donnée en la stockant dans une variable. Seuls les ports d'entrée peuvent être utilisés à gauche de l'opérateur « ? » et seules les variables sont utilisées pour recevoir les données émises par ces ports.

▪ ***Emission de données « ! »*** :
La syntaxe « **nom_de_port!** ; » permet d'émettre un acquittement et n'est permise qu'avec le type SR. La syntaxe **« nom_de_port!expression;** » permet d'émettre une donnée contenue dans une expression à travers un port. Seuls les ports de sortie peuvent être utilisés à gauche de l'opérateur « ! ». Les types des opérandes doivent être strictement identiques.

2.1.3.4.4 Autre opérateurs

Les opérateurs cités plus haut mis à part, on retrouve en CHP les opérateurs de conversion de signe, d'affectation/conversion, de comparaison, logiques, arithmétiques, et de décalage. Ils sont reportés en annexe.

2.2 Représentation intermédiaire d'un circuit asynchrone

2.2.1 Représentation mixte Réseau de Pétri (PN)- Graphe de Flot de données (DFG)

La combinaison PN-DFG est l'outil mathématique utilisé pour exprimer la forme intermédiaire de la description comportementale du circuit à réaliser. La synthèse micropipeline s'effectue à partir de cette forme intermédiaire qui explicite d'une part la partie contrôle qui permet de définir les transformations nécessaires, les données à utiliser, les

commandes et les validités de données et d'autre part la partie opérative qui permet de connaître les dépendances fonctionnelles entre les données et les opérations.

La représentation intermédiaire de circuits asynchrones peut être modélisée par la figure 2.2.

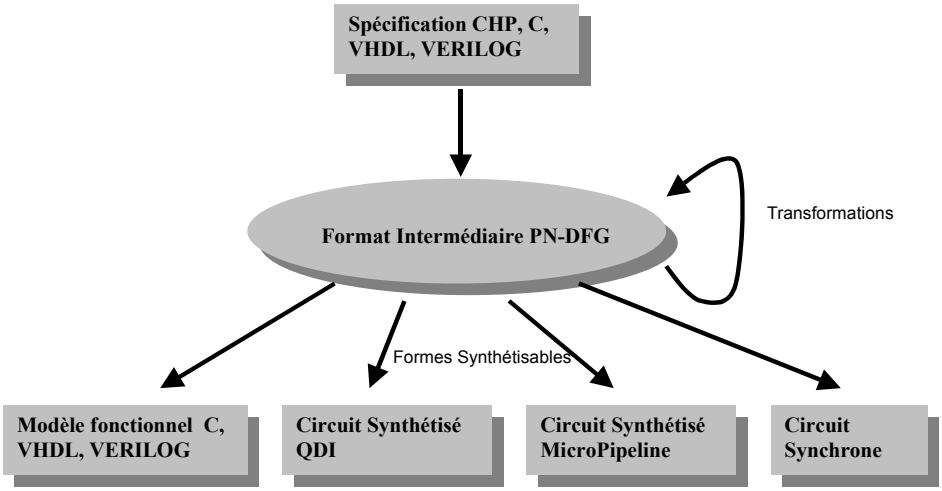


Figure 2.2 : Représentation intermédiaire de circuits asynchrones

Ce modèle de représentation permet de découpler le «*front-end*» de l'outil de son *« back-end »*. L'avantage direct de ce découplage apparaît comme étant l'extensibilité de l'outil. En effet, il semble aisé d'ajouter au niveau *front-end* un compilateur spécifique à un autre langage. D'un autre côté, cela ouvre la possibilité de cibler un autre style d'implémentation de circuits asynchrones au niveau *back-end*.

2.2.2 Réseau de Petri (ou Petri Net)

Les réseaux de Petri ou PN [PET 81] sont essentiellement employés pour traduire le comportement logique des systèmes, notamment pour décrire les systèmes contenant des concurrences, des conflits et des synchronisations. Afin d'étudier le comportement temporel des systèmes, on a introduit les PN temporisés et stochastiques [LIN 98]. Ces réseaux de Petri permettent la description de systèmes dynamiques à événements discrets de toute nature. D'autres variantes de PN ont été proposées dans la littérature parmi lesquelles on retrouve les réseaux continus et colorés pour décrire les systèmes continus, ou encore les PN hybrides qui contiennent une partie discrète et une partie continue permettant ainsi de décrire des systèmes mixtes. Une présentation formelle cohérente et complète de toutes ces variantes se trouvent dans [DAV 97].

Le réseau de Pétri de base est dit **autonome**. Formellement, il correspond à un graphe orienté constitué d'un quintuplet Q = < P , T , Pre , Post , M0 > où :

- P est un ensemble fini non vide de **places ou états possibles**,
- T est un ensemble fini non vide de transitions entre **places** ou **états possibles**,
- Pre: (PxT)→N est l'application définissant les poids des arcs **d'entrée** des transitions Pre(p,t),
- Post: (PxT)→N est l'application définissant les poids des arcs **de sortie** des transitions Post(p,t),
- M_0: P→N est la fonction qui renvoie le nombre de jetons par place à **l'initialisation**.

Si le nombre maximum de jetons pouvant transiter par un arc reliant une place à une transition ou inversement est égal à 1 cela se traduit par Pre(p,t)=1 et Post(p,t)=1. Le réseau de Petri est dit **classique** ou **ordinaire**. Dans le cas contraire, le PN est dit **généralisé**.

L'**autonomie** du PN vient du fait qu'une transition est dite **sensibilisée** (ou franchissable) sous la seule condition qu'il existe au moins un jeton ou marqueur dans toutes ses places d'entrée. Egalement, une transition est dite **tirée** (franchie, mise à feu) si elle est d'une part sensibilisée et d'autre part activée. L'opération d'activation consiste à enlever un jeton de toutes ses places d'entrée et à ajouter un jeton en plus dans toutes ses places de sortie.
A titre illustratif on donne l'exemple du PN représenté en figure 2.3.

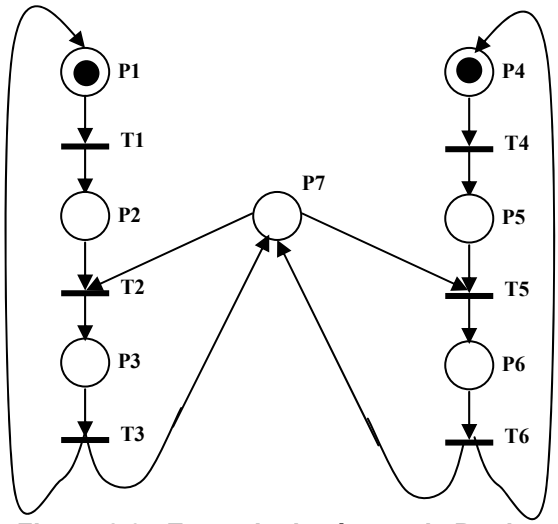


Figure 2.3 : Exemple de réseau de Petri

Sa matrice de pré-incidence Pre(p,t) est :

	T1	T2	T3	T4	T5	T6
P1	1	0	0	0	0	0
P2	0	1	0	0	0	0
P3	0	0	1	0	0	0
P4	0	0	0	1	0	0
P5	0	0	0	0	1	0
P6	0	0	0	0	0	1
P7	0	1	0	0	1	0

Sa matrice de post-incidence Post(p,t) est :

	T1	T2	T3	T4	T5	T6
P1	0	0	1	0	0	0
P2	1	0	0	0	0	0
P3	0	1	0	0	0	0
P4	0	0	0	0	0	1
P5	0	0	0	1	0	0
P6	0	0	0	0	1	0
P7	0	1	1	0	0	1

Si la mise à feu des transitions a lieu le PN présente alors un nouveau marquage M'(p) tel que M'(p) = M(p) – Pre(p,t) + Post(p,t). On appelle W(p,t) la matrice d'incidence ; elle vaut Post(p,t) – Pre(p,t).

	T1	T2	T3	T4	T5	T6
P1	-1	0	1	0	0	0
P2	1	-1	0	0	0	0
P3	0	1	-1	0	0	0
P4	0	0	0	-1	0	1
P5	0	0	0	1	-1	0
P6	0	0	0	0	1	-1
P7	0	-1	1	0	-1	1

D'où M'(p) = M(p) + W(p,t).

Pour une meilleure compréhension on donne certaines propriétés des réseaux de Petri.

- Un PN est dit **borné** si à n'importe quel état du réseau les places qui le composent possèdent un nombre borné de jetons (M(pi)=<k , k∈N). Ce genre de PN se caractérise par un espace d'états fini.
- Un PN est dit **pseudo-vivant** si indépendamment du marquage M atteint à partir du marquage initial M0, il existe une transition T telle qu'il y ait un franchissement possible. C'est à dire qu'il n'y a jamais de blocage.
- Un PN est dit **vivant** si indépendamment du marquage M atteint à partir du marquage initial M0, et indépendamment des transitions T, il existe une séquence de franchissement incluant T. C'est à dire qu'il n'y a pas de branche morte.
- Un PN est dit **ré-initialisable** (ou **propre**) si indépendamment du marquage M atteint à partir du marquage initial M0, il existe une séquence de franchissement pour passer de M à M0.

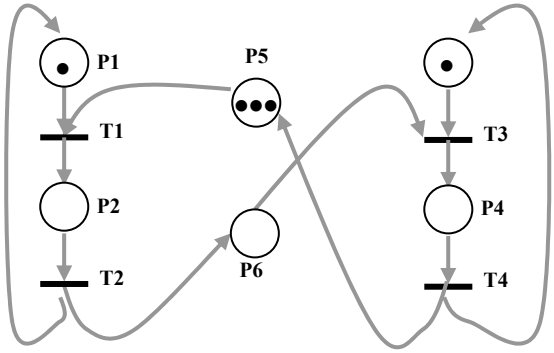
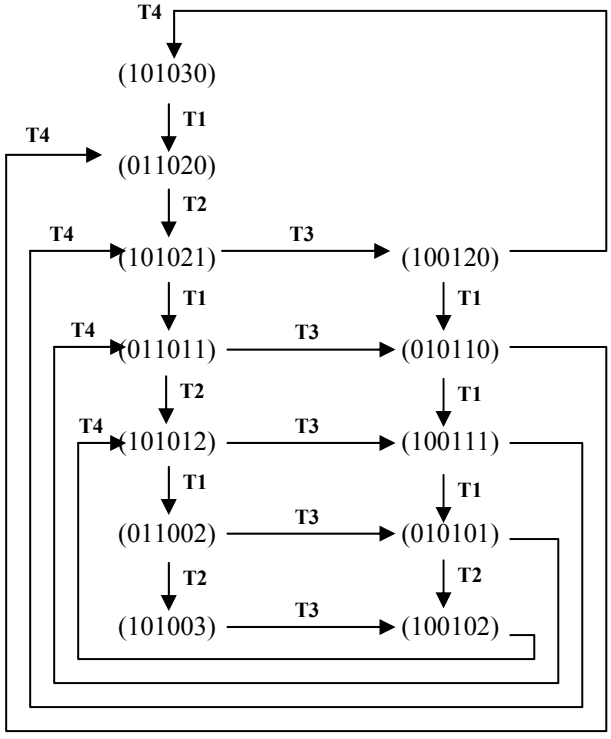


Figure 2.4 : Exemple de réseau de Petri

On propose l'exemple de PN en figure 2.4 pour illustrer ces propriétés. Dans cet exemple, le graphe des marquages évolue comme le montre la figure 2.5 et cela nous permet de conclure que :



- l'on distingue 12 états ;
- le PN est borné car k=3 ;
- le PN est pseudo-vivant car il y a toujours une transition possible ;
- le PN est vivant (pas de branche morte) ;
- le PN est ré-initialisable.

Figure 2.5 : Evolution des marquages du graphe représenté en figure 2.4

A un degré supérieur du PN autonome, on définit le réseau de Petri **synchronisé** un PN dont les transitions sont associées à des événements. La transition est alors dite étiquetée car elle est associée à une condition de franchissement que l'on peut noter conventionnellement $label_T : T \rightarrow \Sigma_C$, où Σ_C est l'ensemble des conditions de franchissement.

On peut augmenter les contraintes en associant des durées aux places. Le PN est dit alors **P-temporisé** car chaque place est associée à une instruction que l'on peut noter conventionnellement $label_P : P \rightarrow \Sigma_I$, où Σ_I est l'ensemble des instructions.

Si un réseau de Petri est P-temporisé et synchronisé il est alors dit **interprété**. Il comprend alors une partie « opérative » dont l'état est défini par un ensemble de variables d'état. L'état est modifié par des opérations associées aux places et détermine la valeur des conditions associées aux transitions. La condition de franchissement d'une transition t devient alors plus contraignante que pour un PN autonome ; elle n'est satisfaite que si :
- la transition t est sensibilisée ;
- la condition associée est vérifiée ; c'est à dire que $label_T(t)$ est vraie ;
- l'événement associé se produit.

C'est à ce type de réseau de Petri que nous nous intéressons dans cette thèse. Un exemple illustratif est donné en figure 2.6.

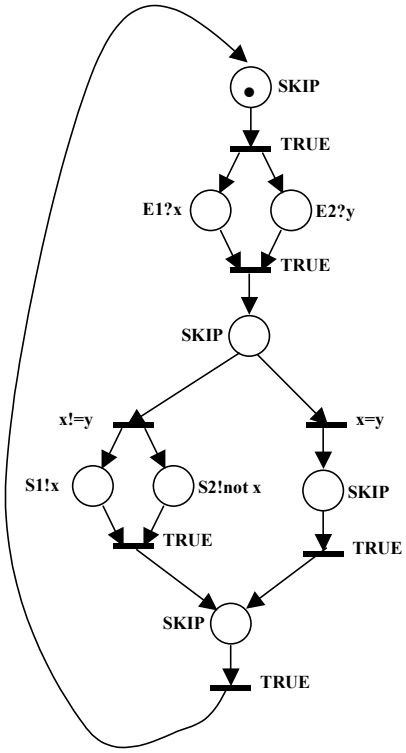


Figure 2.6 : Exemple de réseau de Petri interprété

A présent nous sommes capables de représenter les différentes structures de contrôle du CHP étendu. Ces structures sont développées dans le chapitre trois.

2.2.3 Graphe de flot de données (GFD *ou DFG pour Data Flow Graph*)

Les graphes flot de données (DFG) [ACK 82] sont des graphes finis d'arcs dirigés reliant des nœuds. Chaque nœud est une opération, et chaque arc est un transfert de données entre une opération productrice et une ou plusieurs (diffusion) opérations consommatrices. L'exécution de chaque opération et de chaque transfert de données est répétitive, d'où la notion de ``flot''. Chaque opération, à chacune de ses exécutions, consomme une donnée sur chacun de ses arcs d'entrée et combine ces données pour produire une donnée sur chacun de ses arcs de sortie. Lorsqu'une opération a besoin lors de sa *n*-ième exécution de consommer une donnée produite lors de la *(n-1)*-ième exécution d'une autre opération, il faut intercaler entre ces deux opérations une opération particulière appelée ``retard'' (le *1/z* des éléments de traitement de signal dans le cas synchrone), qui consomme une donnée sur son arc d'entrée **après** avoir produit sur son arc de sortie la donnée lue sur son arc d'entrée lors de son exécution précédente (une donnée initiale lors de sa première exécution). Sur chaque arc, chaque donnée doit être produite avant de pouvoir être consommée, donc les arcs traduisent une relation d'ordre d'exécution ``s'exécute avant'' entre les opérations, et en conséquence un graphe flot de données ne peut contenir de cycles que s'il y a au moins un retard dans chaque cycle. Ainsi un graphe flot de données n'impose qu'un ordre partiel sur l'exécution de ses opérations, et deux opérations qui ne sont pas en relation d'ordre peuvent être exécutées dans n'importe quel ordre, y compris en parallèle, si les ressources le permettent.

Parmi les DFG les plus utilisés, se trouvent les DFG à « gros grains », et les DFG à « grains fins ». Dans le premier cas, un nœud indique une fonction ou un ensemble de fonctions arithmétiques ou logiques réalisées sur des signaux d'entrées, représentés par des arcs. Dans le deuxième cas chaque nœud est une opération élémentaire (granularité la plus fine possible).

La modélisation des algorithmes par des graphes flots de données présente de nombreux avantages. Outre l'avantage de rendre le formalisme de conception proche du formalisme de spécification, et de ce fait, de fiabiliser l'interface entre le spécificateur et le réalisateur, cette approche possède un certain nombre de qualités objectives. Elle permet d'exprimer le parallélisme maximal; les seules contraintes de séquencement et de synchronisation sur les actions sont les contraintes de dépendance entre les données. Aucun choix de réalisation (parallèle ou non) ne vient polluer la conception. La modélisation par DFG permet aussi de faciliter la vérification, la construction et la transformation de programmes, et fournit directement une représentation graphique et modulaire du programme.

Les performances des graphes de flots de données ont été largement étudiées dans [KEN 81], [LEE 91]. Les DFG sont excellents pour représenter les calculs décrits par des fonctions complexes, mais ils ne sont pas adaptés pour représenter les structures de contrôle d'un système. Pour cette raison, ils sont très populaires pour la description des composants et des systèmes de traitement numérique du signal (DSP).

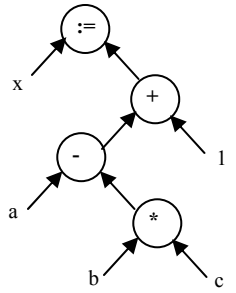


Figure 2.7 : Exemple de réseau de Graphe de Flot de Données (GFD ou DFG)

Dans le cadre de cette thèse nous nous limiterons au DFG à faible granularité où chaque nœud représente une opération élémentaire. A titre d'illustration de graphe de flot de données pour l'expression x := a - b*c + 1 est donné en figure 2.7.

2.3 Formalisation des règles DTL définissants une spécification de code haut niveau synthétisable

De façon analogue à la synthèse des circuits synchrones où des règles dites RTL (*Register Transfer Level*) spécifient les transferts de données entre registres, nous avons défini pour les circuits asynchrones des règles d'écriture de code dites DTL (*Data Transfer Level*) qui indiquent la façon dont sont transférées les données entre processus. Ces règles garantissent une synthèse systématique et indépendante du langage d'entrée utilisé. Elles peuvent légitimement être considérées comme une spécification synthétisable de circuits asynchrones parce que issu de l'étude des formes synthétisables de ce type de circuit. Elles permettent de définir un sous-ensemble du code décrivant les circuits asynchrones dans le sens où elles restreignent l'écriture de ce code dans le dessein de le rendre synthétisable. Autrement dit, un circuit asynchrone qui remplit ces règles est automatiquement synthétisé en cellules standard par la méthode de synthèse que nous proposons. Dans le cas contraire, la spécification d'un circuit asynchrone non DTL est transformée par décomposition en un ensemble de processus conformes aux règles DTL.

2.3.1 Etude des formes synthétisables

Aujourd'hui toutes les constructions possibles des langages d'entrées des outils de synthèse synchrone ne sont pas admises par ces derniers. En effet, il est procédé à l'identification de sous-ensembles de ces langages qui définissent une forme synthétisable de circuits synchrones. On cite à titre illustratif, le sous-ensemble RTL pour les langages VHDL et Verilog.

Tout comme cela a été le cas pour le style de conception RTL, le modèle de spécification DTL est issu de l'étude de la spécification et de l'implémentation des éléments de mémoire. Lors de la conception d'un circuit asynchrone les éléments de mémoire peuvent apparaître à trois différents niveaux. En premier lieu, au niveau du langage lui-même où de la mémoire est nécessaire dès lors qu'un calcul utilise une variable elle-même définie dans une étape d'exécution antérieure. Il s'agit donc d'une mémoire « fonctionnelle » puisqu'elle provient de la spécification elle-même. En second lieu, de la mémoire dite « poignée de main » est également requise pour implémenter les protocoles de type « poignée de main » parce que l'ordonnancement des actions de communication est modifié par des machines d'états finis. Enfin, de la mémoire dite de « données » existe à chaque fois qu'un circuit asynchrone implémente un rendez-vous entre des signaux de données. Notons que l'on ne considère ici, que les signaux de données et non les signaux d'acquittement, d'où le nom donné à ce type de mémoire.

Nous nous focalisons dans ce qui suit sur l'identification de la mémoire fonctionnelle car elle constitue la base de la définition du modèle de spécification DTL. La mémoire de données se traduit par des éléments de mémoires existants dans la bibliothèque de cellules tel que l'élément C dit « porte de Muller » qui traduit un rendez-vous. La mémoire de protocole sera traitée lors de l'implémentation des protocoles plus tard dans le manuscrit.

2.3.1.1 Modélisation d'un circuit asynchrone

Un circuit asynchrone peut être vu comme une boîte noire (fonction) qui produit une réponse (sortie) en réaction à un stimuli (entrée). Dans le cas où la sortie est calculée exclusivement

sur la base de la valeur des entrées le circuit est dit combinatoire. Il est dit séquentiel si, en plus de la valeur des entrées, la réponse du circuit dépend de son évolution au cours du temps passé c'est à dire des états précédents de la sortie.
Les machines d'états finis (FSM) de Mealy permettent de modéliser ce type de circuit. Formellement, une FSM de Mealy est définie par un sextuplet Q = (I, O, S, s^0, θ, Γ) où I = $< i_1, i_2, \dots, i_n >$ est l'ensemble des entrées, O = $< o_1, o_2, \dots, o_m >$ l'ensemble des sorties, S = $< s_1, s_2, \dots, s_p >$ l'ensemble des états, θ l'ensemble des fonctions de transition d'état à état tel que θ : IxS→S, et Γ l'ensemble des fonctions de génération des sorties tel que Γ : IxS→O. L'état initial de la FSM est dénotée par $s^0 = <s_1^0, s_2^0, ..., s_p^0>$. Notons que l'ensemble des variables d'états S peut être vide, alors le circuit est dit combinatoire.
Ainsi le modèle FSM du circuit ainsi défini est obtenu (figure 2.8) en faisant correspondre les entrées, sorties et états de ce dernier avec ceux de la FSM.

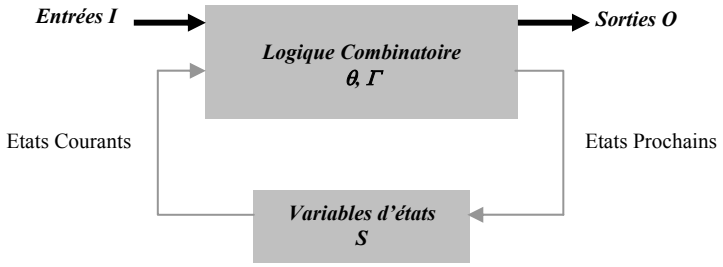


Figure 2.8 : Modèle FSM d'un circuit asynchrone/synchrone

Intéressons nous à présent aux différents cas où de la mémoire fonctionnelle est générée.

2.3.1.2 Mémoire fonctionnelle causée par une variable locale

On considère ici l'utilisation des variables locales à l'intérieur d'un processus. Une variable déclarée, lue, et affectée dans un processus peut ou non représenter une « mémoire fonctionnelle ». En effet, les variables dites « intermédiaires » ne génèrent pas de mémoire fonctionnelle car au cours de l'exécution du processus elles sont lues sur les canaux d'entrée, puis font l'objet de calculs intermédiaires (commandes gardées, opérations arithmétiques ou logiques, fonctions complexes), puis sont émises sur les canaux de sortie. Dans ce cas l'ensemble des variables d'états S est vide. Au contraire, les valeurs des variables dites « mémorisantes » (ou de mémoire) dépendent de l'état précédent du processus et génèrent par conséquent de la mémoire fonctionnelle. Typiquement, elles peuvent correspondre à des variables d'états, des registres, ou des mémoires. Dans ce cas, l'ensemble des variables d'états S est non vide.

En d'autres termes les processus dont les sorties dépendent exclusivement des entrées n'utilisent pas de variables de mémoire. Ces processus sont alors qualifiés de « combinatoires », ce qui signifie que leur implémentation se fait sans mémoire fonctionnelle. Appliqué sur les modèles de processus communiquant, un processus combinatoire émet des valeurs sur ses canaux de sorties qui sont exclusivement calculés à partir d'informations provenant des canaux d'entrée.

Inversement les processus qui requièrent des variables de mémoire sont dits « séquentiels ». Dans ce cas, le calcul des sorties implique aussi bien les entrées que les variables d'états affectées durant un calcul précédent.

2.3.1.3 Mémoire fonctionnelle causée par l'opérateur séquentiel « ; »

De toute évidence, l'utilisation de l'opérateur de séquentialité noté « ; » dans un programme crée des états et par conséquent des mémoires fonctionnelles. A titre illustratif, dans le processus [E?x ; E?x ; S!x ; LOOP] décrit en langage CHP, le canal E est lu deux fois successivement et la dernière valeur lue est propagée en sortie par l'écriture sur le canal de sortie S. Ce processus requiert un état (donc une mémoire) induit de fait par l'opérateur séquentiel qui caractérisent le premier puis le second accès en lecture sur le canal E.
Toutefois, tous les opérateurs séquentiels ne génèrent pas de la mémoire fonctionnelle. En effet, l'opérateur séquentiel qui sépare la deuxième lecture du canal d'entrée E et l'écriture dans le canal S n'implique pas d'état parce qu'il existe une vraie dépendance entre ces deux actions.
Du point de vue de l'implémentation, les dépendances vraies ne requièrent aucun matériel supplémentaire. Dans cet exemple simple, la valeur provenant de la deuxième lecture du canal d'entrée E est propagée au canal de sortie S. L'opérateur séquentiel dans ce cas ne « coûte » rien. Inversement, la séquentialité a un coût lorsqu'il n'existe pas de vraies dépendances. Ce coût correspond à l'implémentation de l'état requis pour assurer la séquentialité.

2.3.1.4 Mémoire fonctionnelle causée par l'opérateur parallèle « , »

L'utilisation de l'opérateur parallèle peut générer de la mémoire fonctionnelle. Cela est notamment le cas lorsqu'une variable est partagée en écriture, c'est à dire si elle est affectée, tout en étant lue ou modifiée dans une branche d'exécution concurrente. Ceci concerne également un canal de communication partagé en écriture (conflit) ou en lecture (violation de la propriété de communication point à point). Hormis dans les trois cas sus-cités, où le code est non synthétisable, les opérateurs parallèles n'induisent aucune mémoire, notamment lorsqu'une variable de lecture est partagée uniquement en lecture.

2.3.1.5 Mémoire fonctionnelle causée par l'initialisation

Lorsque l'on modélise des circuits avec des processus communiquant concurrents, nous pouvons rencontrer deux types d'initialisation. D'abord, l'initialisation de variable locale qui requiert d'initialiser proprement l'état du processus, n'est pas synthétisable car il s'agit d'un élément de mémoire (cf § 2.3.1.2). Ensuite, l'initialisation de canal consiste à émettre une valeur sur un canal de sortie une fois seulement, avant que le processus ne commence une infinité de boucle d'exécution. Bien que cela implique une écriture séquentielle sur un canal de sortie, ce type d'initialisation est synthétisable par une implémentation particulière que nous ultérieurement dans ce manuscrit dans la partie réservée à la synthèse des contrôleurs.

2.3.2 Règles DTL

Sur la base de cette analyse des éléments de mémoire impliqués dans l'implémentation des circuits asynchrones, nous avons défini un ensemble de règles d'écriture qui permettent de réaliser le circuit conformément au modèle FSM présenté précédemment. A l'intérieur d'un processus, la spécification DTL restreint l'utilisation de variables d'état donc de mémoire fonctionnelle. Un code décrivant un circuit asynchrone est dit conforme DTL que s'il est seulement constitué de processus communicants combinatoires et de processus où les canaux de communication sont initialisés. Ainsi, un processus conforme DTL doit satisfaire les règles 1 à 6 décrites ci-dessous.

- **1** L'utilisation de variables partagées est prohibée hormis dans le cas où elles sont seulement consultées.

- **2** Dans toutes les exécutions possibles du processus, une variable est d'abord affectée avant d'être lue.
- **3** Les valeurs émises vers les canaux de sorties ne doivent dépendre que des valeurs reçues des canaux d'entrées.
- **4** Les canaux ne peuvent être accédés séquentiellement que dans le cas où les canaux de sorties sont initialisés une seule fois en début de processus.
- **5** Les instructions séquentielles doivent expliciter une vraie dépendance.
- **6** Les mêmes canaux ne peuvent être accédés concurremment.

Ces règles traduisent donc trois types de restriction. La première exprimée par les règles 1 à 3 concerne l'utilisation des variables de mémorisation, et permet de conclure que l'on ne peut utiliser que les variables intermédiaires. Cela permet d'avoir un processus de type consommation-production où toute valeur émise sur un canal de sortie dépend uniquement des valeurs lues sur les canaux d'entrée. Ces processus sont combinatoires. La seconde restriction traduite par les règles 4 et 5 porte sur l'opérateur séquentiel et empêche toute manifestation de conflit entre deux protocoles successifs sur un même canal. Enfin la règle 6 prémunit contre l'apparition de conflit de données sur un même canal : un canal d'entrée ne peut être consommé deux fois concurremment, de même que deux valeurs différentes ne peuvent être émises par un canal de sortie.

A ce niveau de réflexion, nous pouvons affirmer qu'un code spécifiant un circuit asynchrone et respectant l'ensemble des règles définies précédemment est systématiquement synthétisé en cellules standards par la méthode de synthèse que nous présentons ultérieurement. Mais il semble légitime de se demander si ces règles permettent aux concepteurs de décrire un quelconque circuit asynchrone. De même, on peut s'interroger sur ce qu'il convient de faire lorsqu'un code descriptif d'un circuit asynchrone viole ces règles. C'est ce que nous allons voir dans ce qui suit.

2.3.3 Transformation d'un code non DTL en code conforme DTL

Un programme descriptif d'un circuit asynchrone qui ne respecte pas le style d'écriture DTL peut être transformé en une forme compatible DTL. Plus clairement, un processus quelconque non combinatoire – c-à-d incluant des états – peut, par décomposition, être traduit en un ensemble de processus combinatoires. Nous considérons deux types de processus non combinatoires, le premier constitué d'une ou de plusieurs variables de mémorisation, et le second possédant un ou plusieurs opérateurs séquentiels. Nous illustrons ici nos propos sur la base de codes CHP, mais l'idée proposée reste générale et peut s'appliquer à tout autre langage de processus communiquant tel que Balsa ou Tangram.

2.3.3.1 Extraction des variables de mémorisation

Nous souhaitons modéliser un processus P (figure 2.9) exprimé en code CHP et comportant une variable de mémoire « x ».

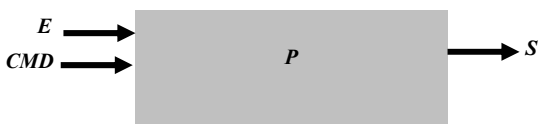


Figure 2.9 Processus non synthétisable (variable de mémoire « x »)

```
P ≡  [ CMD ?cmd;
       @[
           cmd=0 => E?x ; BREAK
           cmd=1 => S!x ; BREAK
         ]; LOOP
     ]
```

Ce processus n'est pas DTL puisque la variable « x » est un élément mémoire (cf § 2.3.1.2). On extrait cet élément de mémoire, en créant un nouveau processus qui lui est associé, pour obtenir une spécification conforme au modèle FSM représenté figure 2.8 (calcul des sorties de P en fonction des entrées et de l'état courant et calcul de l'état suivant). Deux solutions sont proposées pour l'extraction de cet élément de mémoire. La première se base sur l'utilisation de registres conventionnels, et la seconde se base sur l'utilisation de registres circulaires.

▪ ***Extraction de la variable de mémorisation par la méthode des registres conventionnels***
Le nouveau processus P'' créé est exprimé en code CHP.

```
P'' ≡  [ RW ?rw;
         @[
             rw=0 => W?x ; BREAK
             rw=1 => R!x ; BREAK
           ]; LOOP
         ]
```

D'emblée il apparaît que ce processus n'est pas non plus conforme DTL. Pour cela, on *suppose* que ce processus est *synthétisé* et *inclus une fois pour* toute dans la bibliothèque de cellules à partir de quoi la synthèse d'un tel circuit devient *directe par instanciation*.

Le processus initial contenant la variable mémoire x est alors transformé en un processus synthétisable P' exprimé en code CHP.

```
P' ≡  [ CMD ?cmd;
        @[
            cmd=0 => E?x ; RW!0 , W!x ; BREAK
            cmd=1 => RW!1 , R?x ; S!x ;  BREAK
          ]; LOOP
        ]
```

Le processus d'origine P est donc transformé en appliquant deux règles. Tout accès en écriture à la variable x est associé aux deux actions parallèles **[**RW**!**0**,** W**!**x**]** et tout accès en lecture à la variable x est associé aux deux actions parallèles **[**RW**!**1, R**?**x**]**

La figure 2.10 représente la vue équivalente du processus P après extraction de la variable « x ».

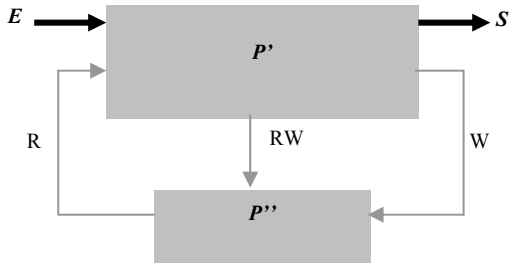


Figure 2.10. Extraction de variable mémorisante par registre conventionnel

La variable est par conséquent stockée dans le processus P'' et la mémoire fonctionnelle associée à « x » ne se trouve désormais plus à l'intérieur du processus initial P0. Tous les éléments de mémoire d'un processus peuvent être extraits de cette manière pour faire en sorte que ce processus ne contienne pas de mémoire fonctionnelle.

▪ ***Extraction de la variable de mémorisation par la méthode des registres circulaires***
Le nouveau processus P'' crée est associé à la variable « x » s'exprime en CHP.

```
P'' ≡  [ CV!0;    -- initialisation
          [ NV?x ; CV!x ; LOOP ]        -- propagation de la valeur
        ]                               -- CV : Current Value
                                        -- NV : Next Value
```

Ce processus est un *buffer* complet avec initialisation de canal. Il est appelé « registre circulaire » parce que la variable « x » circule à travers ce buffer à chaque fois qu'elle est accédée.

Le processus initial contenant la variable mémoire x est alors transformé en un processus synthétisable P' exprimé en code CHP.

```
P' ≡  [ CMD ?cmd;
        @[
            cmd=0 => E?x ; NV!x ; BREAK
            cmd=1 => CV?x ; S!x ;  BREAK
          ]; LOOP
      ]
```

Le processus d'origine P contenant la variable « x » est donc transformé en appliquant deux règles suivantes. Tout accès en écriture à la variable « x » est remplacé par les deux actions parallèles **[**CV**? ,** NV**!**x**]** et tout accès en lecture à la variable « x » est remplacé par les deux actions séquentielles **[**CV**?**x **;** NV**!**x**].**

La figure 2.11 représente la vue équivalente du processus P après extraction de la variable « x ».

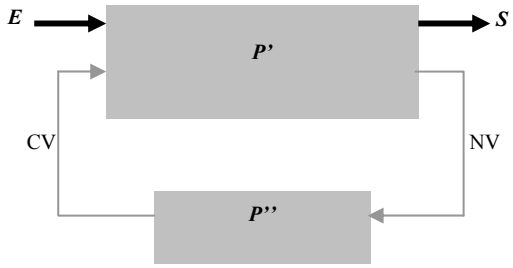


Figure 2.11. Extraction de variable mémorisante par registre circulaire

Notons que pour éviter le problème de blocage généré dans un cycle fermé (tel que celui créé par les 2 processus P' et P''), ce dernier doit au moins contenir trois demi-buffers. L'initialisation du processus P'' est en fait un demi buffer ajouté dans la boucle (il y a déjà deux demi-buffer, l'un pour le processus P' l'autre pour le processus P'' lui-même). L'initialisation de P'' inhibe tout blocage.

Ainsi, en lieu et place de la transformation précédente où la variable était stockée dans un registre accessible aléatoirement en mode lecture/écriture, la variable est ici stockée dans un registre circulaire qui impose un accès séquentiel ; l'émission vers le canal « CV » se fait après la réception à partir de « NV ». Par conséquent une mise à jour de la variable « x » correspond à une lecture factice et à un accès en écriture au registre circulaire. La donnée stockée dans le registre circulaire doit d'abord être consommée pour libérer le registre. Réciproquement, lorsque « x » est lue, la variable est consommée et il s'avère nécessaire de réécrire sa valeur dans le registre circulaire afin qu'elle soit disponible pour le prochain accès. Dans ce cas, une optimisation qui limite l'activité et donc la consommation, consiste à consulter le contenu de la variable, et à ne pas la consommer, dans le but d'éviter un rafraîchissement. La variable est ainsi mise à jour seulement lors d'un accès en écriture.

Toutes les variables de mémoire d'un processus peuvent ainsi être extraites et modélisées avec un buffer circulaire pour faire en sorte que le processus d'origine soit exempt de mémoire fonctionnelle. Cette décomposition est particulièrement efficace lorsque le processus d'origine spécifie une lecture séquentielle puis des accès en écriture aux variables parce que le registre circulaire se comporte comme un registre en pipeline. Ceci est le cas par exemple, lors de la modélisation de machines à états finis où l'état est d'abord lu puis mis à jour. Dans de tels cas, l'extraction de registres simples serait inefficace parce que cela requièrerait de procéder séquentiellement d'abord à une lecture ensuite à une écriture du registre.

Les deux transformations proposées pour extraire les variables de mémoire d'un processus sont très générales et offrent des compromis intéressants. En effet, comme cela a été suggéré plus haut, ces transformations ne seront pas appliquées selon les mêmes conditions. Une décomposition utilisant un registre est particulièrement intéressante lorsque la variable de mémoire est accédée aléatoirement. Inversement, les registres circulaires sont plus adéquats lorsque la variable de mémoire est d'abord lue puis ensuite mise à jour. C'est le cas dans les machines d'états finis pour le calcul des sorties et des états.

2.3.3.2 Extraction des opérateurs séquentiels

Comme mentionné précédemment, les accès séquentiels à un canal sont prohibés en spécification DTL puisqu'ils exigent de la mémoire. Une solution pour extraire ces opérateurs séquentiels en vu de rendre le code compatible à la FSM présentée en figure 2.8 consiste à rendre explicite (par encodage) les états associés aux opérateurs séquentiels, puis à les modéliser par des registres circulaires. Le principe de ces deux transformations peut être illustré au travers du séquencement de deux blocs d'instructions conformes DTL codé en CHP.

$$P \equiv [\, A \,;\, B \,;\, LOOP \,]$$

Un état est d'abord affecté à chaque action qui suit un opérateur séquentiel. Un état initial est aussi réservé pour exécuter l'ensemble des actions qui précèdent le premier opérateur séquentiel. Deux états sont donc nécessaires, l'un pour le bloc B et le second pour l'état initial. La variable est extraite par la méthode des registres circulaires.

```
P' ≡  [ CS ?cs;
        @[
            cs=0 => A , NV!1 ; BREAK
            cs=1 => B , NS!0 ;  BREAK
          ]; LOOP
      ]
//
P'' ≡  [ CV!0;
          [  NV?nv ;  CV!nv  ;  LOOP  ]

        ]
```

Le processus P est alors équivalent à deux processus parallèles P' et P''. Le processus P' prend en charge l'exécution des blocs d'instructions du processus initial (décodage de l'état courant et calcul de l'état suivant). Le processus P'' sert à propager l'état. Le nombre d'états nécessaires à la décomposition est le nombre d'actions d'opérateur séquentiel dans le processus plus un.

Une technique est proposée pour réduire le nombre d'états nécessaires et permet d'optimiser la consommation et d'augmenter la performance du circuit. Elle est fondée sur la manière d'affecter et d'encoder les états du processus initial. A titre illustratif prenons l'exemple d'un décodeur d'instructions simples toutes exemptes de mémoire fonctionnelle. Selon la valeur du canal G différentes séquences d'ordres sont générées.

```
P ≡ [  G?g ;
       [
         g = 0   =>   A ; B ; BREAK
         g = 1   =>   C ; D ; BREAK
         g = 2   =>   E ; BREAK
       ] ; LOOP
    ]
```

- ***La première transformation***

On code les états associés à B et à D. La variable d'état est extraite et modélisée comme un registre circulaire, et le programme du décodeur est transformé en deux processus concurrents P' et P''.

```
[  CS?cs ;
       @[
          cs = 0   =>  G?g ;
                 @[
                    g = 0   =>   A , NS!1; BREAK
                    g = 1   =>   C , NS!2; BREAK
                    g = 2   =>   E; BREAK
                  ]
           cs = 1   =>   B , NS!0; BREAK
           cs = 2   =>   D , NS!3; BREAK
       ] ; LOOP

]
```

//

```
[  CS ! 0 ;
       [
          NS?x   ;   CS!x ; BREAK
        ]; LOOP
]
```

Notons que le nombre d'états nécessaire (cinq) peut être réduit si les séquences des listes d'actions peuvent être factorisées à partir des différentes commandes gardées.

▪ ***La seconde transformation***

Elle consiste à associer un état distinct à chaque opérateur séquentiel qui apparaît dans une commande gardée. Ici, l'état est également modélisé par un registre circulaire et le décodeur se réécrit comme suit :

```
@[   #G=0 => CS?cs ;
      @ [
          cs = 0  =>    A , NS!1; BREAK
          cs = 1  =>    B , NS!0 , G? ; BREAK
         ]
     #G=1 => CS?cs ;
      @[
         cs = 0    =>    C , NS!1; BREAK
        cs = 1    =>    D , NS!0, G? ; BREAK
        ]
      #G=2 => E , G? ; BREAK
] ; LOOP
```

//

```
[ CS ! 0 ;
        [
           NS?x   ;    CS!x ; BREAK
         ]; LOOP
]
```

Cette transformation requiert l'utilisation du « probe » (#). Le nombre d'états introduits par cette transformation ne dépend pas du nombre total de listes d'actions séquentielles dans les commandes gardées, mais du nombre maximum de ces listes dans les commandes gardées. Dans le cas du décodeur, le nombre d'états introduit est de quatre parce que la commande gardée « 2 » a la plus longue séquence de liste d'actions (à savoir quatre).

Il est intéressant de noter que cette transformation produit des modèles très modulaires vu que les ensembles d'états associés aux commandes gardées sont indépendants les uns des autres. Cela se voit bien si l'on crée un ensemble d'états « CS0 » et « CS1 » par commande gardée. La ré-écriture du programme donne alors :

```
@[ #G=0 => CS0?cs0 ;
         @[
          cs0 = 0=>    A , NS0!1; BREAK
          cs0 = 1 =>   B , NS0!0 , G? ; BREAK
           ]
 #G=1 => CS1?cs1 ;
     @[
        cs1 = 0 =>   C , NS1!1; BREAK
       cs1 = 1 =>D , NS1!2, G? ; BREAK
       ]
 #G=2 => E , G?
] ;LOOP
```

//

```
[ CS0 ! 0 ;
        [
           NS0?x ; CS0!x ; BREAK
         ]; LOOP
]
```

//

```
[ CS1 ! 0 ;
        [
           NS1?x ; CS1!x ; BREAK
         ]; LOOP
]
```

Lors de la conception de circuits asynchrones, cette propriété est essentielle pour faire en sorte que les cas favorables soient les plus rapides et les plus faibles consommateurs en puissance. Cette propriété peut évidemment être exploitée dans un but d'optimisation en termes de vitesse et de consommation en puissance.

La première transformation proposée, à l'inverse, ne permet pas une telle séparation entre des sous-ensembles d'états et peut compliquer les commandes gardées simples. C'est le cas de la troisième commande gardée dans l'exemple du décodeur, qui reste inchangée après la transformation « 2 » alors qu'elle est plus compliquée après application de la transformation « 1 ».

On pourrait également remarquer que le programme obtenu avec la seconde transformation lit le canal « G » à la fin de chaque séquence d'actions, alors que le programme d'origine exécute cette lecture au tout début. Ceci peut être facilement corrigé par l'ajout d'un buffer dans le canal « G ».

En conclusion, les transformations proposées permettent la ré-écriture de programmes séquentiels pour les rendre conformes à la spécification DTL. En effet, on obtient bien des programmes exempts de mémoire fonctionnelle, en commençant par extraire les variables de mémoire puis en enlevant les opérateurs séquentiels.

2.4 Conclusion

Ce chapitre nous a donc permis de décliner la spécification d'un circuit asynchrone à trois niveaux. Au niveau conception, nous avons introduit le langage de description de haut niveau CHP défini par Alain Martin et auquel des extensions ont été rajoutées pour faciliter la description de circuits asynchrones complexes, la vérification par simulation de ces circuits ainsi que leur implémentation physique via la synthèse. La représentation intermédiaire grâce à la combinaison de PN-DFG obtenue par la compilation du langage CHP permet de découpler le back-end du front-end et d'assouplir les extensions qui pourraient être apportées à l'outil tant en amont qu'en aval. Enfin, en dernier lieu, la spécification DTL vient poser les règles permettant d'obtenir une spécification synthétisable des circuits asynchrones. Comme cela a été mentionné, ces règles restreignent essentiellement les éléments de mémoire et permettent d'écrire un code décrivant un circuit asynchrone dépourvu de mémoire fonctionnelle, ce qui garantit une synthèse systématique en portes logiques.
Dans le chapitre trois, nous expliquons la technique du pipeline asynchrone et ses caractéristiques essentielles, puis nous explicitons le modèle de circuit traduisant l'architecture que nous ciblons pour la synthèse micropipeline.

CHAPITRE III

3. Modèle de circuit cible micropipeline

La conception de circuits asynchrones constitue une alternative viable face aux contraintes qui affectent les circuits synchrones complexes fonctionnant à fréquence élevée, tels que les problèmes de propagation et de distribution d'horloge [DAL 98]. Les circuits pipelinés à grain fin qui constituent une catégorie particulière de circuits asynchrones développés, ont démontré des performances élevées en vitesse [SUT 01], [NYS 01], [SIN 00], [SIN 00b], [SIN 01], [LINE 98], [FER 02], [OZD 02], [OZD 02b] et [TUG 02]. De plus ils permettent d'éviter la génération, l'optimisation, et la vérification de contrôleurs complexes [YUN 98], car ils implémentent un contrôle distribué constitué de contrôleurs élémentaires de faible complexité.

Dans ce chapitre, nous expliquons d'abord le fonctionnement de base du pipeline asynchrone, puis nous soulignons les travaux existants sur ce type de circuits, enfin nous proposons pour conclure le modèle de circuit qui sera la cible de la méthode de synthèse micropipeline proposée.

3.1 Pipeline Asynchrone

Typiquement, un système asynchrone peut être assimilé à un ensemble de blocs fonctionnels communicants localement et mutuellement au travers de canaux en utilisant un protocole de communication de type « poignée de main ». « Pipeliner » un ensemble de blocs fonctionnels revient à les décomposer et à les organiser de manière à ce qu'ils constituent plusieurs étages concurrents. De cette façon, on arrive à augmenter le débit de sortie du système tout en assurant un faible coût en surface. Un exemple de pipeline est une file de type FIFO (First In First Out) dans laquelle les premiers jetons envoyés à l'entrée, sont également les premiers en sortie.

Par comparaison à un pipeline synchrone où les jetons avancent d'un étage à l'autre sur le rythme d'un cycle d'horloge, le pipeline asynchrone ne considère pas d'horloge globale pour synchroniser la circulation des données : les jetons se déplacent au fur et à mesure qu'une place se libère devant eux. On peut distinguer les circuits asynchrones, comme mentionné au chapitre 1, selon un certain nombre de paramètres tels que la taille des composants fonctionnels, le parallélisme des protocoles de communication type « poignée de main », le codage des données, et les hypothèses temporelles qui assurent le fonctionnement du circuit.

Nous nous intéressons dans ce chapitre aux circuits asynchrones micropipeline utilisant le protocole quatre phases avec le codage « données groupées » (ou BD pour Bundled Data).

3.1.1 Fonctionnement de base d'un pipeline

Dans un pipeline synchrone (figure 3.1), c'est l'arrivée d'un front d'horloge sur tous les registres du pipeline qui provoque le déplacement des données. Cela entraîne qu'une fois dans le pipeline, deux données d'entrée restent toujours séparées du même nombre d'étages.

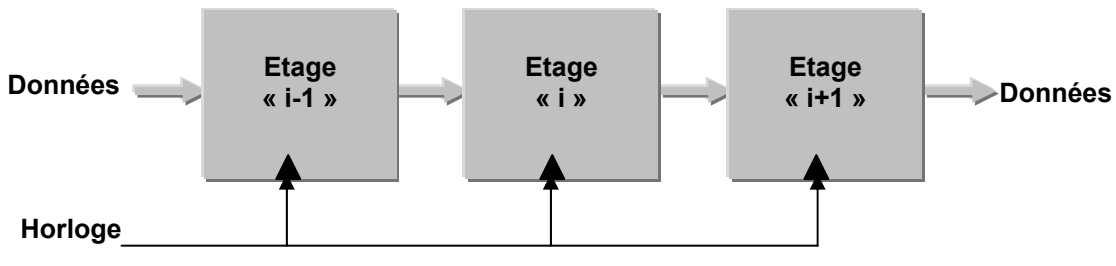


Figure. 3.1 : Pipeline Synchrone

Dans le pipeline asynchrone (figure 3 .2), cette synchronisation relativement forte des données entre elles est avantageusement assouplie. En effet, c'est un signal de requête qui est utilisé pour indiquer la disponibilité de données valides à un étage du pipeline, et signale à ce dernier qu'il peut donc entamer son calcul. L'avantage immédiat est que si aucune requête n'est effectuée, aucun calcul n'est lancé, ce qui assure une faible consommation à grain fin.

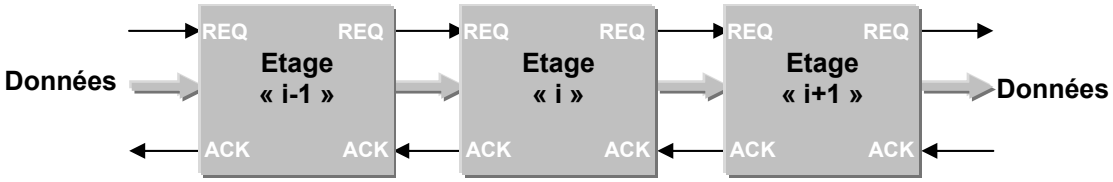


Figure. 3.2 : Pipeline Asynchrone

Par ailleurs, tant que le traitement des données courantes n'est pas terminé, on n'émet aucun signal de requête en aval, et l'on impose le blocage des données en amont. Ceci est assuré par l'attente du signal d'acquittement, lequel permet alors d'informer l'étage en amont de la bonne réception des données et du fait qu'il peut générer de nouvelles données. Une bulle est ainsi créée entre les étages et il n'y pas de risque de perdre le flot de donnée entre les étages.

Donc, aucune nouvelle donnée ne peut être produite par l'étage en amont tant qu'il n'a pas reçu d'acquittement, et l'étage en aval reste bloqué tant que l'acquittement qu'il a envoyé n'a pas été reçu par l'étage courant. Ainsi, les données progressent dans un pipeline asynchrone aussi longtemps qu'elles ne rencontrent pas de ressource occupée, et ceci indépendamment des données qui les suivent.

Ce verrouillage en amont et en aval fournit la propriété intéressante de blocage local du pipeline asynchrone.

Outre les propriétés de faible consommation à grain fin, et de blocage localisé, les pipelines asynchrones se distinguent par la faible variation du courant lors des commutations et par la possibilité d'utiliser des verrous transparents sans réduire la vitesse du flot de données [SUT 89]. Ces propriétés découlent de la disposition des étages du pipeline asynchrone qui sont activés en fonction des étages voisins en amont et en aval.

3.1.2 Performance d'un pipeline

La performance d'un pipeline se mesure par un certain nombre de paramètres que sont la capacité mémoire, le débit, la latence, et le temps de cycle. La capacité mémoire d'un étage de pipeline est le nombre maximum de jetons qu'il peut contenir sans bloquer l'entrée du pipeline. Le débit d'un pipeline est le nombre de jetons par seconde qui passe par un étage donné. La latence est le temps nécessaire pour qu'une donnée traverse les étages d'un pipeline. Le temps de cycle est le temps maximum qui sépare la prise en compte de deux données successives dans cet étage. Il est égal à la latence totale de la boucle activée dans l'étage (plus généralement, il faut considérer le maximum parmi les latences des différentes boucles mises en jeu). Le temps de cycle correspond aussi au débit maximal (local) de l'étage. Dans le cas d'un protocole 4 phases il faut inclure le temps de remise à zéro de la boucle, ce qui correspond à un double parcours de la boucle par jeton.

3.1.3 Pipeline linéaire et non-linéaire

La plupart des nouveaux pipelines asynchrones introduits ([SIN 00], [SIN 00b], [SIN 01], [LINE98], [SUT 89], [FER 02], [OZD 02], [OZD 02b] et [TUG 02]), ciblent des applications

de pipelines linéaires, telles que les FIFO. Or la complexité des structures de pipeline croît avec celle du circuit asynchrone. D'où l'importance de pouvoir traiter également les applications de pipelines non linéaires que nous présentons dans ce qui suit en soulignant les problèmes qui leurs sont associés.

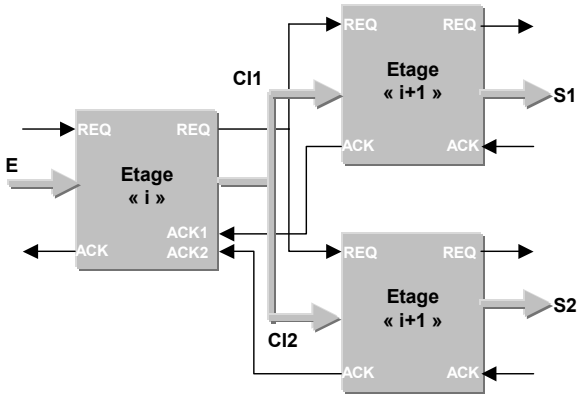
3.1.3.1 Pipeline linéaire

Un étage de pipeline linéaire est un étage qui a un seul canal d'entrée et un seul canal de sortie.

3.1.3.2 Pipeline non linéaire

Un étage de pipeline non linéaire est un étage qui peut avoir plusieurs canaux d'entrées et plusieurs canaux de sorties. La fourche, la convergence, le multiplexage, et le démultiplexage sont autant d'exemples de pipelines non linéaires.

Une fourche est un étage de pipeline avec un canal d'entrée et plusieurs canaux de sortie. Les données sont diffusées dans tous les canaux de sortie (figure 3.3). Dans le cas du protocole quatre phases, une fourche ayant plusieurs canaux de sorties doit recevoir un signal d'acquittement de tous ces derniers pour revenir à zéro.




```
COMPONENT Fourche_F
PORT (e : IN DR ;
      s1 : OUT DR;
      s2: OUT DR)
CHANNEL CI1, CI2 : DR;
BEGIN

 PROCESS Fourche_F_Proc1
 PORT (e : IN DR ;
       CI1 : OUT DR;
       CI2: OUT DR)
 VARIABLE x : DR;
 BEGIN
  [ e?x ; CI1!F1(x ) ; CI2!F2(x ) ;
    LOOP];
 END;
```

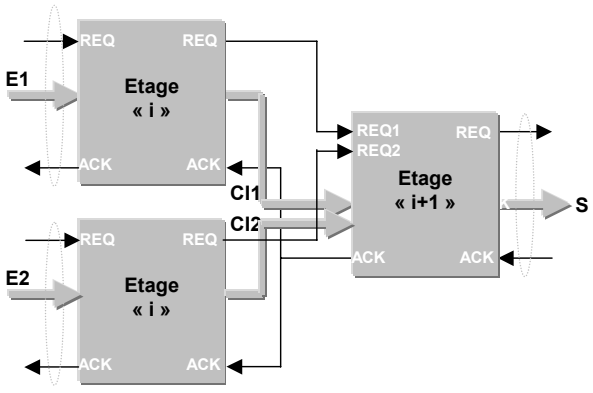
```
 PROCESS Fourche_F_proc2
 PORT (CI1 : IN DR ;
       s1: OUT DR)
 VARIABLE x : DR;
 BEGIN
  [ CI1?x ; s1!F1(x ) ; LOOP];
 END;

PROCESS Fourche_F_proc3
 PORT (CI2 : IN DR ;
       s2: OUT DR)
 VARIABLE x : DR;
 BEGIN
  [ CI2?x ; s2!F2(x ) ; LOOP];
 END;

END;
```

Figure. 3.3 : Pipeline Asynchrone Non Linéaire :Fourche (*Fork*)

Une convergence est un étage de pipeline avec plusieurs canaux d'entrée et un canal de sortie. Les données sont fusionnées dans le canal de sortie (figure 3.4). Dans le cas du protocole quatre phases, une convergence qui reçoit des données de ses canaux d'entrée doit émettre un signal d'acquittement sur tous les canaux d'entrée.




```
COMPONENT Convergence_F
PORT (e1 : IN DR ;
      e2 : IN DR;
      s: OUT DR)
CHANNEL CI1, CI2 : DR ;
BEGIN
 PROCESS Covergence_F_proc1
 PORT (CI1 : IN DR ;
       CI2 : IN DR;
       s: OUT DR)
 VARIABLE x,y : DR;
 BEGIN
  [ CI11?x ; CI2!y ; s!F(x,y ) ; LOOP];
 END;
```

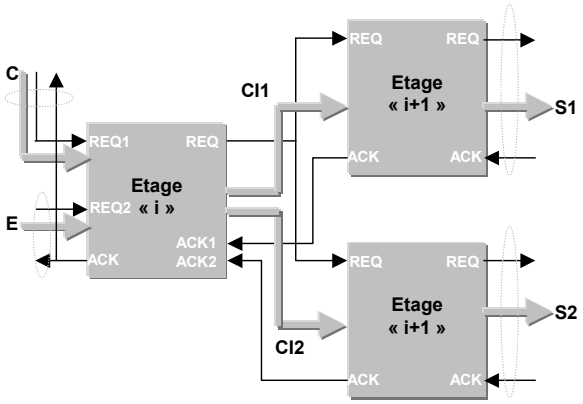
```
 PROCESS Convergence_F_proc2
 PORT (e1 : IN DR ;
       CI1: OUT DR)
 VARIABLE x : DR;
 BEGIN
  [ e1?x ; CI1!x ; LOOP];
 END;

 PROCESS Convergence_F_proc3
 PORT (e2 : IN DR ;
       CI2: OUT DR)
 VARIABLE x : DR;
 BEGIN
  [ e2?x ; CI2!x ; LOOP];
 END;

END;
```

Figure. 3.4 : Pipeline Asynchrone Non Linéaire : Convergence (*Join*)

Un démultiplexage (figure 3.5) est une combinaison de fourche et de convergence. En effet, en plus du traditionnel canal d'entrée, un second canal d'entrée dit de « choix » est utilisé pour indiquer l'envoi de la donnée d'entrée soit à l'un des deux canaux de sortie, soit à une quelconque combinaison des canaux de sorties, soit à aucun d'entre eux (cette dernière option est notée « skip »).




```
COMPONENT Demux_F
PORT (e : IN DR ;
      c : IN DR ;
      s1 : OUT DR;
      s2: OUT DR)
CHANNEL CI1, CI2 : DR;
BEGIN

 PROCESS Demux_F_Proc1
 PORT (e : IN DR ;
       c : IN DR ;
       CI1 : OUT DR;
       CI2: OUT DR)
 VARIABLE x,y : DR;
 BEGIN
 BEGIN
  [c?x ; e?y ;
    @[ x=0 => CI1!F1(y) ; BREAK
       x=1 => CI2!F2(y) ; BREAK
     ]
  ] LOOP;
 END;
```

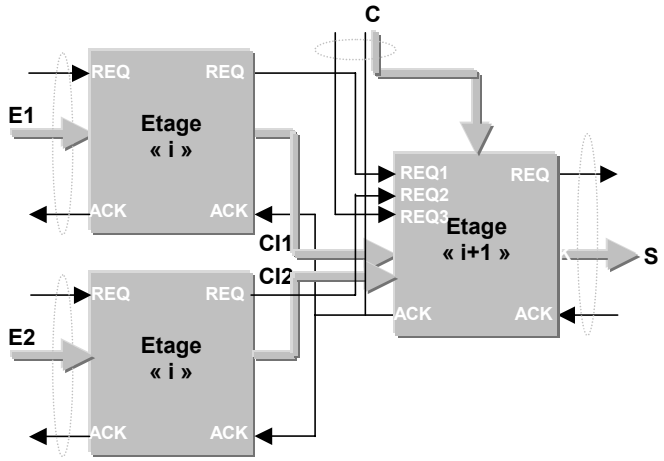
```
 PROCESS Demux_F_proc2
 PORT (CI1 : IN DR ;
       s1: OUT DR)
 VARIABLE x : DR;
 BEGIN
  [ CI1?x ; s1!F1(x ) ; LOOP];
 END;

 PROCESS Demux_F_proc3
  PORT (CI2 : IN DR ;
        s2: OUT DR)
 VARIABLE x : DR;
 BEGIN
  [ CI2?x ; s2!F2(x ) ; LOOP];
 END;

 END;
```

Figure. 3.5 : Pipeline Asynchrone Non Linéaire : Démultiplexeur (*Split*)

Un multiplexage (figure 3.6) est une convergence où le signal de « choix » vient d'un autre étage du pipeline et permet de contrôler le canal entrant qui doit être lu.




```
COMPONENT Mux_F
PORT (e1 : IN DR ;
      e2 : IN DR;
      c : IN DR;
      s: OUT DR)
CHANNEL CI1, CI2 : DR ;
BEGIN
 PROCESS Covergence_F_proc1
 PORT (CI1 : IN DR ;
       CI2 : IN DR;
       c : IN DR;
       s: OUT DR)
 VARIABLE x,y : DR;
 BEGIN
  [c?x;
    @[ x=0 => CI1?y ; s!F(y) ; BREAK
       x=1 => CI2?y ; s!F(y) ; BREAK
     ]
  ] LOOP;
 END;
```

```
 PROCESS Mux_F_proc2
 PORT (e1 : IN DR ;
       CI1: OUT DR)
 VARIABLE x : DR;
 BEGIN
  [ e1?x ; CI1!x ; LOOP];
 END;

 PROCESS Mux_F_proc3
 PORT (e2 : IN DR ;
       CI2: OUT DR)
 VARIABLE x : DR;
 BEGIN
  [ e2?x ; CI2!x ; LOOP];
 END;

 END;
```

Figure. 3.6 : Pipeline Asynchrone Non Linéaire : Multiplexeur (*Merge*)

3.1.3.3 Problèmes associés au traitement des fourches et convergences

On rencontre deux problèmes majeurs dans la conception des circuits asynchrones non linéaires. Il s'agit de la synchronisation d'un étage de pipeline avec de multiples destinations (cas des fourches), et avec de multiples sources (cas des convergences).

3.1.3.3.1 *Environnement lent ou blocage dans les fourches*

Dans les pipelines linéaires tels que le PS0 [WIL 91], ou les pipelines « look-ahead » [SIN 00] et ceux de grandes capacités [SIN 00b] on suppose que tous les étages réagissent assez rapidement aux impulsions d'acquittement [WIL 91]. Nous disons rapidement car dans ce type de pipeline, certains acquittements entre les étages sont essentiellement des impulsions chronométrées et font que les communications inter-étages ne sont pas persistantes. A titre

d'exemple lorsqu'un étage du pipeline affirme son signal d'acquittement, entraînant une précharge de l'étage précédent, il suppose que cette précharge est assez rapide mais ne procède pas à la vérification de la fin de la précharge avant de désactiver son signal d'acquittement. Cette « supposition de précharge rapide » est en général satisfaite dans les pipelines linéaires.

Toutefois, dans les pipelines non linéaires tels que les fourches la non persistance des communications inter-étages peut constituer une contrainte importante. Les signaux d'acquittement multiples, parce qu'ils sont des impulsions pouvant être non recouvrantes, ne peuvent par conséquent être fusionnés en utilisant une porte de rendez-vous simple.

Dans la figure 3.7 qui représente une fourche à deux sorties, l'étage de fourche E1 contrôle les étages E2 et E3. Pour un fonctionnement correct, E1 doit recevoir les deux signaux d'acquittement des étages E2 et E3. Or si l'étage E3 est retardé par rapport à l'étage E2 (environnement lent), alors l'acquittement de E3 pour E1 est retardé également, pendant que l'étage E2 envoie son acquittement (non persistant) à E1. Dans ce cas de figure, il est clair que les deux acquittements de E2 et E3 pour E1 ne peuvent arriver en même temps, et que la porte de Muller qui se trouve dans la fourche –précisément pour synchroniser les acquittements de E2 et E3 vers E1- ne peut produire le signal de précharge pour E1. Il faut donc absolument s'assurer que les acquittements des étages de sorties, soient synchronisés.

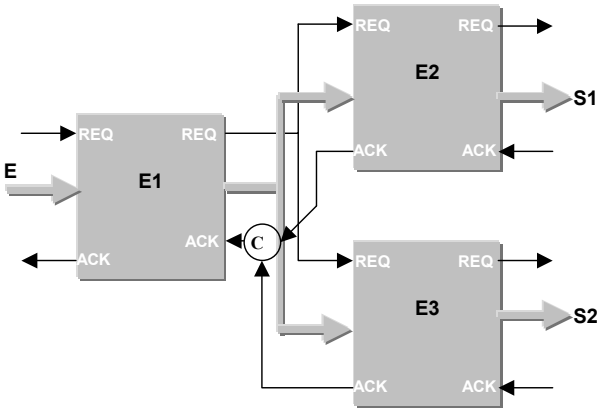


Figure. 3.7 : Synchronisation de Fourche à 2 sorties via une porte de Muller

3.1.3.3.2 *Environnement lent ou blocage dans les convergences*

Le deuxième problème majeur dans la conception de circuits asynchrones non linéaires est celui de la synchronisation des canaux d'entrée dans une convergence.

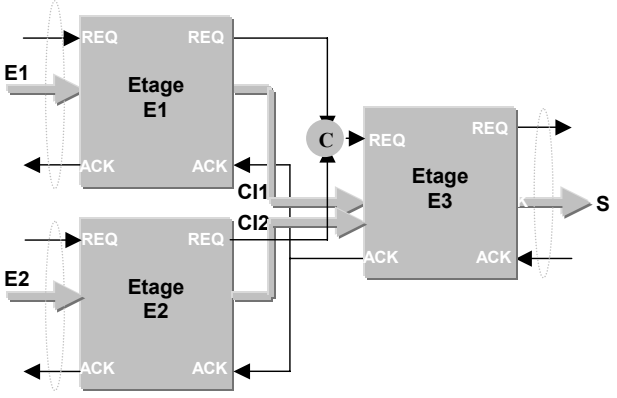


Figure. 3.8 : Synchronisation des canaux d'entrée dans une convergence

Dans la figure 3.8 qui représente une convergence à deux entrées, les deux étages de convergence E1, E2 fusionnent dans l'étage E3. En supposant que ce dernier soit un bloc de fonction « gourmand » il peut dès la réception de données provenant de E1 (sans attendre ceux provenant de E2 supposé lent-) générer une donnée à sa sortie. Par conséquent E2 va recevoir trop tôt un acquittement de l'étage E3. Un scénario plausible est que la sortie de l'étage lent sera traitée par E3 comme une donnée non désirée, et empêchera de la sortie la synchronisation entre les étages du pipeline. Il s'agit en fait de s'assurer que les requêtes des étages de convergence soient synchronisées.

3.2 Travaux antérieurs

La conception de circuits asynchrones, au delà du fait qu'elle constitue une alternative à la conception de circuits synchrones -qui posent notamment des problèmes liés aux différences des temps de propagation- permet également d'assurer une interface robuste avec des composants plus lents. Plusieurs pipelines asynchrones rapides à grain fin ont été proposés pour la conception de circuits asynchrones à grande vitesse, tels que IPCMOS [SCH 00], GasP [SUT 01][COA 01] et les circuits en mode impulsion [NYS 01]. Ces conceptions fournissent des circuits très rapides mais elles font des hypothèses temporelles très fortes qui se traduisent par des contraintes rigoureuses sur la taille des transistors et sur la vérification « post-layout ».

Singh et Nowick ont récemment proposé certains pipelines asynchrones logiques dynamiques [SIN 00][SIN 00b] et statiques [SIN 01], qui offrent un haut débit et une latence basse. Ces pipelines peuvent atteindre une performance équivalente avec moins d'hypothèses temporelles. Les pipelines dynamiques n'ont été introduits que pour les chemins de données linéaires (sans fourches ni convergence), bien que certaines solutions soient esquissées dans [SIN 00b] pour manipuler les convergences. [SIN 00] présente également une approche pour traiter les environnements lents ou les blocages dans un nombre limité de cas de pipelines linéaires. Les approches proposées par Singh et Nowick n'abordent toutefois pas les problèmes de synchronisation qui surgissent lors de l'utilisation de fourches ou de convergences.

Les divers modèles de pipelines se caractérisent par le temps de latence, le temps de cycle, et la robustesse de la synchronisation. Le modèle QDI proposé par Martin et Lines [LINE 98] est le modèle le plus robuste dans lequel la correction d'un circuit asynchrone est garantie indépendamment du délai dans les fils. La communication entre processus est réalisée en utilisant l'encodage double rails ou plus généralement l'encodage « 1 parmi N ». Les circuits asynchrones basés sur ces modèles sont faciles à concevoir et ont un rendement acceptable grâce au pipeline à grain fin et au parallélisme [MAR 97]. Ces modèles de pipelines sont traités dans [DIN 03].

Un autre modèle de pipeline est le type micropipeline ou « données groupées » dans lequel les chemins de données synchrones sont utilisés conjointement avec des signaux de requêtes et acquittement [SUT 89]. Ces modèles de pipelines ont l'avantage d'être moins gros en surface, de consommer moins que les autres modèles, et de tirer profit des méthodes de conception de circuits synchrones pour les chemins de données.
Beerel propose une approche qui est une extension des travaux de Berkel [BER 96]. Cette approche décline certains modèles de pipelines qui ciblent l'encodage double rails ou « 1 parmi N » et l'encodage de type « données groupées » [FER 02], [OZDA02], [OZDA02b], et [TUGS02]. Ces modèles offrent des pipelines à grain fin à haute vitesse et des pipelines

linéaires et non linéaires. Leur inconvénient est que lors de la synthèse automatisée ils demandent une bibliothèque de cellules complexes. [FERR02] propose une approche dite « single-track » dans laquelle un canal encodé par un codage « 1 parmi N » est utilisé pour porter des données tout en servant également pour le signal d'acquittement.

3.3 Modèle et structures de circuit cible

Dans ce qui précède nous avons cité quelques types de circuits implémentant le mécanisme de pipeline dans la conception de circuits asynchrones. Caltech [LINE98] propose différents types de circuits implémentant le pipeline QDI. Ces différents types de circuits QDI, qui constituent donc autant de modèles, se distinguent par leurs protocoles de communication qui sont soit séquentiel, soit WCHB, soit PCHB, soit PCFB [LINE98].

Dans le cadre de cette thèse nous nous intéressons au type micropipeline [SUTH89]. Nous allons dans ce qui suit présenter les divers modèles de circuit micropipeline en déclinant leurs structures correspondantes. Pour chaque modèle, défini par un protocole particulier (séquentiel, WCHB, PCHB ou PCFB), on présente aussi bien les structures linéaires que non linéaires. L'ensemble des structures linéaires et non linéaires sert à construire des circuits micropipeline avec le protocole correspondant. En particulier un circuit asynchrone est construit en composant les structures pipelines de base.

Un circuit asynchrone compatible avec la spécification DTL (cf. §2.3) est toujours implémenté par un bloc de calcul et un bloc de contrôle. Le bloc de calcul est combinatoire et s'occupe de calculer les sorties et les états suivants. Le bloc de contrôle appelé « buffer » s'occupe lui de synchroniser le circuit avec les autres parties du système asynchrone globale, notamment en respectant le protocole de communication type « poignée de main ».

D'un point de vue architectural les blocs de contrôle et de calcul sont mis en cascade, avec deux variantes possibles : le buffer est placé soit en sortie du circuit soit à son entrée (figure 3.9).

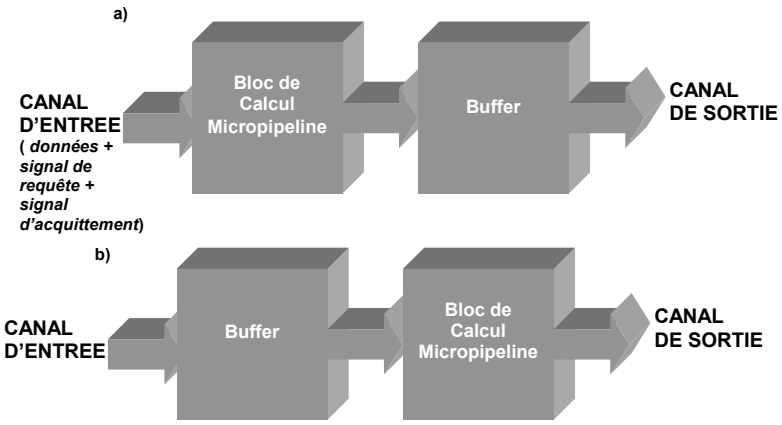


Figure. 3.9 : Implémentation de circuits asynchrones micropipeline compatibles DTL
c) buffer en sortie de circuit
b) buffer en entrée de circuit

Une représentation plus fine du mircopipeline de la figure 3.9 est donnée en figure 3.10

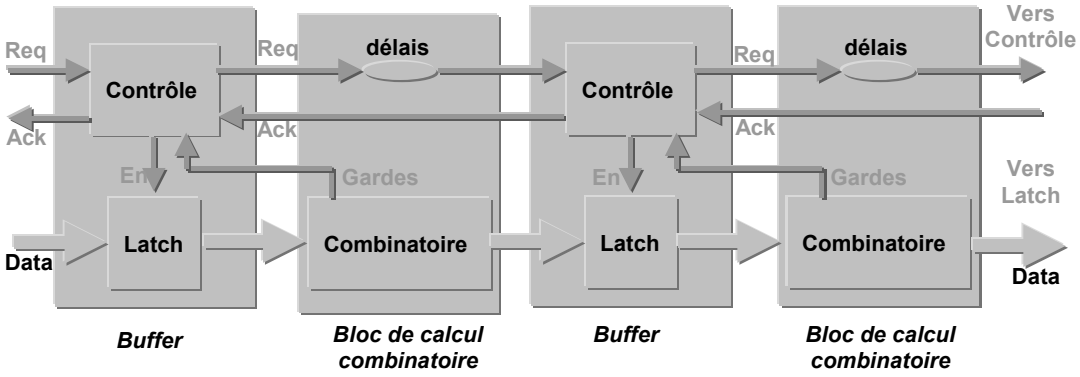


Fig. 3.10 : Schéma bloc d'un micropipeline (ou pipeline bundled data) simple.

La figure 3.10 montre une FIFO où l'on a intercalé des circuits combinatoires (ou blocs de fonctions) entre les latchs se trouvant dans les « buffers ». Le « buffer » de la figure 3.9 est constitué du contrôle (protocole « poignée de main ») ainsi que du latch. Le bloc de calcul combinatoire de la figure 3.9 englobe quand à lui le bloc fonction ainsi que les éléments de délais. Par ailleurs, ce circuit peut être vu comme un chemin de donnée synchrone classique, constitué de latchs et de circuits combinatoires séquencés par une pseudo horloge à fréquence variable. Il peut également être vu comme une structure asynchrone flot de données composée de deux types de « handshake components » (librairie de composants asynchrones) : les latchs et les blocs de fonctions [SPA].

Notre modèle de circuit cible micropipeline est donné en figure 3.11. En effet, cette représentation est dictée par le fait que la partie chemin de donnée -qui contient les latchs en plus des fonctions combinatoires- sont synthétisés selon la méthode synchrone, alors que la partie contrôle doit être synthétisée par la méthode que nous présentons ultérieurement. C'est pourquoi pour le micropipeline, l'on sépare le flot de synthèse des contrôleurs de celui des chemins de données. Dans le chapitre consacré à la synthèse, nous montrons que cette séparation ou *split* s'effectue lors de la génération des équations de dépendances.

En micropipeline, le latch sert à ne considérer le flot de calcul que lorsque ce dernier est terminé. Le signal de fin de calcul est déterminé par des hypothèses temporelles (délais) incluses dans le contrôleur. Ces hypothèses dépendent du bloc de calcul combinatoire et

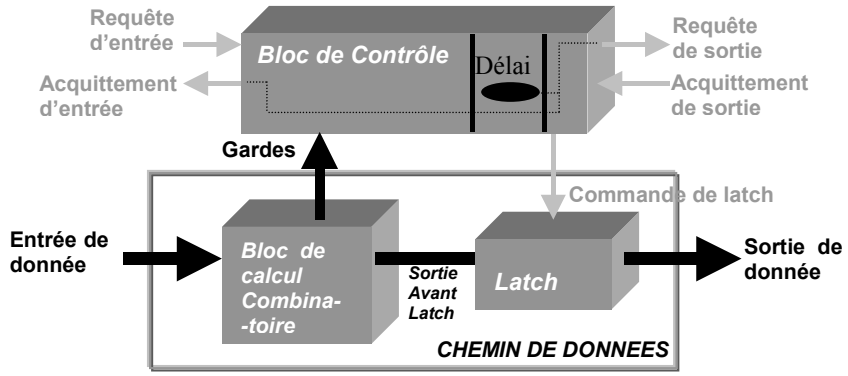


Fig. 3.11 : Schéma bloc de la structure du circuit micropipeline cible

servent également à adapter les signaux de synchronisation (requêtes et acquittements) en fonction des éléments de calcul. Par ailleurs, en plus des sorties de données, la fonction combinatoire en micropipeline calcule les gardes booléennes (condition de sélection ou choix). Une fois calculées, ces gardes sont transmises au contrôleur.

Nous présentons dans ce qui suit les différentes structures micropipelines linéaires et non linéaires pour l'implémentation du contrôle telle que définie en figure 3.11 c'est à dire incluant les éléments de délais.

3.3.1 Structure pour micropipelines linéaires

La figure 3.12 représente un « demi-buffer » agrémenté d'un bloc fonction « F ». Nous utilisons cet exemple pour discuter l'implémentation des structures linéaires selon le protocole de communication sélectionné.

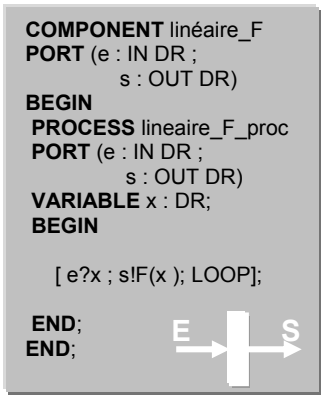


Fig. 3.12 : Schéma et Code CHP descriptif d'un circuit asynchrone linéaire

3.3.1.1 Structure séquentielle

Le protocole séquentiel est utilisé pour implémenter des blocs fonctionnels logiques qui n'ont pas besoin de synchronisation type « poignée de main » entre l'entrée et la sortie. La communication entre l'entrée et la sortie est transparente.

La figure 3.13 montre l'implémentation en protocole séquentiel du circuit de la figure 3.12. Le bloc fonctionnel se charge du calcul de la sortie. Le signal de requête de la sortie est directement relié au signal de requête d'entrée via un élément de retard qui représente le délai nécessaire au calcul de la sortie S.
Le signal d'acquittement de l'entrée est directement relié au signal d'acquittement de la sortie.

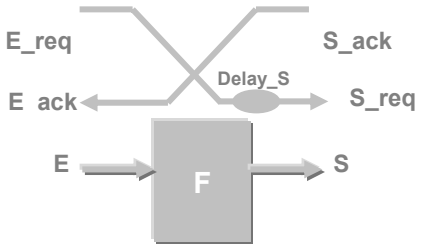


Fig. 3.13 : Micropipeline séquentiel d'une structure linéaire

Ce protocole est le plus séquentiel. La donnée sur le canal E est transmise de l'entrée vers la sortie, puis n'est acquittée que quand la sortie a été acquittée. Cette séquence est valable aussi bien pour la phase de calcul que pour la phase de remise à zéro. Par conséquent, le protocole séquentiel est le protocole le plus lent en temps de cycle puisque tout est séquentiel.

3.3.1.2 Structure WCHB (« Weak Condition Half-Buffer »)

Le protocole WCHB (avec les protocoles PCHB et PCFB que l'on voit plus loin) est issu de solutions de ré-ordonnancement des étapes d'une communication entre un canal de sortie et un canal d'entrée. Ces solutions reposent sur l'expansion des communications dans les canaux conformément à un protocole quatre phases [LINE95].

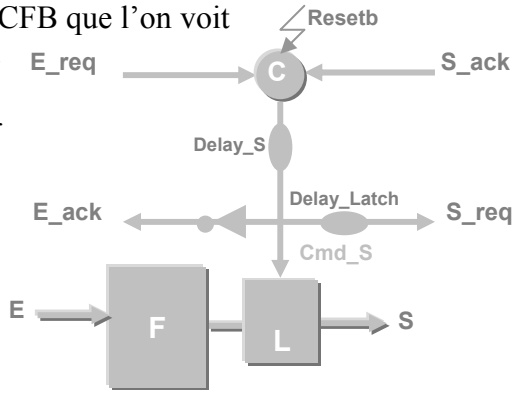


Fig. 3.14 : Micropipeline WCHB d'une structure linéaire

Dans le protocole WCHB, il y a synchronisation des phases montantes (mise à un) et descendantes (remise à zéro) entre les canaux d'entrée et de sortie.

La figure 3.14 montre l'implémentation en protocole WCHB du circuit 3.12. Notons que les signaux d'acquittement E_ack et S_ack sont actifs au niveau bas. Au départ après le signal de reset, les signaux de requêtes E_req et S_req sont au niveau bas et les signaux d'acquittement E_ack et S_ack sont au niveau haut. Puis lors de l'arrivée de la donnée, E_req passe au niveau haut. Le signal à la sortie de la porte de Muller passe alors au niveau haut puis est retardé par un délai équivalent au temps nécessaire au calcul de la sortie S. Après cela et de façon concurrente, le signal d'acquittement d'entrée E_ack descend grâce à l'inverseur, le signal de requête de la sortie S_req monte après avoir été retardé d'un délai correspondant au passage des données au travers du latch L, et le signal de commande de latch déclenche le fonctionnement de ce dernier. Ensuite l'environnement côté sortie acquitte la donnée reçue en faisant descendre le signal S_ack, et l'environnement côté entrée réagit à la descente du signal E_ack en remettant à zéro le signal E_req. Cela provoque le retour à zéro du signal S_req (auquel l'environnement côté sortie réagit en faisant passer S_ack au niveau haut), et le retour à un du signal E_ack. Nous retrouvons ainsi l'état initial du circuit.

Selon [LINE98], puisque les canaux E et S ne peuvent contenir simultanément deux données distinctes, le circuit de la figure 3.12 (avec F fonction unité) s'appelle « demi-buffer » (ou Half Buffer). On remarque que le protocole WCHB assure la symétrie des signaux E_req et S_ack dans les phases montante et descendante de la sortie de ce demi-buffer. Cela permet d'avoir une implémentation régulière de la partie contrôle d'un circuit asynchrone.

3.3.1.3 Structure PCHB (« PreCharge Half-Buffer »)

La phase montante du protocole PCHB est identique à celle du protocole WCHB. Par contre, il y a désynchronisation au niveau des remises à '0' des signaux de requête et d'acquittement, pour autoriser la communication au niveau de la sortie ***indépendam--ment*** des entrées et ainsi gagner en vitesse dans le pipeline. En effet, il y a d'abord remise à zéro du canal de sortie, puis remise à zéro du canal d'entrée. En d'autres termes, il n'est plus obligatoire d'attendre la remise à zéro des requêtes d'entrée pour terminer les autres protocoles en sortie. De la sorte, le gain de vitesse devient significatif lorsque le nombre de canaux d'entrée est important.

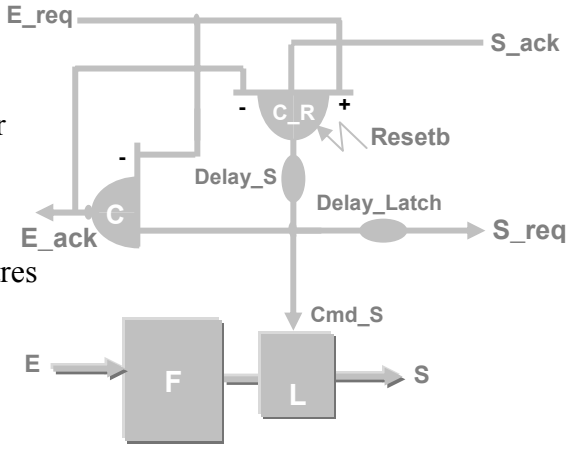


Fig. 3.15 : Micropipeline PCHB d'une structure linéaire

La figure 3.15 montre l'implémentation en protocole PCHB du demi-buffer de la figure 3.12. Au départ après le signal de reset, les signaux de requêtes E_req et S_req sont au niveau bas et les signaux d'acquittement E_ack et S_ack sont au niveau haut. Puis lors de l'arrivée de la donnée, E_req passe au niveau haut. Le signal à la sortie de la porte de Muller dissymétrique C_R passe alors au niveau haut puis est retardé par un délai équivalent au temps nécessaire au calcul de la sortie S. Après cela et de façon concurrente, le signal d'acquittement d'entrée E_ack descend grâce à la Muller dissymétrique inversée C, le signal de requête de la sortie S_req monte après avoir été retardé d'un délai correspondant au passage des données au travers du latch L, et le signal de commande de latch déclenche le fonctionnement de ce dernier. Ensuite l'environnement côté sortie acquitte la donnée reçue en faisant descendre le signal S_ack, ce qui a pour effet de faire passer la sortie S_req au niveau bas (auquel l'environnement côté sortie réagit en faisant passer S_ack au niveau haut). Enfin, l'environnement côté entrée réagit à la descente du signal E_ack en remettant à zéro le signal E_req. Cela provoque le retour à « 1 » du signal E_ack. Nous retrouvons de la sorte l'état initial du circuit.

3.3.1.4 Structure PCFB (« PreCharge Full-Buffer »)

La phase montante du protocole PCFB est identique à celle du protocole WCHB.
Par contre dans la phase de remise à zéro, on découple totalement les canaux d'entrée et de sortie. Les deux communications sur les entrées et les sorties se font de façons totalement concurrentes. Ce découplage offre de meilleures performances au niveau vitesse mais l'implémentation de ce protocole devient plus coûteuse, car pour respecter les conditions de codage complet il faut insérer une variable d'état (En) afin de pouvoir distinguer dans quelle partie du protocole on se trouve.

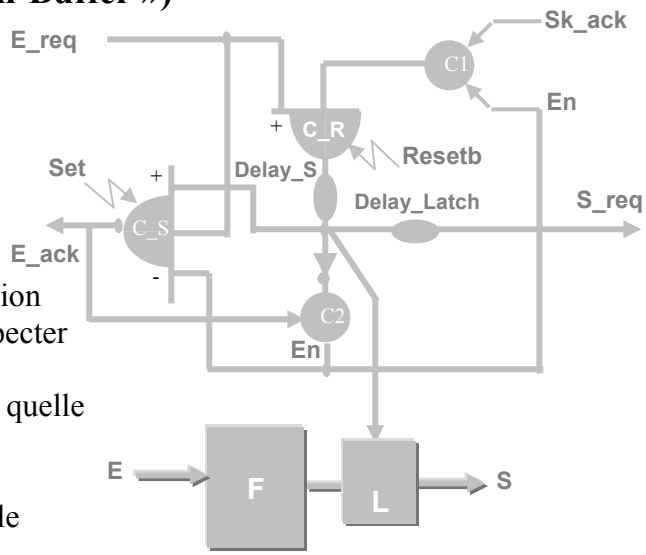


Fig. 3.16 : Micropipeline PCFB d'une structure linéaire

La figure 3.16 montre l'implémentation en protocole PCFB du demi-buffer de la figure 3.12.

Au départ après le signal de reset, les signaux de requêtes E_req et S_req sont au niveau bas, les signaux d'acquittement E_ack et S_ack ainsi que le signal en sortie de la porte de Muller C2 (En) sont au niveau haut. Puis lors de l'arrivée de la donnée, E_req passe au niveau haut. Le signal à la sortie de la porte de Muller dissymétrique C_R passe alors au niveau haut puis est retardé par un délai équivalent au temps nécessaire au calcul de la sortie S. Après cela et de façon concurrente, le signal d'acquittement d'entrée E_ack descend grâce à la Muller dissymétrique inversée C_S, le signal de requête de la sortie S_req monte après avoir été retardé d'un délai correspondant au passage des données au travers du latch L, le signal en sortie de la Muller C2 passe à « 0 », et le signal de commande de latch déclenche le fonctionnement de ce dernier. Ensuite l'environnement côté sortie acquitte la donnée reçue en faisant descendre le signal S_ack, ce qui a pour effet de faire passer la sortie S_req au niveau bas (auquel l'environnement côté sortie réagit en faisant passer S_ack au niveau haut). De façon concurrente à l'action de l'environnement en sortie, l'environnement côté entrée réagit à la descente du signal E_ack en remettant à zéro le signal E_req. Cela provoque le retour à « 1 » du signal E_ack et du signal en sortie de C2 (En). Nous retrouvons de la sorte l'état initial du circuit.

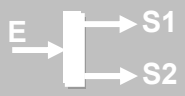
3.3.2 Structure pour micropipelines non linéaires

Pour discuter de l'implémentation des structures non linéaires selon les différents protocoles nous présentons quatre structures de base qui permettent la conception de façon automatique d'un circuit asynchrone micropipeline. Il s'agit des structures de fourche, de convergence, de multiplexage, de démultiplexage. L'opérateur de probe (sonde de canal) est traité dans le chapitre consacré à la synthèse des contrôleurs.

3.3.2.1 Structure séquentielle

La figure 3.17 montre une fourche à deux sorties implémentée avec le protocole séquentiel.

```
COMPONENT Fourche_F
PORT (e : IN DR ;
        s1 : OUT DR;
        s2: OUT DR)
BEGIN
 PROCESS Fourche_F_proc
 PORT (e : IN DR ;
        s1 : OUT DR;
        s2: OUT DR)
 VARIABLE x : DR;
 BEGIN
  [ e?x ; s1!F1(x ) ; s2!F2(x ) ; LOOP];
 END;
END;
```



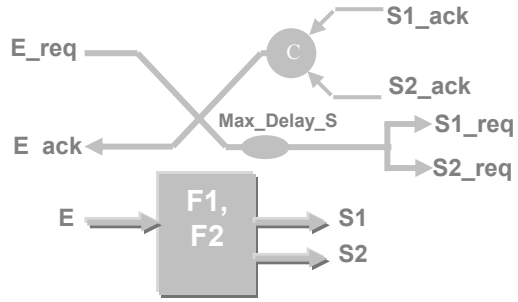
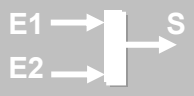


Fig. 3.17 : Micropipeline séquentiel de la fourche

Les fourches à plus de deux sorties sont implémentées de façon similaire. Comme en §3.3.1.1, les requêtes de sortie et les acquittements d'entrée sont directement obtenus, respectivement des requêtes d'entrée et des acquittements de sortie. Seulement, il faut résoudre le problème de blocage ou de ralentissement dans l'un des canaux des sorties grâce à une porte de Muller (C) qui synchronise les deux signaux d'acquittement de sortie. De la sorte le canal de sortie n'est acquitté que quand les deux sorties ont reçu la donnée.
Une autre différence de l'implémentation de la fourche est que le retard « max_delay_S » dans la ligne de la requête de sortie représente le temps maximum nécessaire au calcul des deux sorties S1 et S2.

Le circuit dual de la fourche, est la structure de type convergence dont l'implémentation en protocole séquentiel est donnée figure 3.18.

```
COMPONENT Convergence_F
PORT (e1 : IN DR ;
        e2 : IN DR;
        s: OUT DR)
BEGIN
 PROCESS Covergence_F_proc
 PORT (e1 : IN DR ;
        e2 : IN DR;
        s: OUT DR)
 VARIABLE x,y : DR;
 BEGIN
  [ e1?x ; e2!y ; s!F(x,y ) ; LOOP];
 END;
END;
```



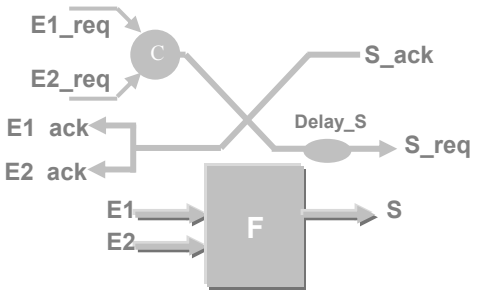
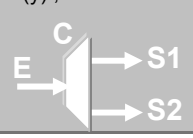


Fig. 3.18 : Micropipeline séquentiel de la convergence

Cette structure agit comme une jonction qui synchronise les deux canaux d'entrée. L'envoi de la donnée sur la sortie S implique la présence des données sur les entrées E1 et E2. Cette exigence est assurée par la porte de Muller C à l'entrée de la partie contrôle. Par ailleurs, les deux signaux d'acquittement d'entrée E1_ack et E2_ack proviennent tous deux du canal de sortie S. Cela est justifié par la contribution concurrente des canaux E1 et E2 à la génération de S.

Les structures de multiplexage (figure 3.20) et de démultiplexage (figure 3.19) sont plus complexes que les structures de fourche et de convergence. Un canal de sélection C est nécessaire pour choisir la sortie à émettre dans le cas du démultiplexeur et l'entrée à propager dans le cas du multiplexeur. Ce canal de sélection est également codé en « donnée groupée ». Ce qui signifie qu'il dispose en plus du bus de donnée, de signaux de requête et d'acquittement.

```
COMPONENT Demux_F
PORT (e : IN DR ;
        c : IN DR ;
        s1 : OUT DR;
        s2: OUT DR)
BEGIN
 PROCESS Demux_F_proc
 PORT (e : IN DR ;
         c : IN DR ;
        s1 : OUT DR;
        s2: OUT DR)
 VARIABLE x,y : DR;
 BEGIN
  [c?x ; e?y ;
    @[ x=0 => s1!F(y) ; BREAK
       x=1 => s2!F(y) ; BREAK
     ]
  ] LOOP;
 END;
END;
```



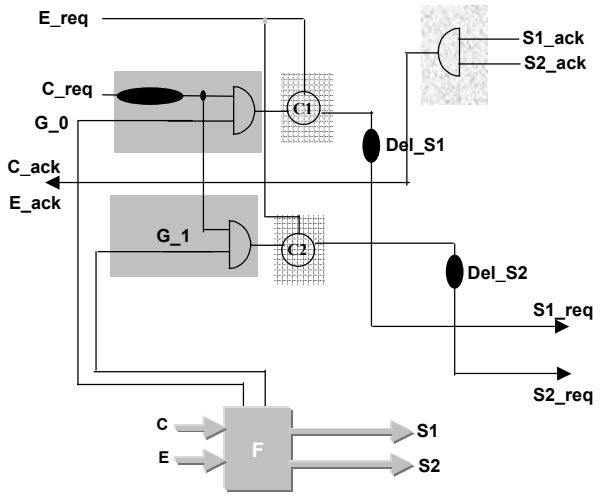
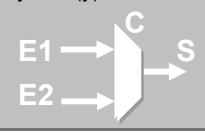


Fig. 3.19 : Micropipeline séquentiel d'un démultiplexeur

```
COMPONENT Mux_F
PORT (c : IN DR ;
        e1 : IN DR ;
        e2 : IN DR;
        s: OUT DR)
BEGIN
 PROCESS Mux_F_proc
PORT (c : IN DR ;
        e1 : IN DR ;
        e2 : IN DR;
        s: OUT DR)
 VARIABLE x,y : DR;
 BEGIN
  [c?x ;
    @[ x=0 => e1?y ; s!F(y) ; BREAK
       x=1 => e2?y ; s!F(y) ; BREAK
     ]
  ] LOOP;
 END;
END;
```



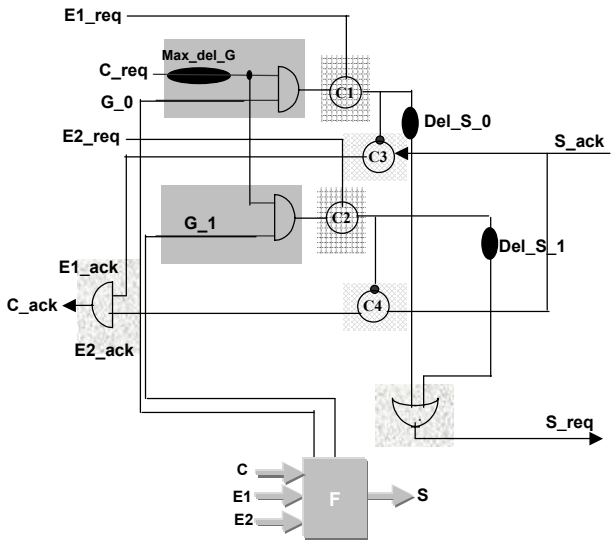


Fig. 3.20 : Micropipeline séquentiel d'un multiplexeur

Le calcul de la donnée de sélection peut être complexe et par conséquent nécessiter tout un bloc de calcul. Ce dernier est compris dans le chemin de donnée synthétisé en synchrone. Nous développons cet aspect ultérieurement dans la partie réservée à la synthèse du chemin de donnée.

L'implémentation de la partie du circuit de contrôle concernant la gestion de la requête de sélection (implémentation des gardes, cf. §5.2.2) se traduit par une porte logique « AND » (rectangles gris sur les figures 3.19 et 3.20). Les portes de Muller C1 et C2 servent à synchroniser les canaux d'entrée et jouent le même rôle que la porte de Muller C dans la figure 3.18.

Pour le mutiplexeur les portes de Muller C2 et C4 nous permettent de n'acquitter que le canal d'entrée sélectionné (soit E1 soit E2). Les inverseurs placés sur l'entrée de ces portes de Muller sont justifiés par le fait que les signaux d'acquittement sont actifs niveau bas. Par ailleurs, la porte OR en sortie du multiplexeur (figure 3.20) permet de sélectionner de façon exclusive l'une des deux requêtes d'entrée (donc l'une des données d'entrée). Cette exclusivité est garantie par l'exclusivité dans le calcul des signaux de sélection ou choix par la partie « chemin de donnée ».

Pour le démultiplexeur, c'est l'exclusivité des canaux de sortie qui garantie qu'un et un seul des signaux d'acquittement de sortie est acquitté. Comme les signaux d'acquittement sont actifs niveau bas, on utilise pour cela une porte logique « AND » dont la sortie représente l'acquittement C_ack du canal de sélection (figure 3.19).

3.3.2.2 Structures WCHB

La figure 3.21 représente une fourche et une convergence implémentées avec le protocole WCHB. Contrairement au protocole séquentiel, les données en protocole WCHB ne sont envoyées dans les canaux de sorties que quand ces derniers sont disponibles. Ainsi le signal de requête de la donnée entrante E_req doit attendre le signal d'acquittement de la sortie S_ack avant de générer les signaux de requête de sortie S_req et d'acquittement d'entrée E_ack. Autant pour la fourche que pour la convergence c'est la porte de Muller C1 qui assure cette contrainte pour respecter le protocole de type « poignée de main » à la sortie.

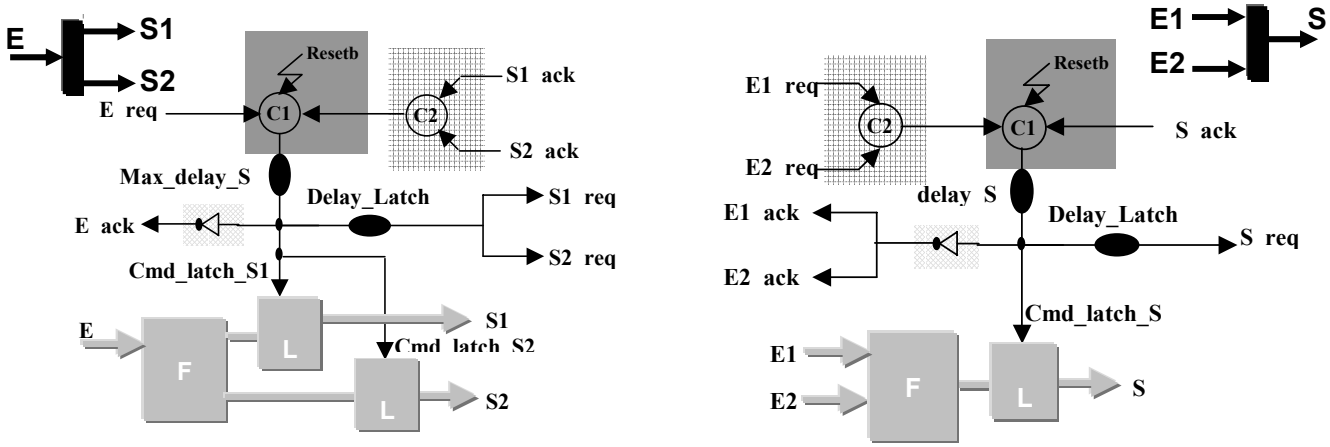


Fig. 3.21 : Micropipeline de fourche et Micropipeline de convergence en protocole WCHB

Les portes de Muller C2 assurent la synchronisation des signaux d'acquittement de sorties pour la fourche et de requête d'entrées pour la convergence. Pour la génération des signaux d'acquittement d'entrée on utilise des inverseurs car ces sont actifs au niveau bas.

L'implémentation de ces structures peut être obtenue de façon généralisée en appliquant les modèles discutées figures 3.9 et 3.10 (couple bloc de calcul / contrôle).

La figure 3.22 illustre l'implémentation en WCHB d'un démultiplexeur et d'un multiplexeur. Soulignons les signaux de sélection (calculs des gardes) produits par le bloc fonctionnel et émis au bloc de contrôle. Les portes « AND » utilisées pour la génération des acquittements, réalisent des « OR » exclusif car les signaux d'acquittement sont actifs niveau bas. La porte « OR » du multiplexeur joue le même rôle pour les signaux de requêtes actifs niveau haut.

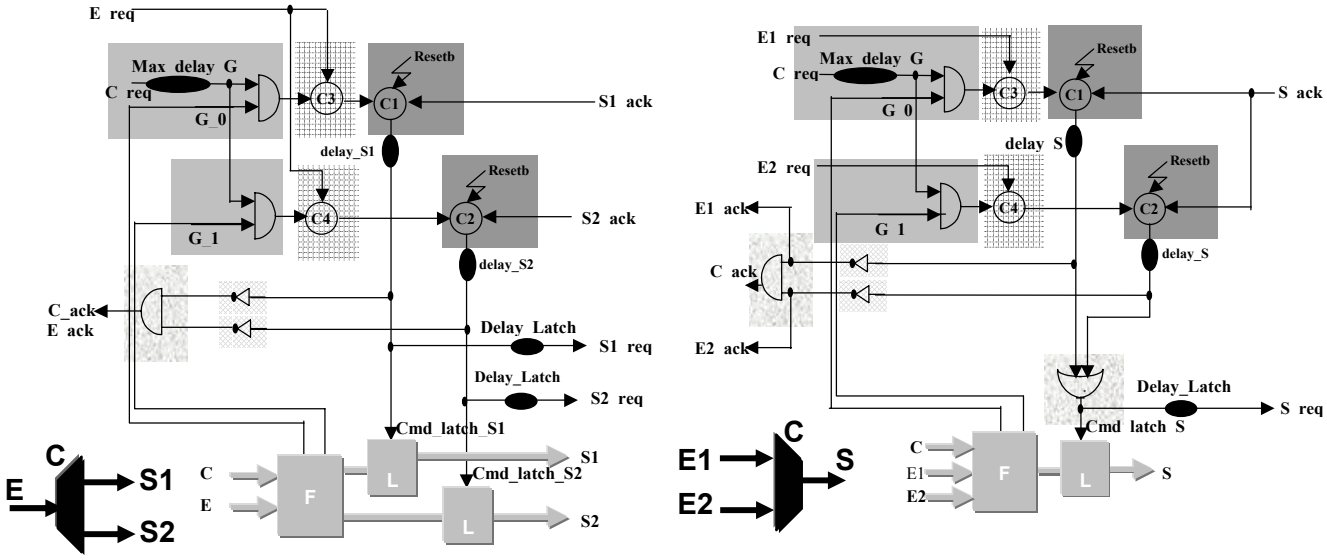


Fig. 3.22 : Micropipeline de démultiplexeur et Micropipeline de multiplexeur en protocole WCHB

3.3.2.3 Structure PCHB

La figure 3.23 représente une fourche et une convergence implémentées avec le protocole PCHB. La différence avec la structure linéaire réside dans les portes de Muller de synchronisation C3.

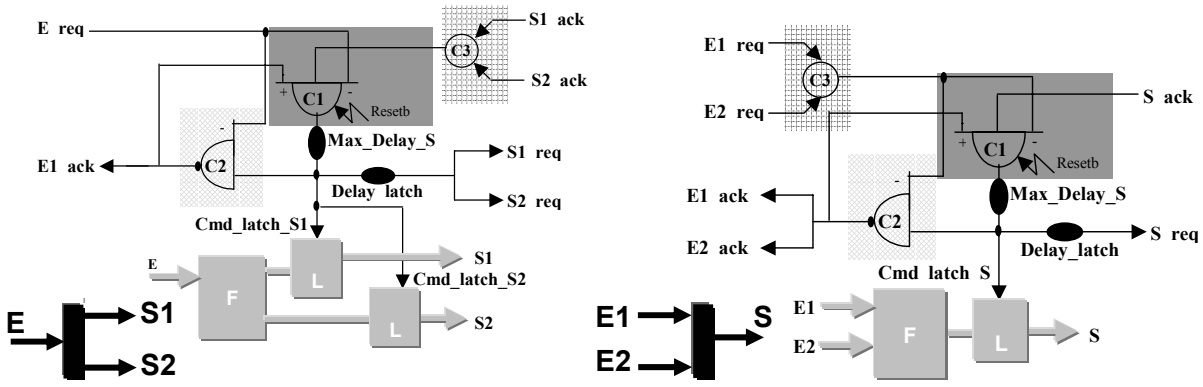


Fig. 3.23 : Micropipeline de fourche et Micropipeline de convergence en protocole PCHB

La figure 3.24 illustre les structures de multiplexage et de démultiplexage en protocole PCHB. De même qu'en WCHB, les signaux de sélection sont produits par la partie « chemin de donnée » et envoyés vers le contrôle. Les portes de Muller C5 et C6 prennent, tout comme dans le cas des structures linéaires la somme des requêtes d'entrée au niveau le plus fin (y compris après la garde ou sélection).

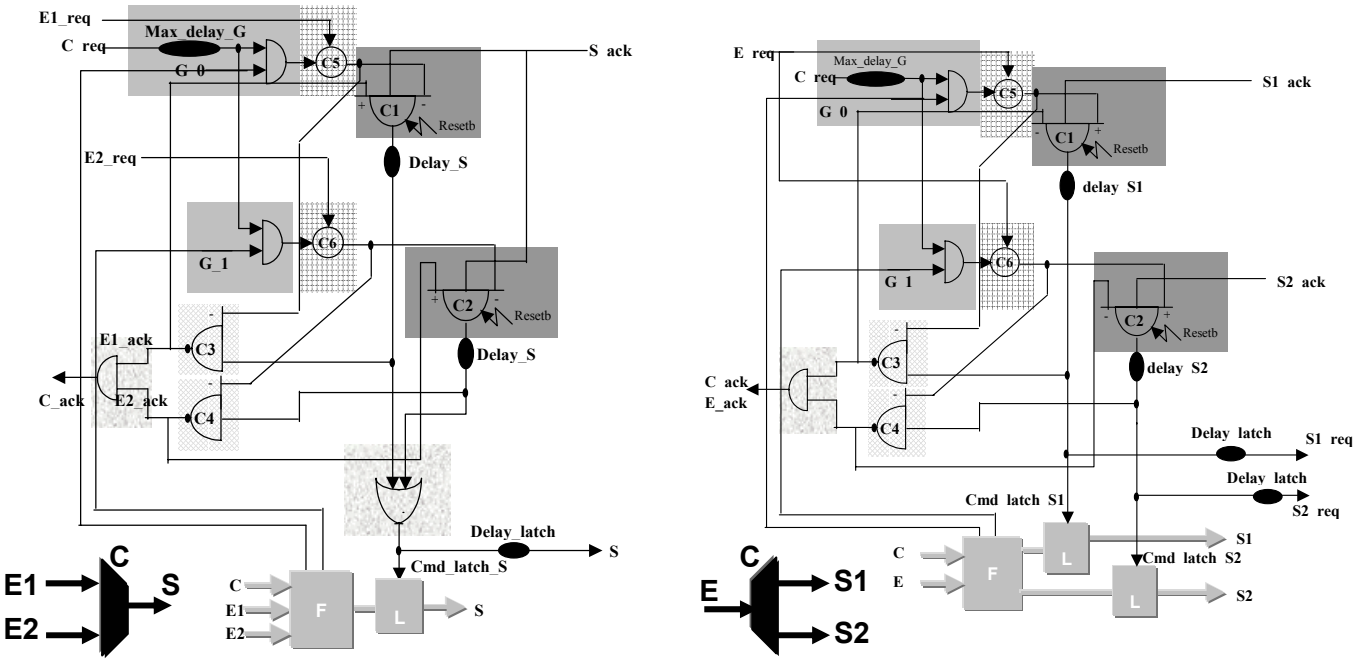


Fig. 3.24 : Micropipeline de démultiplexeur et Micropipeline de multiplexeur en protocole PCHB

3.3.2.4 Structure PCFB

La figure 3.25 illustre les structures de fourche et de convergence en protocole PCFB. De même qu'en WCHB et PCHB, la différence avec la structure linéaire réside dans les portes de Muller de synchronisation (grillage carré).

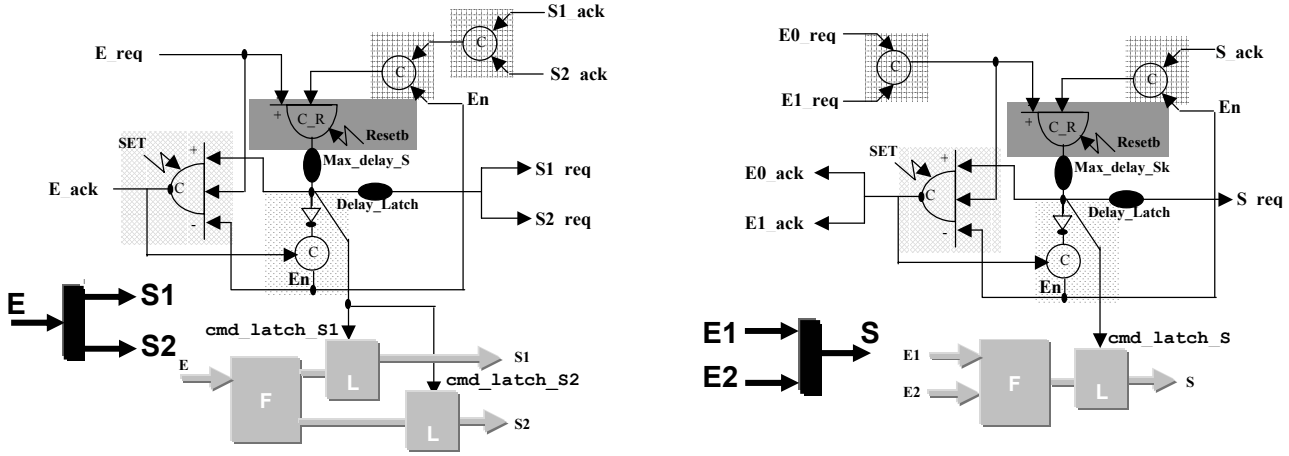


Fig. 3.25 : Micropipeline de fourche et Micropipeline de convergence en protocole PCFB

La figure 3.26 illustre les implémentations des structures de multiplexage et de démultiplexage. De même qu'en WCHB et PCHB, notons les signaux de sélection produits par le bloc de calcul et émis vers le contrôle.

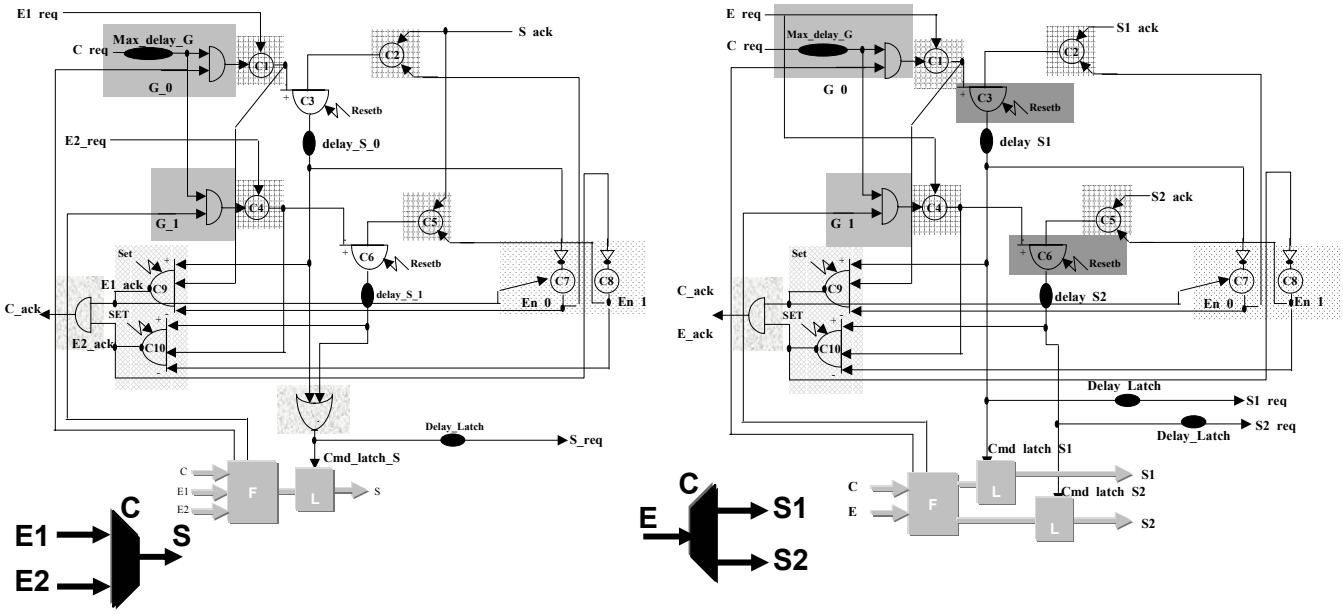


Fig. 3.26 : Micropipeline de démultiplexeur et Micropipeline de multiplexeur en protocole PCFB

3.4 Conclusion

Nous avons exposé dans ce chapitre le fonctionnement du pipeline asynchrone et fait ressortir ses propriétés majeures qui le distingue avantageusement du pipeline synchrone, notamment la faible consommation, le blocage localisé, la capacité de donnée variable. Nous avons identifié pour les structures asynchrones non linéaires, les problèmes liés au temps de propagation dans les différentes branches du pipeline et proposés des solutions de synchronisation réalisables avec une porte de Muller.

Par ailleurs, nous nous intéressons aux pipelines asynchrones avec « données groupées » ou micropipeline qui se distinguent des autres types de pipelines asynchrones cités dans ce chapitre, notamment en étant moins gros en surface, en consommant moins et surtout en permettant de réutiliser les méthodes des circuits synchrones pour la synthèse de la partie « chemin de donnée ». Nous avons ensuite proposé pour le type micropipeline plusieurs modèles de circuits cibles se composant d'une partie de contrôle et d'une partie de chemin de donnée et permettant de synthétiser des circuits asynchrones ciblés micropipeline de façon systématique.

CHAPITRE IV

4. Flot de synthèse et Génération des équations de dépendances

L'implémentation d'un circuit asynchrone relativement à celle d'un circuit synchrone présente plus de contraintes. En effet, en plus de l'exigence classique de la synthèse synchrone qui consiste à obtenir un circuit fonctionnellement équivalent à la spécification, la synthèse asynchrone insensible aux délais doit produire un circuit exempt d'aléa. Dans le cas synchrone, il suffit pour éviter tout changement d'état intempestif, de garantir la stabilité des signaux aux fronts montants de l'horloge, alors qu'en asynchrone, les signaux n'étant pas échantillonnés, tout aléa est potentiellement la source d'une exécution erronée.

Les circuits asynchrones de type micropipeline introduits par [SUTH89] sont composés de parties contrôles insensibles aux délais qui commandent des chemins de données conçus en utilisant le modèle de délai borné. Les problèmes d'aléas dans ces circuits sont écartés en ajoutant des retards sur les signaux de contrôle. En fait, cela est assimilé à un fonctionnement en temps discret dans lequel on n'autorise la mémorisation de données que lorsque ces dernières sont stables à la sortie des parties combinatoires.

Le développement de cette classe de circuits asynchrones avait pour objectif premier de permettre un pipeline élastique. Or, dans la proposition initiale de Sutherland, les délais sont fixes et le circuit effectue le calcul en un temps pire cas. Cette contrainte majeure ne permet pas de tirer profit de la variation dynamique de la chaîne critique des opérateurs de traitement.

C'est pourquoi notre modèle de circuit cible, s'affranchit de cette contrainte en faisant en sorte que les opérateurs combinatoires fournissent leur signal de fin de calcul en fonction des données. De la sorte, on sort du cadre des circuits utilisant le modèle de délai borné et le circuit peut être rendu indépendant de la vitesse puisque la seule hypothèse temporelle consiste à assurer que les données succèdent immédiatement à l'occurrence du signal de contrôle E_req (requête d'entrée d'un étage de pipeline annonçant explicitement l'arrivée d'une donnée). Cette hypothèse peut être rendue équivalente à supposer des délais nuls dans les connexions entre les étages.

Dans ce chapitre nous commençons par présenter le flot de la méthode développée pour synthétiser des circuits asynchrones de type micropipeline. Avant d'expliciter les différentes étapes de cette synthèse jusqu'à la génération des équations de dépendances, nous proposons une vue globale sur la correspondance entre la spécification CHP du circuit à synthétiser et la description VHDL du circuit asynchrone micropipeline généré. A l'entrée du flot, la spécification CHP du circuit est traduite en un arbre syntaxique dont la partie instruction est compilé en une représentation intermédiaire basée sur les réseaux de Petri et les graphes de flots de données. Ensuite, le réseau de Pétri est vérifié de manière à s'assurer qu'il est conforme à la spécification DTL, ce qui garantit une implémentation asynchrone type micropipeline. Le PN est alors simplifié par fractionnement en plusieurs sous-ensembles de PN chacun représentant un choix déterminé de la structure de sélection, puis nous procédons à la génération des équations de dépendances sans considérer un quelconque protocole ou une quelconque technologie cible. Ces équations expriment les relations de causalité entre les canaux de sortie et les canaux d'entrée du circuit. Nous généralisons cette représentation en équations de dépendances à un circuit asynchrone quelconque (compatible DTL).

4.1 Flot détaillé de synthèse de circuits asynchrones micropipeline

La méthode de synthèse développée dans le cadre de cette thèse est représentée par le flot donné en figure 4.1. Le code CHP est analysé par un compilateur (lexique et syntaxe) qui génère alors un arbre syntaxique.

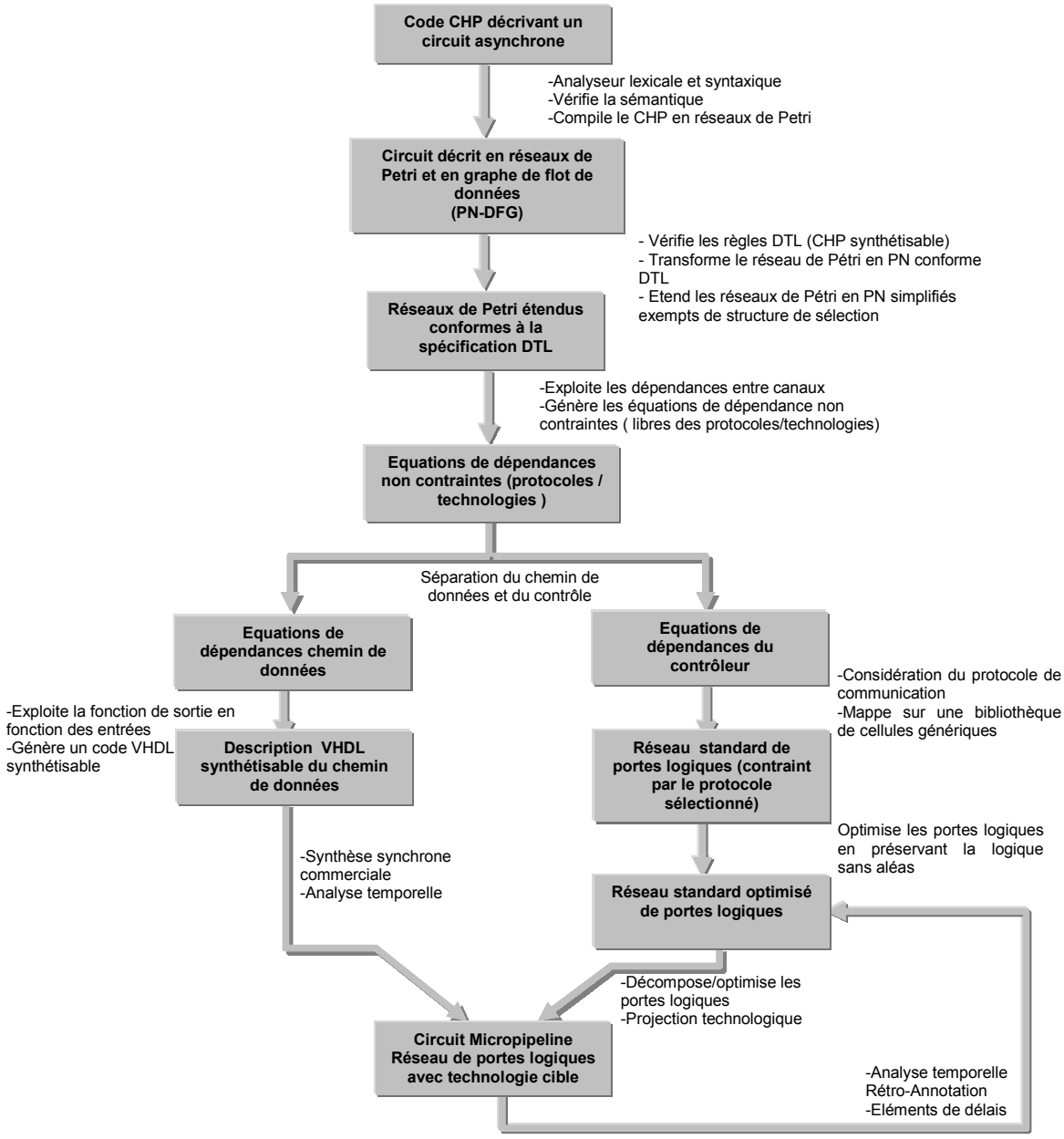


Fig. 4.1 : Flot détaillé de la synthèse de circuits asynchrones Micropipeline

Cet arbre est testé par des règles de sémantique prédéfinies qui vérifient la correction sémantique de la spécification. Puis la partie instruction du code, exprimée en structure arboresecente, est traduite en structure de graphe à nœuds hétérogènes (réseau de pétri + graphe de flots de données), ce qui permet de spécifier le comportement du circuit (contrôle)

autant que la dépendance des données (chemin de données) entre les canaux du circuit asynchrone. La combinaison PN-DFG (cf. §2.2.1) permet également de séparer le *front-end* du *back-end* de l'outil, ce qui rend indépendantes chacune de ces deux parties. L'une des conséquences immédiates de cette séparation *front-end* / *back-end* est la possibilité qui est alors donnée aux concepteurs de l'outil de synthèse d'enrichir le *front-end* avec un autre outil de compilation pour parser un autre langage de description haut niveau, ou encore de générer un autre style d'implémentation de circuits asynchrone dans le *back-end.*

On s'assure ensuite que le réseau de Petri est synthétisable et qu'une implémenation micropipeline existe en vérifiant qu'il est compatible à la spécification DTL. A ce niveau on remarque que le PN le plus complexe est celui qui traduit un circuit de sélection. C'est pourquoi nous le simplifions par fractionnement (« *unfolding* ») en plusieurs sous-ensembles de PN chacun représentant un choix déterminé de la structure de sélection. Cette opération simplifie le travail de synthèse en aval en restreignant les structures de pipeline non linéaires vu dans le chapitre 3 (cf. §3.3.2) aux cas de fourche et de convergence (divergence et convergence en « ET »). Les cas de multiplexage et de démultiplexage sont traités par regroupement des différents sous-réseaux de Petri obtenus par « *unfolding* ».

Les équations de dépendances correspondantes aux différentes structures de circuits asynchrones linéaires et non linéaires sont déclinées à partir des sous réseaux de Petri simplifiés. Ces équations, générées sans aucune hypothèse sur le protocole de communication ou sur une quelconque technologie cible, expriment les relations de causalité entre d'une part les signaux de sortie et les signaux d'entrées de contrôle et d'autre part entre les bus de sorties et les bus d'entrées de données.

Dans le chapitre cinq nous traitons la suite du processus de synthèse qui sépare le contrôleur du circuit asynchrone à synthétiser et son chemin de données à partir de ces équations de dépendances. Le chemin de données est décrit sous forme d'un code VHDL synthétisable que l'on soumet à une synthèse synchrone. Cette dernière nous fournit des informations de délais que nous exploitons dans la synthèse du contrôleur par rétro-annotation. Pour la synthèse du circuit de contrôle nous générons les équations de dépendances adaptées selon le protocole de communication quatre phases choisi, puis le contrôleur est décrit sous forme d'une *netlist* de portes utilisant une bibliothèque de cellules standard (conception de type « semi-custom »). Le contrôleur est enfin soumis aux phases d'optimisation logique et de projection technologique qui sont fortement liées.

4.2 Correspondance CHP / VHDL du circuit asynchrone micropipeline

Un code CHP général se présente sous la forme d'un composant dans lequel plusieurs processus peuvent s'exécuter concurremment. On peut le représenter comme en figure 4.2.

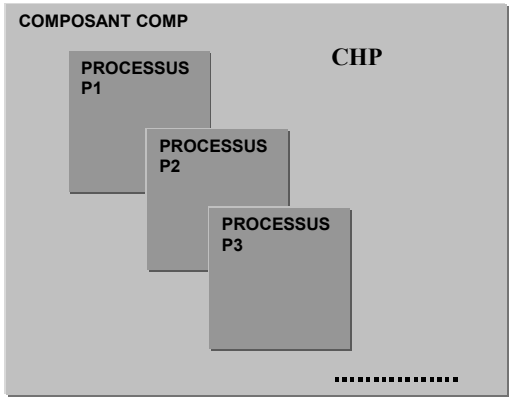


Figure 4.2 : Une représentation d'un code CHP général

En structure mémoire l'ensemble du code jusqu'au niveau des instructions est représenté sous forme d'un arbre syntaxique, et le bloc d'instruction sous forme de PN-DFG. Le chemin de données du modèle de circuit cible est décrit en VHDL synthétisable puis synthétisé par des outils commerciaux de synthèse synchrone. Pour sa part, le contrôleur est généré sous forme d'une *netlist* de portes utilisant une bibliothèque de cellules standard.

Un composant CHP contenant plusieurs processus est traduit en un composant micropipeline comprenant plusieurs sous-composants. Chacun de ses sous-composants contient deux composants, l'un représentant le chemin de données, l'autre le contrôleur du circuit asynchrone micropipeline à générer.

La figure 4.3 se propose de représenter cette correspondance entre le code CHP initial et la description VHDL structurel de notre modèle de circuit cible.

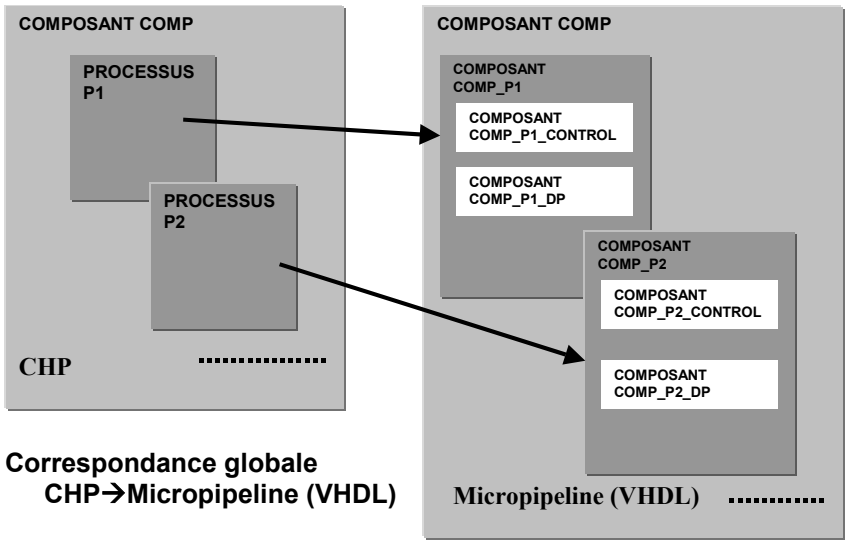


Figure 4.3 : Correspondance globale CHP→Micropipeline (VHDL)

La correspondance entre le code CHP exprimé dans l'arbre syntaxique (*i.e : jusqu'au niveau précédent l'entrée dans la partie instruction de la grammaire CHP)* et le code VHDL descriptif du circuit micropipeline traduit la correspondance de la figure 4.3. Les détails de cette correspondance étant donnés en annexe, nous nous contentons de souligner qu'à chaque port CHP de type MR[B][L] (excepté pour le type SR), on associe trois ports VHDL représentant respectivement la donnée, la requête et l'acquittement. Egalement, les variables CHP ne sont plus visibles en VHDL car les calculs intermédiaires se font directement sur les ports et le délai induit par ces calculs est bien considéré dans les contrôleurs.
Traitons à présent, la correspondance entre les instructions CHP exprimées par la combinaison PN-DFG, et le code VHDL décrivant le circuit à générer. Avant cela, notons qu'elles permettent de générer les instructions VHDL :

- du composant de contrôle qui décrit les requêtes de sortie et les acquittements d'entrée en plus des commandes de latchs,
- du composant de chemin de données qui décrit les sorties de données ainsi que le calcul des gardes,
- du composant VHDL rassemblant les deux composants précédents et représentant le processus en CHP,
- du composant VHDL représentant le composant en CHP.

4.3 Du CHP vers la combinaison PN-DFG

La combinaison PN-DFG permet de modéliser les circuits asynchrones à tout niveau de complexité (cf. §2.2.2). En effet, les réseaux de Petri sont souvent employés pour traduire le comportement logique des systèmes mixtes séquentiels et concurrents, alors que les graphes de flot de données sont idéaux pour exprimer les dépendances de données. Le modèle de réseau de Petri que l'on utilise dans notre synthèse est le PN interprété. C'est à dire que chaque place du réseau de Pétri est associée à une action de communication ou à un calcul (PN P-temporisé) et chaque transition est associée à une condition booléenne de franchissement (PN synchronisé).

La traduction d'un code CHP en représentation PN-DFG exige de pouvoir traduire toutes les structures du langage CHP. Le comportement du circuit exprimé par les structures de contrôle est représenté par le réseau de Petri, alors que les dépendances de données sont représentées par les DFG, lesquelles sont associés aux places (actions de communication et calcul) et aux transitions (gardes sous forme de conditions booléennes).

4.3.1 Expressions

Une expression du langage CHP est représentée par un DFG dont les nœuds sont associées aux différents opérateurs disponibles en CHP étendu (arithmétiques, logique, de communication, ou de décalage) et dont les feuilles sont associées aux canaux, constantes ou variables du code CHP. Un exemple d'une expression traduite en représentation DFG est donné en figure 4.4.

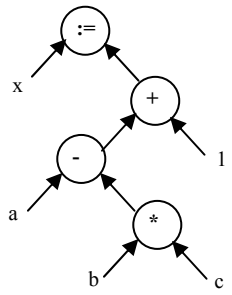


Figure 4.4 : Exemple d'un Graphe de Flot de Données (GFD ou DFG)

4.3.2 Gardes

Une garde du langage CHP est une expression booléenne. Elle est donc traitée pareillement et représentée par un DFG dont les nœuds sont particulièrement des opérateurs arithmétiques, ou de comparaison (« = », « /= », « < », …) ou de « probe ». Par ailleurs, les DFG représentant les expressions de garde sont associés aux transitions du PN.

4.3.3 Instructions

Une instruction du langage CHP est une action de communication sur les canaux (« emission », « réception », …) ou une affectation de variable locale. L'instruction « skip » ne fait rien. Le DFG représentant l'instruction est associé à une place du PN (fig.4.5).

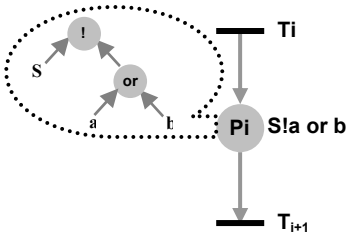


Fig. 4.5 : représentation en PN-DFG de l'instruction « S ! a or b »

4.3.4 Opérateur séquentiel « ; »

Deux instructions CHP en séquence s'expriment en représentation PN par deux places successives séparées par une transition dont la condition booléenne (garde) est toujours vraie (fig. 4.6).

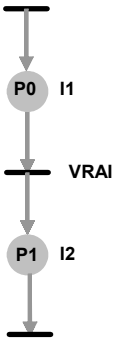


Fig. 4.6 : représentation en PN de l'opérateur séquentiel « ; »

4.3.5 Opérateur de parallélisme « , »

Deux instructions CHP concurrentes se traduisent par deux places en parallèle (fig.4.7) partant d'une première transition (divergence en « ET » éventuellement gardée par une condition autre que TRUE) et arrivant dans une seconde transition (convergence en « ET » forcément gardée par une condition toujours TRUE).

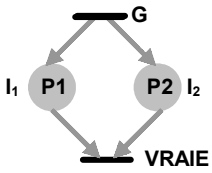


Fig. 4.7 : représentation en PN de l'opérateur parallèle « , »

4.3.6 Opérateur de sélection déterministe « @ + BREAK » ou indéterministe « @@ + BREAK »

Le PN traduisant l'opérateur de sélection (figure 4.8) se base sur les structures de divergence et de convergence en « OU ». Comme les gardes doivent être mutuellement exclusives, on prévoit le cas où deux gardes sont vraies simultanément. Dans ce cas, on arrête le déroulement du programme.

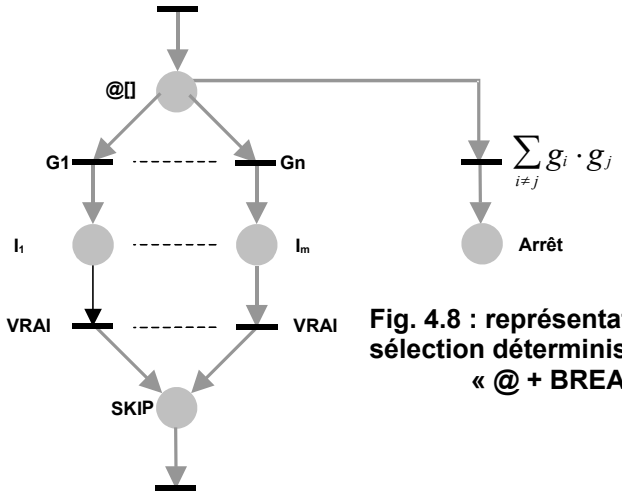


Fig. 4.8 : représentation en PN de l'opérateur de sélection déterministe :
« @ + BREAK»: @[g_1 => $inst_1$; BREAK
g_2 => $inst_2$; BREAK
… g_n => $inst_n$; BREAK]

4.3.7 Opérateur de répétition/sélection déterministe « @ + LOOP » ou indéterministe « @@ + LOOP»

Le PN traduisant l'opérateur de répétition/sélection (figure 4.9) se base également sur les structures de divergence et de convergence en « OU ». En plus du cas ou deux gardes sont vraies simultanément, on arrête également le programme dans le cas où aucune garde n'est vraie.

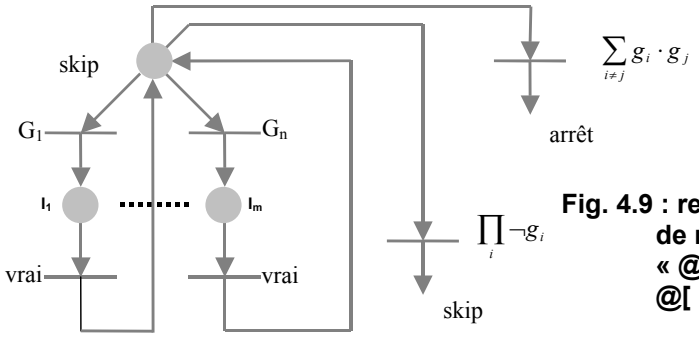


Fig. 4.9 : représentation en PN de l'opérateur de répétition/sélection déterministe
« @ + LOOP» :
@[g_1 => $inst_1$; LOOP
g_2 => $inst_2$; LOOP
… g_n => $inst_n$; LOOP
]

4.4 Vérification de la conformité du PN à la spécification DTL

Les règles DTL ont été posées pour garantir une implémentation d'un circuit asynchrone. Fondamentalement, elles restreignent les possibilités d'écriture des codes CHP (cf. §2.3.2). A ce niveau la vérification de ces règles s'effectue sur le PN. Elle consiste à vérifier :

- **l'initialisation** : Dans le cas où un canal de sortie est initialisé puis accédé séquentiellement, il s'agit de vérifier que l'initialisation se fait une et une seule fois en début d'exécution du processus ;

- **l'accès séquentiel** : Lors du parcours de chaque branche d'exécution du PN, les accès aux canaux au niveau des places sont notés. Si un canal de sortie est accédé en écriture deux fois, alors le premier accès doit correspondre à une initialisation en début de processus. Dans le cas contraire, ou encore lorsqu'un canal d'entrée est séquentiellement accédé en lecture, il y a violation des règles DTL sur les accès séquentiels et la procédure de synthèse est stoppée ;

- **l'accès parallèle** : Lors du parcours de chaque branche d'exécution du PN, les accès en écriture dans les canaux de sorties dans des branches parallèles distinctes sont notés. Si un canal de sortie est accédé concurremment en écriture, il y a violation des règles DTL sur les accès parallèles et la procédure de synthèse est stoppée ;

- **les vraies dépendances de données lors des affectations séquentielles et de la compatibilité des types de données** : Il s'agit ici de vérifier que chaque variable locale qui affecte une autre variable (x := y+z) ou qui lit une donnée sur un canal d'entrée (E?x) ou encore chaque canal de sortie qui reçoit un donnée (S!y), sont affectés avec des valeurs reçues par les canaux d'entrée.

  Si une variable locale lit deux fois successivement sur un canal d'entrée, on ne garde que la dernière des affectations. Si une variable locale est affectée par d'autres variables locales non issues de canaux d'entrée une erreur est levée. Si une variable locale non issue d'un canal d'entrée est émise dans un canal de sortie une erreur est également levée.

  En outre, la compatibilité des types de données est vérifiée (hormis pour le cas de l'affectation qui accepte la conversion de type) en comparant les bases et les longueurs des opérandes.

L'ensemble de ces vérifications assure qu'au niveau de la spécification :
- les valeurs émises sur les canaux de sorties sont des fonctions combinatoires des valeurs lues sur les canaux d'entrées ;
- le seul état de mémorisation est celui lié à l'initialisation des canaux de sorties et fait l'objet d'un traitement particulier que l'on présente dans la partie consacrée à la synthèse des chemins de données. Les états de mémoire associés au protocole de communication sont traités ultérieurement lors de la génération des équations de dépendances avec considération du protocole.

Par ailleurs, d'autres vérifications sont effectuées pour s'assurer que le PN conforme DTL est réellement synthétisable, notamment :
- déceler les segments inatteignables du PN, telle une garde dont la condition booléenne associée est toujours fausse. Le segment de réseau suivant cette garde (associée à une transition du PN) est déclaré inatteignable ;
- l'exclusion mutuelle des gardes (pour les structures de selection / répétition) ne pouvant être garantie statiquement à la compilation, est vérifiée à ce niveau en utilisant les diagrammes de décision binaire (BDD) quand cela est possible.

A l'issue de l'ensemble de ces vérifications, on est assuré de l'existence d'une implémentation micropipeline du circuit asynchrone en portes standards.

4.5 Transformation du PN par l'éclatement des structures de sélection (« @ »)

Le cas le plus général du PN représentant un circuit asynchrone est celui qui présente plusieurs branches d'exécution (ou de sélection ou de choix). Dans ce cas précis, il est fort probable qu'un même canal de sortie soit affecté dans plus d'une branche d'exécution. La fonction de sortie pour ce canal de sortie est une réunion exclusive de toutes les fonctions de sortie dans les différentes branches.

Eclater le réseau de Petri, de telle manière qu'il ne présente plus de branche de sélection, facilite les étapes ultérieures de la synthèse et limite les structures non linéaires des circuits asynchrones (cf. §3.1.3.2) aux cas de fourche et de convergence (divergence et convergence en « ET »). Les cas de multiplexage et de démultiplexage sont traités par regroupement (unions exclusives) des différentes sorties affectées dans les sous-réseaux de Petri ainsi obtenus.

La figure 4.10 illustre cette transformation qui assouplit largement la suite de la procédure de synthèse notamment sur le plan de l'implémentation pure, en facilitant grandement le parcours du PN.

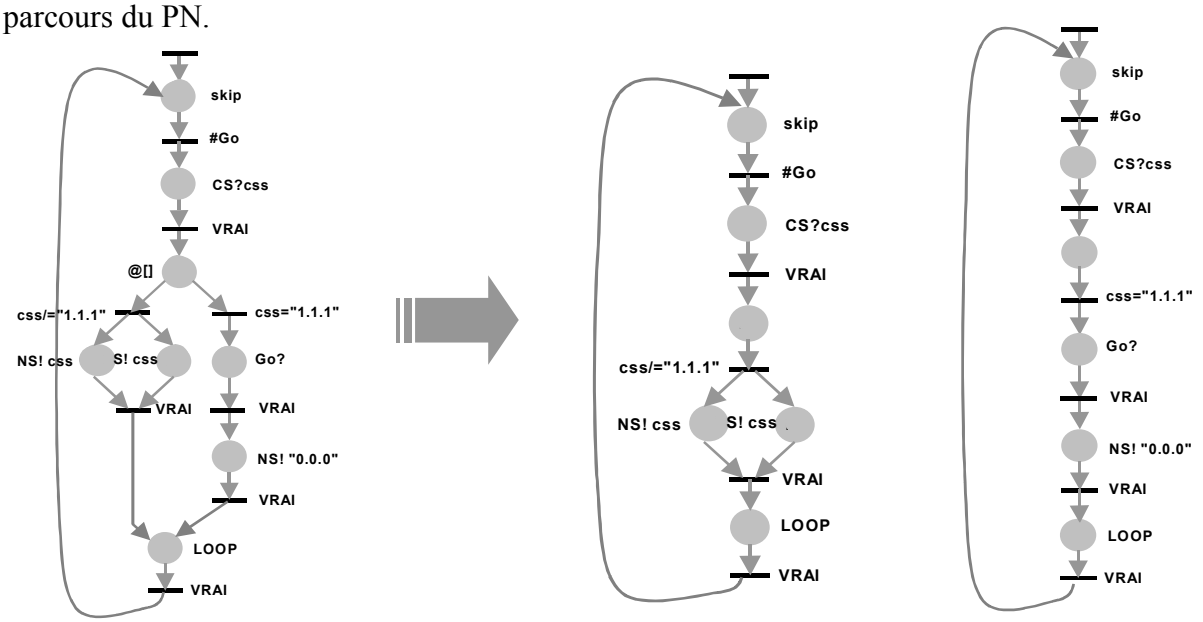


Fig. 4.10 : Eclatement d'un réseau de Petri relatif à un processus à sélection en 2 sous-réseaux de Petri exempts de structure de « choix »

4.6 Génération des équations de dépendances non contraintes par le protocole

4.6.1 Définition de l'équation de dépendances

L'équation de dépendances est une équation conventionnelle qui doit formaliser le fonctionnement d'un circuit asynchrone en vue de sa synthèse. Elle contient les informations qui font que le circuit change d'état et exprime la sémantique des canaux de communication en préservant le comportement du circuit tel que traduit par le PN. La relation de dépendance entre les sorties et les entrées du circuit sont exprimées par les équations de dépendances des sorties en fonction des entrées.

Notons que nous générons ces équations à partir des sous-réseaux de Petri ne contenant plus de structures de sélection. De la sorte l'équation de dépendances exprimant l'ensemble du circuit est obtenue par regroupement (union exclusive) après avoir décliné toutes les équations relatives aux différents sous–réseaux de Petri.

Une équation de dépendances relative à un sous-réseau de Petri (*i.e* : qui correspond à une seule et unique branche d'exécution) et exprimant la relation entre un canal de sortie et les canaux d'entrées dont elle dépend contient les informations suivantes:

- le nom du composant CHP ;
- le nom du processus traité et appartenant à ce composant ;
- un drapeau booléen indiquant si la sortie traitée est initialisée (dans ce cas préciser sa valeur d'initialisation constante) ;
- une branche d'exécution pouvant très bien contenir plusieurs gardes imbriquées, ces dernières -décrivant une seule et unique sélection- doivent être explicitées ;
- les canaux d'entrée « probés » avec 2 paramètres, l'un spécifiant s'il est consommé, et le second spécifiant si le canal est testé et comparé à une valeur ou testé simplement ;
- la validité des données sur les canaux d'entrée dont dépend la sortie (explicitement exprimée par les signaux de requête d'entrée dans le cas micropipeline) / ou les données d'entrées lorsqu'il s'agit des équations de dépendances du chemin de données ;
- la disponibilité du canal de sortie considéré (explicitement exprimée par le signal d'acquittement de sortie) ;
- les fonctions de calcul des sorties de données dépendant exclusivement des entrées de circuit (garanti par le caractère DTL du PN traduisant le code).

La figure 4.11 montre les différentes sorties inhérentes à un circuit asynchrone micropipeline.

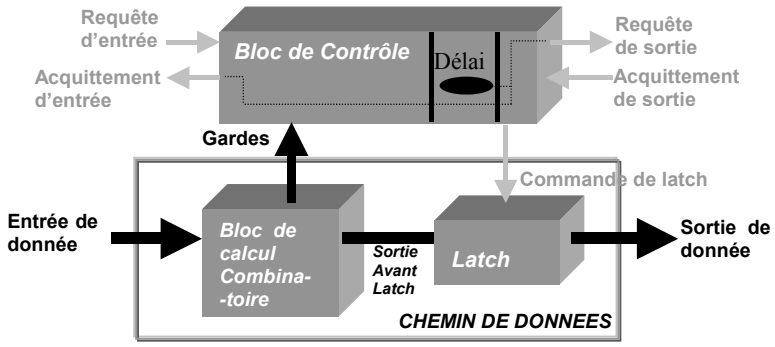


Fig. 4.11 : Schéma bloc de la structure du circuit micropipeline cible

Trois types de sorties existent pour le « contrôle » : l'acquittement d'entrée (***E_ack***), la requête de sortie (***S_req***), et la commande de latch pour une sortie donnée (***cmd_latch_S***). Les chemins de données génèrent deux types de sorties : les sorties de données (***S_data***) et les calculs de garde (***G***) à transmettre au bloc de contrôle.

Globalement pour une sélection (G_i) dans un processus déterminé appartenant à un composant défini, l'équation de dépendances générale pour une requête de sortie (contrôle) ou une sortie de données est donnée en « Equation 4.1 » :

$$S_k^{Gi-req/données} = F_{dep}^{Gi-req/données}(nom\_composant,\ nom\_processus,\ (init{=}0/1, cst),\ \prod_j E_{j-probés}^{Gi-req/données}(consom{=}0/1, valeur),\ \prod_j E_j^{Gi-req/données},\ (G_i{=}F_{garde}^{Gi}(\prod_j E_j^{Gi-données})),\ \prod_k S_k^{Gi-ack}, F_{Sk}^{Gi}(\prod_j E_j^{Gi-données})\ )$$

Equation 4.1

Cette équation concerne la dépendance des canaux de sorties S_k (sortie du chemin de donnée ou requête de sortie du contrôleur) relatifs à la garde (choix) G_i.
Les notations utilisées dans l'écriture de cette équation de dépendances signifient :

$S_k^{Gi-req/données}$: requête de sortie « Sk_req » / sortie de données « Sk_data » relative à la garde Gi,

$F_{dep}^{Gi-req/données}$: Fonction de causalité entre les requêtes de sortie et les requêtes d'entrées et acquittement de sortie ou entre les sorties de données et les entrées de données (*Pour la cas des requêtes de sortie, on considère dans cette fonction les retards dus au calcul des sorties de données dans le bloc combinatoire et au temps de propagation dans les latchs*)

$(init=0/1, cst)$: Si une ou plusieurs sorties du circuit asynchrone est/sont initialisées alors le drapeau « init » est à 1. Dans ce cas, « cst » mentionne la valeur de l'initialisation. Autrement, « init » est à 0.

$\prod_j E_{j-probés}^{Gi-req/données}(consom=0/1, valeur)$: Si le code décrivant le circuit à générer contient des sondes de canal (probes) alors ce paramètre mentionne les canaux « probés ». L'argument « consom » indique si ce canal est consommé dans la garde « Gi », alors que l'argument « valeur » est donné lorsque le canal est testé et comparé à une valeur fixe.

$\prod_j E_j^{Gi-req/données}$: L'ensemble des requêtes des canaux d'entrées, ou l'ensemble des entrées de données relatives à la garde Gi,

$G_i = F_{garde}^{Gi}(\prod_j E_j^{Gi-données})$: Fonction de la garde Gi en fonction des entrées de données,

$\prod_k S_k^{Gi-ack}$: L'ensemble des acquittements de sorties dans la garde Gi,

$F_{Sk}^{Gi}(\prod_j E_j^{Gi-données})$: Valeur de la sortie de donnée Sk de la garde Gi en fonction des entrées de données de cette même garde.

L'équation 4.1 signifie alors que si les conditions de la garde G_i sont satisfaites, que les données sont prêtes sur les canaux d'entrée E_j et que le canal de sortie Sk est disponible (prêt à recevoir des données), alors, après un retard comprenant le temps nécessaire au calcul des sorties de données et au temps de transfert dans les latchs, une requête de sortie Sk_req est émise sur la sortie de contrôle. La sortie de donnée est calculée par la fonction Fs_k et émise sur la sortie Sk.

Dans les trois protocoles 4 phases WCHB, PCHB, et PCFB le signal de commande du latch relatif à une sortie donnée est équivalent au signal de requête de cette même sortie, mais est en avance sur celui-ci d'un délai équivalent au temps de propagation dans le latch. En d'autre termes, l'équation qui gère la génération du signal de commande de latch est identique à l'équation 4.1 mais sans le délai des latch.

L'équation générale pour l'acquittement d'entrée est donnée en « Equation 4.2 » :

$$E_j^{Gi-acq} = F_{dep}^{Gi-acq}(nom\_composant,\ nom\_processus,\ (init=0/1, cst),\ \prod_j E_{j-probés}^{Gi-req}(consom=0/1, valeur),\ \prod_j E_j^{Gi-req},\ (G_i),\ \prod_k S_k^{Gi-ack})$$

Equation 4.2

Les notations utilisées dans cette équation sont identiques à ceux de l'équation 4.1 avec en plus :

E_j^{Gi-acq} : L'acquittement de l'entrée « Ej » relatif à la garde Gi,

F_{dep}^{Gi-acq} : Fonction de dépendance de l'acquittement, exprimant la causalité entre l'acquittement d'entrée d'une part, et les requêtes d'entrée et acquittement de sortie d'autre part.

Ces équations de dépendances reliant les canaux de sorties des circuits asynchrones à leurs canaux d'entrées ne font aucune hypothèse sur le modèle de circuit, le protocole de communication ou la technologie cible.

4.6.2 Equations de dépendances (E.D.) des structures de base (pipelines linéaires et non linéaires) du PN

Les structures non linéaires ne concernent pas les structures de sélection puisque les équations de dépendances sont générées à partir d'un réseau de Petri vérifié et simplifié (éclatement des choix). Nous présentons les équations de dépendances des structures de pipeline linéaire, de fourches et de convergences avant de proposer les équations de dépendances pour un circuit asynchrone quelconque.

4.6.3 E.D. de la structure de pipeline linéaire « place-transition-place »

Dans une branche d'instruction on peut rencontrer une structure séquentielle dont le code CHP général (un code plus particulier considère la garde « G » toujours vraie) est :

```
I1 ; [G => I2]
```

où « I1 » et « I2 » sont des instructions du langage CHP (ex : émission ou réception) et « G » une condition booléenne de garde. Ce type d'écriture signifie que l'instruction « I2 » est exécutée après que l'instruction « I2 » l'ait été et que la condition de garde « G » ait été satisfaite.

Le reseau de Petri traduisant cette structure est donné en figure 4.12. Aux deux places P0 et P1 sont associées les deux instructions « I1 » et « I2 », et à la transition « T0 » est associée la garde « G ».

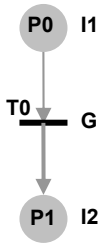


Fig. 4.12 : Réseau de Petri de la structure séquentielle « I1 ; [G => I2] »

Les équations de dépendances générées pour cette structure sont données en équations 4.3 et 4.4.

$$S_{I_2}^{G\ -req/données} = F_{dep}^{G-req/données}(nom\_composant, nom\_processus,$$
$$(init=0),$$
$$\prod_j E_{j-I_1}^{G-req/données},\ (G=F_{garde}^{G}(\prod_j E_{j-I_1}^{G-données})),$$
$$\prod_k S_{k-I_2}^{G-ack}, F_{S_{k-I2}}^{G}(\prod_j E_{j-I_1}^{G-données})\ )$$

Equation 4.3

$$E_{j-I_1}^{G-acq} = F_{dep}^{G-acq}(nom\_composant, nom\_processus,$$
$$(init=0),$$
$$\prod_j E_{j-I_1}^{G-req},\ (G),$$
$$\prod_k S_{k-I_2}^{G-ack}\ )$$

Equation 4.4

Exemple : **[E?x ; S!x ; LOOP]**

Dans cet exemple la garde « G » est toujours vraie, l'instruction « I1 » est une lecture dans le canal d'entrée « E », et l'instruction « I2 » une écriture dans le canal de sortie « S ». Les équations de dépendances générées pour cet exemple sont données en équations 4.5 et 4.6.

$$S^{req/données} = F_{dep}^{req/données}(nom\_composant, nom\_processus,$$
$$(init=0),$$
$$E^{req/données},$$
$$S^{ack}, F_S=E^{données})\ )$$

Equation 4.5

$$E^{acq} = F_{dep}^{acq}(nom\_composant, nom\_processus,$$
$$(init=0),$$
$$E^{req},$$
$$S^{ack}\ )$$

Equation 4.6

Notons que la garde « G » étant toujours vraie dans cet exemple, elle n'apparaît plus dans les équations de dépendances.

4.6.4 E.D. de la structure de divergence en « ET »

Dans une branche d'instruction on peut rencontrer une structure de divergence en « ET » dont le code CHP général (un code plus particulier considère la garde « G » toujours vraie) est :

I1 ; [G => I2 , I3]

où les instructions « I2 » et « I3 » sont exécutées concurrement après que l'instruction « I1 » ait été exécutée et que la condition de garde « G » ait été satisfaite. Le réseau de Petri traduisant cette structure est donné en figure 4.13.

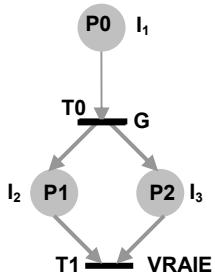


Fig. 4.13 : Réseau de Petri de la structure divergence en "ET" : « I1 ; [G => I2 , I3] »

Les équations de dépendances générées pour cette structure sont données en équations 4.7, 4.8 (sortie de donnée et requête de sortie pour I2 et I3) et 4.9 (acquittement d'entrée pour I1).

Equation 4.7

$$S_{I2}^{G-req/données} = F_{dep}^{G-req/données}(nom\_composant, nom\_processus, (init=0), \prod_j E_{j-I1}^{G-req/données}, (G=F_{garde}^{G}(\prod_j E_{j-I1}^{G-données})), \prod_k S_{k-I2-I3}^{G-ack}, F_{Sk-I2}^{G}(\prod_j E_{j-I1}^{G-données}))$$

Equation 4.8

$$S_{I3}^{G-req/données} = F_{dep}^{G-req/données}(nom\_composant, nom\_processus, (init=0), \prod_j E_{j-I1}^{G-req/données}, (G=F_{garde}^{G}(\prod_j E_{j-I1}^{G-données})), \prod_k S_{k-I2-I3}^{G-ack}, F_{Sk-I3}^{G}(\prod_j E_{j-I1}^{G-données}))$$

Equation 4.9

$$E_{j-I1}^{G-acq} = F_{dep}^{G-acq}(nom\_composant, nom\_processus, (init=0), \prod_j E_{j-I1}^{G-req}, (G), \prod_k S_{k-I2-I3}^{Gi-ack})$$

Nous notons que les deux sorties de requête $S_{I1\text{-}req}$ que $S_{I2\text{-}req}$ ainsi que l'acquittement d'entrée $E_{j\text{-}ack}$ s'activent notamment en attendant que les deux étages qui suivent et qui sont associés à I2 et I3 soient tous deux acquittés. C'est la synchronisation des signaux d'acquittement d'une fourche (cf. §3.1.3.3.1).

Exemple : **[E?x ; [x=0 => S1!x , S2!x ; BREAK] ; LOOP]**
Les équations de dépendances de cet exemple qui représente une fourche à 2 sorties gardée par une condition booléenne imposant que E valle 0, sont données en équations 4.10, 4.11 et 4.12.

$$S1^{G-req/données} = F_{dep}^{G-req/données}(nom\_composant, nom\_processus, (init=0), E^{G-req/données}, (G=(E=0)) S1^{G-ack}, S2^{G-ack}, F_{S1}^{G} = E^{G-données})$$

Equation 4.10

$$S2^{G-req/données} = F_{dep}^{G-req/données}(nom\_composant, nom\_processus, (init=0), E^{G-req/données}, G S1^{G-ack}, S2^{G-ack}, F_{S2}^{G} = E^{G-données})$$

Equation 4.11

$$E^{G-acq} = F_{dep}^{G-acq}(nom\_composant, nom\_processus, (init=0), E^{G-req}, G S1^{G-ack}, S2^{G-ack})$$

Equation 4.12

4.6.5 E.D. de la structure de convergence en « ET »

Dans une branche d'instruction on peut rencontrer une structure de convergence en « ET » dont le code CHP général (un code plus particulier considère la garde « G » toujours vraie) est:

I1 , I2; [G => I3]

où l'instruction « I3 » est exécutée après que les instructions « I1 » et « I2 » aient été exécutées et que la condition de garde « G » ait été satisfaite. Le réseau de Petri traduisant cette structure est donné en figure 4.14.

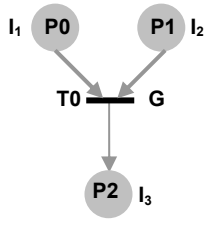


Fig. 4.14 : Réseau de Petri de la structure convergence en "ET" : « I1 ; I2 ; [G => I3] »

Les équations de dépendances générées pour cette structure sont données en équations 4.13, (sortie de donnée et requête de sortie pour I2 et I3) 4.14 et 4.15 (acquittement d'entrée pour I1 et I2).

$$S_{k-I_3}^{G-req/données} = F_{dep}^{G-req/données}(nom\_composant, nom\_processus, (init=0), E_{j-I_1}^{G-req/données}, E_{j-I_2}^{G-req/données}, (G=F_{garde}^{G}(\prod_j E_{j-I_1-I_2}^{G-données})), S_{k-I_3}^{G-ack}, F_{S_{k-I_3}}^{G}(\prod_j E_{j-I_1}^{G-données}, \prod_j E_{j-I_2}^{G-données}))$$

Equation 4.13

$$E_{j-I_1}^{G-acq} = F_{dep}^{G-acq}(nom\_composant, nom\_processus, (init=0), \prod_j E_{j-I_1}^{G-req}, \prod_j E_{j-I_2}^{G-req}, (G), \prod_k S_{k-I_3}^{G-ack})$$

Equation 4.14

$$E_{j-I_2}^{G-acq} = F_{dep}^{G-acq}(nom\_composant, nom\_processus, (init=0), \prod_j E_{j-I_1}^{G-req}, \prod_j E_{j-I_2}^{G-req}, (G), \prod_k S_{k-I_3}^{G-ack})$$

Equation 4.15

Notons que les équations de dépendances 4.14 et 4.15 correspondants aux deux acquittements d'entrées $E_{I1\text{-}ack}$ et $E_{I2\text{-}ack}$ sont identiques car les entrées de convergence sont acquittées simultanément. La synchronisation des entrées de convergence (cf. §3.1.3.3.2) est respectée car lorsque les données arrivent sur une entrée de convergence, on attend les autres données avant de procéder à un quelconque calcul ou d'envoyer une quelconque donnée en sortie de circuit.

Exemple : **[E1?x , E2?y ; S!x+y ; LOOP]**
Les équations de dépendances de cet exemple qui représente une convergence à 2 entrées sont données en équations 4.16, 4.17 et 4.18.

$$S^{req/données} = F_{dep}^{req/données}(nom\_composant, nom\_processus, (init=0), E1^{req/données}, E2^{req/données}, S^{ack}, F_S = E1^{données} + E2^{données}))$$

Equation 4.16

$$E1^{acq} = F_{dep}^{acq}(nom\_composant, nom\_processus, (init=0), E1^{req}, E2^{req}, S^{ack})$$

Equation 4.17

$$E2^{acq} = F_{dep}^{acq}(nom\_composant, nom\_processus, (init=0), E1^{req}, E2^{req}, S^{ack})$$

Equation 4.18

4.6.6 Génération des équations de dépendances d'un circuit asynchrone (ou PN) compatible DTL

Au cours des discussions qui précèdent, nous avons procédé à la définition d'un cadre d'écriture du code CHP avec lequel nous pouvons affirmer qu'il existe de façon sûre une implémentation asynchrone réalisant la fonctionnalité décrite par le code en question. Ce cadre correspond aux règles DTL que la spécification d'entrée doit satisfaire. Elles stipulent fondamentalement que les données des canaux de sortie du circuit sont calculées exclusivement à partir des données courantes des canaux d'entrées, et non pas en fonction des valeurs passées de ces mêmes canaux d'entrées.

Autrement dit, les circuits asynchrones conformes à la spécification DTL (d'ailleurs traduits en réseaux de Petri eux-mêmes compatibles DTL) répondent à une activité du type « consommation-production ». En terme de code CHP, cette activité s'exprime par des lectures de canaux d'entrée en parallèle (consommation concurrente des canaux d'entrées) suivie par des écritures concurrentes sur des canaux de sortie (production parallèle des canaux de sorties), comme suit :

$$\prod_j E_j? \; ; \; \prod_k S_k!$$

Notons que les variables locales sont considérées comme des containers de données servant d'intermédiaires entre les canaux d'entrées et les canaux de sorties (calculs intermédiaires, modification de la taille des données, …), et grâce à la spécification DTL (cf. §4.4), elles n'apparaissent absolument pas dans les équations de dépendances. On leur substitue les canaux d'entrées par lesquels elles sont affectées.

Maintenant, nous avons vu également (cf. §3.1.3) que parmi les différentes structures de pipeline linéaire et non-linéaires, les structures les plus étoffées sont celles du multiplexeur ou du démultiplexeur que nous rassemblons sous le vocable de structures de sélection.

En §4.4, nous avons montré comment ces structures étaient éclatées dans les réseaux de Petri, précisément pour simplifier la procédure de synthèse. De la sorte, une structure de sélection se trouve réduite en plusieurs sous-structures qui peuvent être linéaires ou non-linéaires. Dans ce dernier cas, elles ne peuvent traduire, au pire, que des fourches ou des convergences. Une représentation en code CHP d'une branche d'exécution générale (plusieurs gardes séquentielles imbriquées) s'exprime comme suit :

```
[ Eh?varh ;
        @[ G0 =>  Ej?varj ;
                @[ G00 =>  Em?varm ;
                        @[ G000 =>  En?varn ; Sk? FSk ( varh , varj , varm , varn ) ; BREAK
```

Cette écriture traduit une seule branche d'exécution car les gardes « G_0 », « G_{00} », et « G_{000} » se renforcent mutuellement pour ne constituer qu'une seule garde équivalente au résultat de l'opération « et logique » de l'ensemble des gardes « $G_0 . G_{00} . G_{000}$ ». Par ailleurs, on voit bien dans cette écriture que l'on consomme d'abord tous les canaux d'entrées avant de procéder à la production des canaux de sorties.
Les sous-structures de pipeline asynchrone ainsi définies sont exprimées par des sous-réseaux de Petri (exempts de la structure de choix) lesquels génèrent des sous-équations de dépendances. Chacune de ces sous-équations représente une branche d'exécution du PN, donc un choix particulier et unique (l'exclusivité mutuelle des gardes est vérifiée, cf. §4.3). Ces sous-équations sont reprises pour rappel en équations « 4.19 » et « 4.20 ».

Equation 4.19

$$S_k^{Gi-req/données} = F_{dep}^{Gi-req/données}(nom\_composant, nom\_processus, (init{=}0/1, cst), \prod_j E_{j-probés}^{Gi-req/données}(consom{=}0/1, valeur), \prod_j E_j^{Gi-req/données}, (G_i{=}F_{garde}^{Gi}(\prod_j E_j^{Gi-données})), \prod_k S_k^{Gi-ack}, F_{Sk}^{Gi}(\prod_j E_j^{Gi-données}))$$

Equation 4.20

$$E_j^{Gi-acq} = F_{dep}^{Gi-acq}(nom\_composant, nom\_processus, (init{=}0/1, cst), \prod_j E_{j-probés}^{Gi-req}(consom{=}0/1, valeur), \prod_j E_j^{Gi-req}, (G_i), \prod_k S_k^{Gi-ack})$$

Puis des opérations d'union exclusive (Equations 4.21 et 4.22) regroupent ces sous-équations pour générer les équations de dépendances finales non contraintes (par un protocole ou une technologie cible).

Equation 4.21

$$S_k^{req/données} = F_{Union}^{req/données}(\prod_k S_k^{Gi-req/données})$$

Equation 4.22

$$E_j^{acq} = F_{union}^{acq}(\prod_j E_j^{Gi-acq})$$

D'un point de vue algorithmique ces équations de dépendances sont générées comme suit :

```
Faire parcours de l'arbre syntaxique
  Pour chaque composant Faire
    Enregistrer le nom du composant
    Pour chaque processus Faire
      Enregistrer le nom du processus
      Marquer le drapeau « Init » pour dire si circuit initialisé(cf.4.5.1)
      Pour chaque sous-réseau de Petri
        Faire parcours en respectant la sémantique (notamment
        fourche et convergence)
          ▪ Enregistrer les gardes imbriquées avec leurs calculs
          ▪ Enregistrer les canaux d'entrées
          ▪ Enregistrer les canaux « probés »
          ▪ Marquer les drapeaux « consom » des canaux « probés » et
          « valeur » si le canaux « probé » est testé et comparé
          ▪ Enregistrer les valeurs des variables locales (containers)
          ▪ Enregistrer les canaux de sorties
        Fin parcours
        ▪ Enregistrer la fonction de chaque canal de sortie du sous-réseau de
        Petri en fonctions des canaux d'entrées grâce aux valeurs des
        variables locales
        ▪ Former les sous-équations de dépendance des commandes de latch
        ▪ Former les sous-équations de dépendance des requêtes de sorties
        ▪ Former les sous-équations de dépendance des sorties de données
        ▪ Former les sous-équations de dépendance des acquittements d'entrées
      Fin Pour
    Par union des sous-équations de dépendance:
    ▪ Former les équations de dépendance des commandes de latchs
    ▪ Former les équations de dépendance des requêtes de sorties
    ▪ Former les équations de dépendance des sorties de données
    ▪ Former les équations de dépendance des acquittements d'entrées
    Fin Pour
  Fin Pour
Fin Faire
```

4.6.7 Exemple d'un compteur

Le code CHP du compteur que l'on prend en exemple pour décliner les équations de dépendances non contraintes est donné ci-dessous.

```
-------------------------
component counter
-------------------------
component counter
port ( Go : in BD passive bit;
       S : out BD active bit[3])
channel CS : BD bit[3];
channel NS : BD bit[3];
begin
-------------------------
1er processus
-------------------------
process MAIN
port ( Go : in BD passive bit;
       CS : in BD active bit[3];
       NS : out BD active bit[3];
       S : out BD active bit[3])
variable css  :  bit[3];
[ [Go# =>
     [CS?css;
        [   css /= "1.1.1"[2] => NS!css , S!css
           @css = "1.1.1"[2]  => Go? ; NS!"0.0.0"[
        ]; BREAK
     ]; LOOP
   ]
]
-------------------------
2nd processus
-------------------------
process INCREMENT
port (  CS : out BD passive bit[3];
        NS : in BD passive bit[3])
variable x  :  bit[3];

[CS!"0.0.1"[2];
  [ NS?x ; CS!(x + 1)] ; LOOP
]
-------------------------
end counter;
-------------------------
```

Ce code CHP a été rendu compatible DTL grâce à l'extraction de la variable de mémorisation par la méthode des registres circulaires (cf. §2.3.3.1). De ce fait, il est décrit par deux processus, l'un pour l'incrémentation l'autre pour copier et transférer l'entrée sur la sortie du compteur et pour réinitialiser celui-ci lorsque la valeur 7 est atteinte.

On montre dans la figure 4.15 l'éclatement du sous-réseau de Petri décrivant le processus «*main*» du composant «*counter*» à synthétiser, en 2 réseaux de Petri exempts de la structure de sélection.

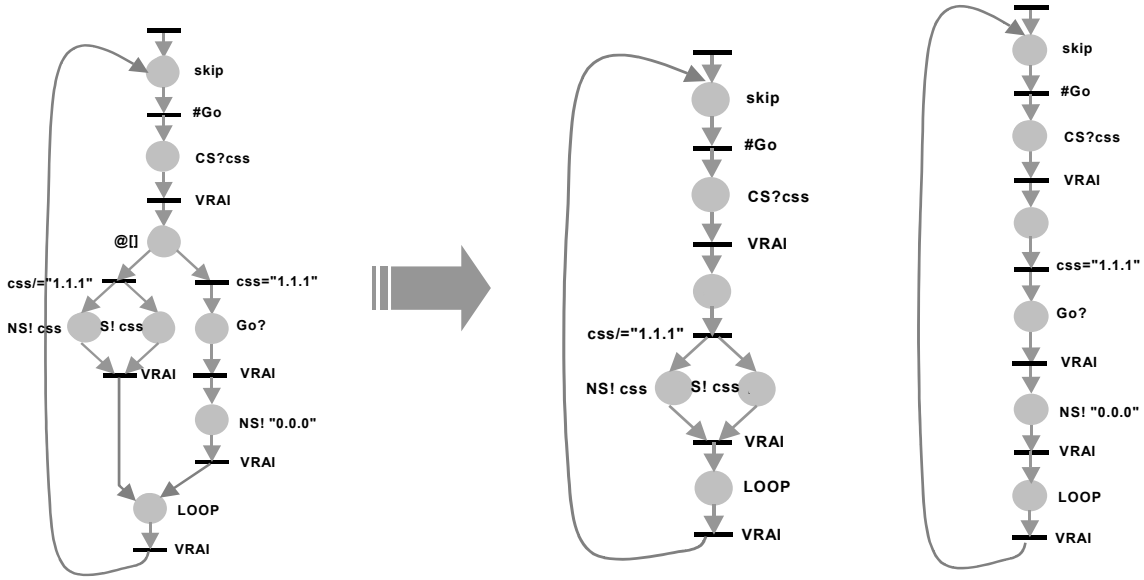


Fig. 4.15 : Eclatement du réseau de Petri du processus « MAIN » en 2 sous-réseaux de Petri exempt de structure de « choix »

Le processus « main » est un processus à sélection avec sonde de canal. Le drapeau « init » est par conséquent à « 0 » tandis que le canal de sonde « Go » apparaît dans l'équation de dépendances 4.23 relative à la sortie « NS » dans la première garde « G0 » :

$$NS^{G0-req/données} = F_{dep}^{G0-req/données}(COUNTER, MAIN, (init=0), Go_{probés}^{G0-req/données}(consom=0), CS^{G0-req/données}, (G_0=(CS/="1.1.1"[2])), NS^{G0-ack}, S^{G0-ack}, (NS^{G0-données}=CS^{données}))$$

Equation 4.23

La sortie « S » étant affectée concurremment à la sortie « NS » dans la même garde « G0 », sa valeur est identique :

$$S^{G0-req/données} = F_{dep}^{G0-req/données}(COUNTER, MAIN, (init=0), Go_{probés}^{G0-req/données}(consom=0), CS^{req/données}, (G_0=(CS/="1.1.1"[2])), NS^{ack}, S^{ack}, (S^{G0-données}=CS^{données}))$$

Equation 4.24

L'acquittement de l'entrée CS considère également le canal de sonde, tout en mentionnant que ce canal n'est pas consommé dans cette garde (« G0 ») :

Equation 4.25

$$CS^{G0-acq} = F_{dep}^{G0-acq}(COUNTER, MAIN, (init=0), Go_{probés}^{G0-req/données}(consom=0), CS^{req/données}, (G0), NS^{ack}, S^{ack})$$

Dans la seconde garde « G1 », les choses sont sensiblement les mêmes à ceci près que le canal de sonde « Go » est dans ce cas consommé, ce qui rend obligatoire son acquittement. Par ailleurs, la sortie de données « NS » est affectée par une valeur distincte, alors que la sortie « S » n'apparaît pas du tout. Bien entendu, la valeur de la garde est différente et exclusive par rapport à « G0 ».

Equation 4.26

$$NS^{G1-req/données} = F_{dep}^{G1-req/données}(COUNTER, MAIN, (init=0), Go_{probés}^{G1-req/données}(consom=1), CS^{G1-req/données}, (G1=(CS="1.1.1"[2])), NS^{G1-ack}, (NS^{G1-données}="0.0.0"[1])$$

Equation 4.27

$$CS^{G1-acq} = F_{dep}^{G1-acq}(COUNTER, MAIN, (init=0), Go_{probés}^{G1-req/données}(consom=1), CS^{req/données}, (G1), NS^{ack})$$

Equation 4.28

$$Go^{G1-acq} = F_{dep}^{G1-acq}(COUNTER, MAIN, (init=0), Go_{probés}^{G1-req/données}(consom=1), CS^{req/données}, (G1), NS^{ack})$$

L'union exclusive des sous-équations de dépendances donne les équations de dépendances totales pour le processus « main ». Les ports de sortie « NS » et d'entrée « CS » apparaissent dans les deux gardes « G0 » et « G1 », alors que « S » n'est affecté qu'en « G0 » et le canal de sonde « Go » n'est acquitté qu'en « G1 » où il est consommé.

Equation 4.29

$$NS^{req/données} = F_{Union}^{req/données}(NS^{G0-req/données}, NS^{G1-req/données})$$

Equation 4.30

$$S^{req/données} = S^{G0-req/données}$$

Equation 4.31

$$CS^{acq} = F_{union}^{acq}(CS^{G0-acq}, CS^{G1-acq})$$

Equation 4.32

$$Go^{acq} = Go^{G1-acq}$$

Le processus « increment » se distingue du processus « main » notamment par le fait qu'il soit « initialisant ». En effet, la sortie « CS » est d'abord initialisée à la valeur décimale « 1 » avant que le processus n'entame sa boucle d'incrémentation. De ce fait, l'argument « init » est mis à 1. L'impact de cette initialisation sur le plan de l'implémentation est traité dans la synthèse des contrôleurs dans le prochain chapitre.

Equation 4.33

$$\begin{aligned} CS^{req/données} = F_{dep}^{req/données}(&COUNTER, INCREMENT,\\ &(init=1, CS_{init}="0.0.1"[2]),\\ &NS^{req/données},\\ &CS^{ack}, (CS^{données}=NS^{données}+"0.0.1"[2])\ ) \end{aligned}$$

Equation 4.34

$$\begin{aligned} NS^{acq} = F_{dep}^{acq}(&COUNTER, INCREMENT,\\ &(init=1, CS="0.0.1"[2])\\ &NS^{req},\\ &CS^{ack}\ ) \end{aligned}$$

4.7 Conclusion

Dans ce chapitre nous avons présenté le mécanisme de passage de la spécification CHP à la représentation combinée PN-DFG, ainsi que l'algorithme de vérification DTL qui s'assure que le code d'entrée est compatible aux règles DTL. Nous avons ensuite montré comment un réseau de Petri est transformé un ensemble de sous-réseaux en vue de simplifier la procédure de synthèse.

La génération des équations de dépendances qui formalisent le fonctionnement d'un circuit asynchrone en vue de sa synthèse est définie et explicitée. Ces équations définissent les sorties du circuit asynchrone en fonction exclusive de ses entrées et ne considèrent pas, pour l'instant, les paramètres de protocole de communication ou de technologie cible.

CHAPITRE V

5. Synthèse de circuits asynchrones micropipeline

Les équations de dépendances définies dans le chapitre précédent constituent une forme de représentation des circuits asynchrones indépendante du protocole de communication et de la technologie cible. Ainsi, elles peuvent être considérées comme un point de départ pour la suite du processus de synthèse qui consiste à synthétiser séparément le chemin de donnée par un outil synchrone, alors que le contrôleur doit être complètement décrit jusqu'au niveau portes, optimisé, puis « projeté » sur une technologie.

Comme nous l'avons vu, les équations de dépendances explicitent la relation de causalité des requêtes de sorties, des acquittements d'entrée, ainsi que des sorties de données en fonction exclusives des entrées du circuit. Nous avons également écrit les équations relatives à un circuit asynchrone quelconque au sens où il est défini dans §4.6.6, en l'occurrence :

Equation 5.1

$$S_k^{Gi-req/données} = F_{dep}^{Gi-req/données}(nom\_composant, nom\_processus, (init{=}0/1, cst), \prod_j E_{j-probés}^{Gi-req/données}(consom{=}0/1, valeur), \prod_j E_j^{Gi-req/données}, (G_i{=}F_{garde}^{Gi}(\prod_j E_j^{Gi-données})), \prod_k S_k^{Gi-ack}, F_{Sk}^{Gi}(\prod_j E_j^{Gi-données}))$$

Equation 5.2

$$E_j^{Gi-acq} = F_{dep}^{Gi-acq}(nom\_composant, nom\_processus, (init{=}0/1, cst), \prod_j E_{j-probés}^{Gi-req}(consom{=}0/1, valeur), \prod_j E_j^{Gi-req}, (G_i), \prod_k S_k^{Gi-ack})$$

Equation 5.3

$$S_k^{req/données} = F_{Union}^{req/données}(\prod_k S_k^{Gi-req/données})$$

Equation 5.4

$$E_j^{acq} = F_{union}^{acq}(\prod_j E_j^{Gi-acq})$$

Ainsi le flot de synthèse à partir des équations de dépendances peut être représenté comme en figure 5.1. Dans ce chapitre nous distinguons la synthèse des chemins de donnée de celle des contrôleurs en réécrivant les équations de dépendances pour chacun des deux cas. Le chemin de données est décrit sous forme d'un code VHDL synthétisable qui est soumis à un outil de synthèse synchrone. Nous extrayons au travers de cette synthèse les valeurs de délais de chaque branche d'exécution pour les fournir au circuit de contrôle qui doit attendre la fin de calcul avant d'émettre ses signaux vers les étages suivants du pipeline asynchrone. La seconde partie de ce chapitre est consacrée à la synthèse des contrôleurs. Nous avons présenté au cours du chapitre trois, traitant des modèles de circuits cibles, les diverses structures de pipelines linéaires et non linéaires en montrant notamment leur implémentation selon le protocole de communication choisi. Nous développons plus en détails la génération de ces circuits pour converger vers des algorithmes de génération automatique de contrôleurs asynchrones. Les phases d'optimisation et de projection technologique sur une bibliothèque

de cellules standard se font conjointement avec le souci permanent de préserver le circuit de tout aléa.

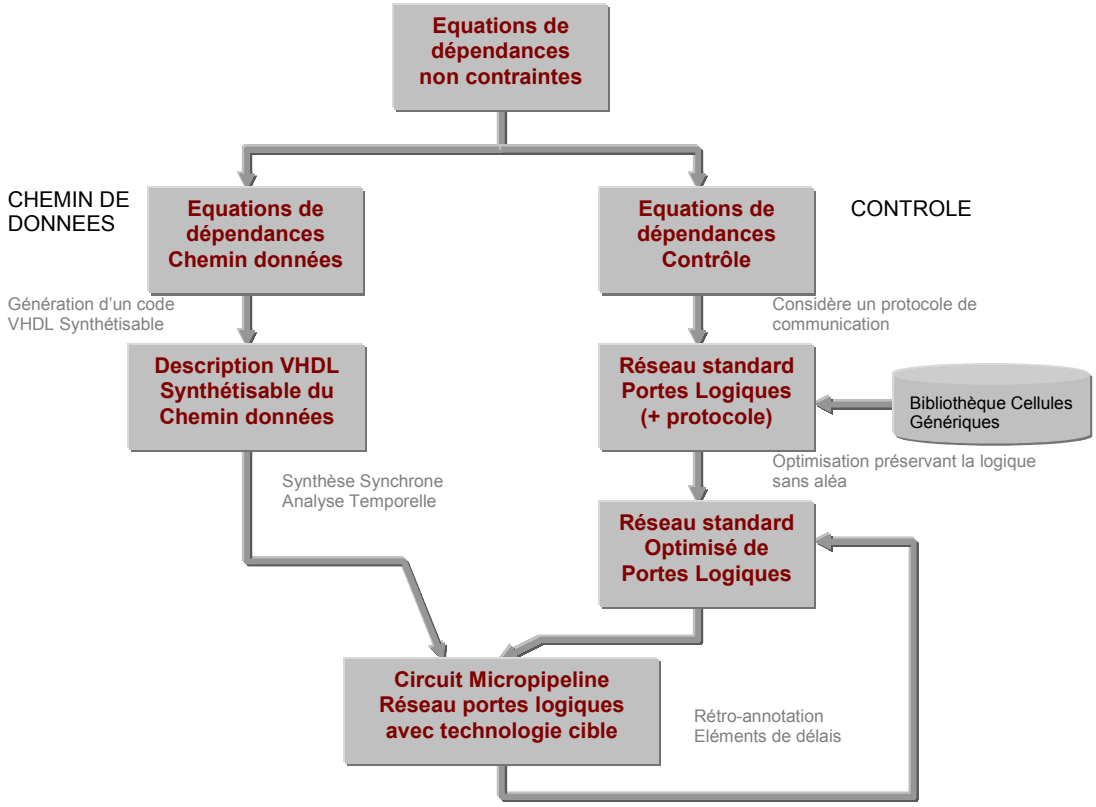


Figure 5.1 : Flot de synthèse micropipeline à partir des équations de dépendances

5.1 Synthèse du chemin de données

Le chemin de données en micropipeline est conçu selon le modèle de délai borné, contrairement au module de contrôle qui est insensible aux délais. Par conséquent, outre le fait que cette partie du circuit soit constituée d'opérateurs combinatoires et de registres ou latchs (cf. §3.3), elle n'a pas à gérer la contrainte du contrôleur qui doit être préservé de tout aléa. Ces caractéristiques permettent au chemin de données d'être synthétisé par les outils synchrones. C'est pourquoi le flot de synthèse est à ce niveau séparé et le traitement des chemins de données et des contrôleurs est distinct. La séparation s'effectue au niveau des équations de dépendances formalisant le fonctionnement du circuit asynchrone à générer. En effet, toutes les informations requises pour le calcul des sorties de données et des éléments de sélection sont disponibles dans l'équation de dépendances générale 5.1.

5.1.1 Génération des équations de dépendances du chemin de données

L'équation de dépendances des sorties de données en fonction des canaux d'entrée se présente comme suit :

$$S_k^{Gi-données} = F_{dep}^{Gi-données}(nom\_composant, nom\_processus, (init{=}0/1, cst), \prod_j E_{j-probes}^{Gi-données}(consom{=}0/1, valeur), \prod_j E_j^{Gi-données}, (G_i{=}F_{garde}^{Gi}(\prod_j E_j^{Gi-données})), F_{Sk}^{Gi}(\prod_j E_j^{Gi-données}))$$

Equation 5.5

Dans cette équation, il est mentionné que les sorties de données dépendent exclusivement des entrées courantes du circuit. L'information mentionnant les canaux d'entrée *probés* (sonde de canal) indépendamment des autres canaux d'entrée n'est pas importante. Simplement, nous avons gardé la même configuration que l'équation de dépendances générale de façon conventionnelle. Le cas où une initialisation d'un canal de sortie est effectuée dans le code avant d'entrer dans la boucle d'exécution ddu processus correspondant, est importante car elle change la structure du circuit (cf. §5.1.2.2). L'information cruciale demeure bien entendu, la nature précise de la fonction Fsk, qui relie les sorties avec les entrées de données uniquement.

Par ailleurs, l'équation 5.5 est une sous-équation de dépendances qui décrit la relation entre une sortie de donnée S_k relative à la garde Gi (*i.e :* dans une seule branche d'exécution). L'équation de dépendances d'une sortie en considérant l'ensemble des branches d'exécution est déduite en faisant l'union de l'ensemble des sous-équations de dépendances.

Equation 5.6 $$S_k^{données} = F_{Union}^{données}(\prod_k S_k^{Gi-données})$$

5.1.2 Génération du VHDL synthétisable pour le chemin de donnée

Les informations contenues dans l'équation de dépendances 5.5 des sorties de données nous permettent de générer une description « ***VHDL synthétisable*** » du chemin de donnée. Nous avons vu que le schéma bloc du circuit asynchrone micropipeline à générer après synthèse se présentait comme le montre la figure 5.2.

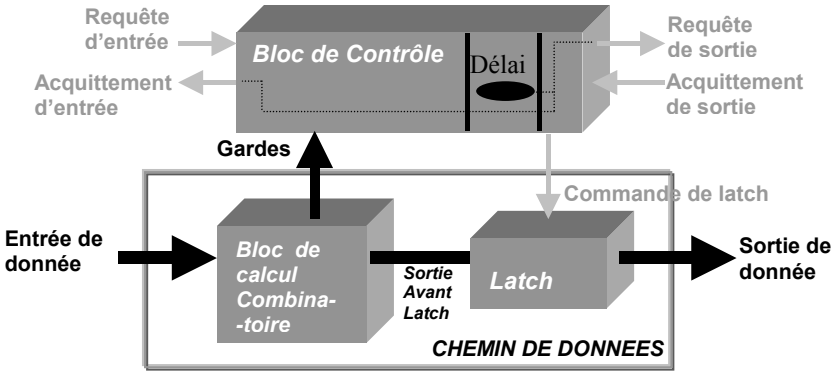


Fig. 5.2 : Schéma bloc de la structure du circuit micropipeline cible

Le chemin de données est constitué du bloc de fonction où s'effectuent les calculs combinatoires et de latchs. Le bloc de calcul reçoit à son entrée les flots de données rythmés localement par le signal de requête d'entrée du bloc de contrôle associé (qui sont aussi les requêtes de sorties du contrôle de l'étage précédent). Il produit les sorties de données qu'il dirige vers les latchs. Ces derniers assurent leur mémorisation avant qu'elles ne soient transmises à l'étage suivant lors de l'activation des signaux de commande de latch « ***cmd_latch_S*** » par le bloc de contrôle associé.

Le bloc combinatoire procède également au calcul des gardes qui traduisent les différentes branches d'exécution lors du traitement des structures de pipeline de « sélection » (multiplexage et démultiplexage). Ce calcul dépend, tout comme pour les sorties de données, exclusivement des valeurs courantes des canaux d'entrée et non des valeurs passées. Ces

gardes sont transmises au bloc de contrôle qui les exploite comme nous le voyons plus en détail ultérieurement dans la partie réservée à la synthèse des contrôleurs.

Globalement donc, le chemin de données génère deux types de sorties vers l'environnement : les gardes « $\mathbf{G_i}$ » pour le bloc de contrôle associé et les sorties de données « $\mathbf{S_k}$ » pour les étages suivants du pipeline. Ces deux types de sortie sont tous deux calculés exclusivement en fonctions des entrées de données courantes. Le chemin de données génère également un autre type de sortie interne : les sorties de données dénotées « $S_{k\text{-}BFL}$ » (sorties avant latch ou « *before latch* ») transmises des opérateurs combinatoires vers les latchs.
L'équation de dépendances 5.5 fournit la fonction de calcul de la sortie « S_k » pour une garde déterminée, ainsi que celle de la garde « G_i » en fonction des données des canaux d'entrée.

5.1.2.1 Cas particulier du protocole séquentiel

Le modèle de circuit cible donné figure 5.2 est général et indépendant des protocoles de communication présentés en §3.1.3. Toutefois, le protocole de communication séquentiel présenté en §3.3.1.1 fait office d'exception. En effet, ce protocole s'affranchit de latchs car il ne nécessite pas de mémoriser les données dans la partie chemin de données. Les données sont échangées au rythme du contrôle local entre les blocs de calcul combinatoire sans passer via les latchs.

5.1.2.2 Cas particulier des circuits d'initialisation

Un autre cas particulier est celui où le code du circuit asynchrone à synthétiser commence d'abord par initialiser une ou plusieurs des sorties de ce circuit avant d'entamer l'exécution de la boucle du processus. Dans ce cas particulier, le modèle de circuit à générer présenté en figure 5.2 nécessite d'être complété. En effet, l'initialisation d'une des sorties du circuit requiert l'ajout d'un second « buffer » dit « d'initialisation » (fig.5.3) pour chaque sortie initialisée du circuit. Nous traitons plus en détail ce type de circuit lors de la synthèse des contrôleurs.

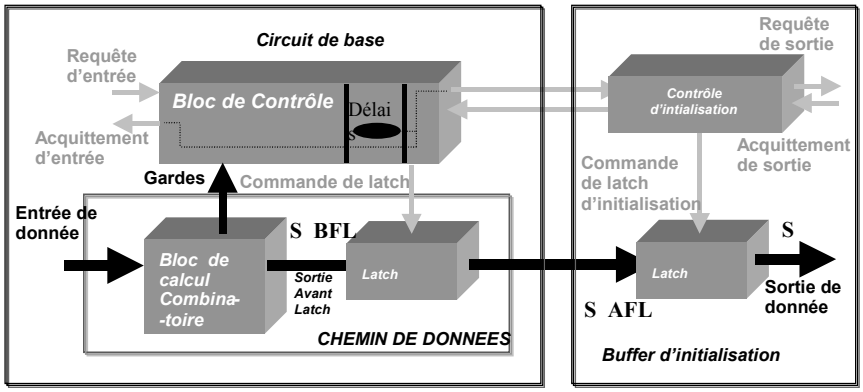


Fig. 5.3 : Modèle de circuit cible micropipeline pour les circuits asynchrones initialisés

5.1.2.3 Uniformisation du type de données

Les types de données du code CHP peuvent tous être ramenés au type générique « MR[B][L] ». L'équivalent VHDL de ce type est un vecteur « std_ulogic-vector » de taille ((L*[Log2(B)$_{sup}$)). Le type « MR[B][L] » est donc équivalent au type std_ulogic-vector((L*[Log2(B)$_{sup}$)-1 downto 0).

5.1.2.4 Génération du code VHDL synthétisable

Le code VHDL synthétisable décrivant le bloc de chemin de données d'un processus du circuit asynchrone à générer est un couple entité/architecture classique. L'entité contient les déclarations des entrées de données, des sorties de données, des gardes à générer, et des commandes de latch reçues des contrôleurs. Il est important de bien respecter la politique de nommage qui consiste à précéder les noms des gardes et des commandes de latchs par le nom du composant et du processus associés. L'architecture du composant commence par déclarer les signaux de gardes (Gi) et de sorties de l'opérateur combinatoire (S_BFL) car ces derniers sont utilisés dans le code généré en tant qu'entrée/sortie. A ce niveau, si le processus traité est « initialisant », nous déclarons le signal de donnée qui relie le circuit de base au buffer d'initialisation (S_AFL). Après avoir calculé les gardes et les sorties de données de l'opérateur combinatoire, les sorties du chemin de données sont obtenues par instantiation de latchs. Un modèle général de ce code est donné en annexe A.3.

5.1.3 Illustration

Si l'on reprend l'exemple du compteur (Cf. §4.6.7) :

```
-------------------------
component counter
-------------------------
component counter
port ( Go : in BD passive bit;
       S : out BD active bit[3])
channel CS : BD bit[3];
channel NS : BD bit[3];
begin
-------------------------
1er processus
-------------------------
process MAIN
port ( Go : in BD passive bit;
       CS : in BD active bit[3];
       NS : out BD active bit[3];
       S : out BD active bit[3])
variable css  :  bit[3];
[
  *[#Go =>
     [CS?css;
       @ [ css /= "1.1.1"[2] => NS!css , S!css
           css = "1.1.1"[2]  => Go? ; NS!"0.0.0"[2]
         ]
     ]
  ]
```

```
-------------------------
2nd processus
-------------------------
process INCREMENT
port (  CS : out BD passive bit[3];
        NS : in BD passive bit[3])
variable x  :  bit[3];

[CS!"0.0.1"[2];
  *[ NS?x ; CS!(x + 1)]
]
-------------------------
end counter;
-------------------------
```

Dans la première garde du processus « main », les sorties de données NS et S sont concurremment affectées et l'entrée de donnée Go n'est pas consommée. De ce fait l'équation de dépendances de données qui gère ces deux données pour la garde G0 sont identiques (Equation 5.7) :

$$NS^{G0-données} = S^{G0-données} = F_{dep}^{G0-données}(COUNTER, MAIN, (init=0), Go_{probes}^{G0-données}(consom=0), CS^{G0-données}, (G_0=(CS/="1.1.1"[2])), NS^{ack}, S^{ack}, (NS^{G0-données}=S^{G0-données}=CS^{données}))$$

Equation 5.7

L'équation de dépendances inhérente à la sortie de donnée NS affectée dans la seconde garde du processus « main » est donnée en équation 5.8 :

Equation 5.8

$$NS^{G1-données} = F_{dep}^{G1-données}(COUNTER, MAIN, (init=0), Go_{probes}^{G1-données}(consom=1), CS^{G1-données}, (G1=(CS="1.1.1"[2])), NS^{G1-ack}, (NS^{G1-données}="0.0.0"[1]))$$

La sortie de donnée NS est finalement gérée par l'union exclusive des deux équations 5.7 et 5.8 :

Equation 5.9

$$NS^{données} = F_{Union}^{données}(NS^{G0-données}, NS^{G1-données})$$

La sortie S n'est affectée que dans la garde G0, alors il vient:

Equation 5.10

$$S^{données} = S^{G0-données}$$

La sortie CS du processus « increment » est gérée par l'équation 5.11. Remarquons que le paramètre « init » est actif car le processus « increment » est « initialisant ».

Equation 5.11

$$CS^{données} = F_{dep}^{données}(COUNTER, INCREMENT, (init=1, CS_{init}="0.0.1"[2]), NS^{données}, CS^{ack}, (CS^{données}=NS^{données}+"0.0.1"[2]))$$

A partir des équations de dépendances 5.7 à 5.10 on génère le code VHDL synthétisable du chemin de donnée correspondant au processus « main » :

```
-- Librairie IEEE standard
library IEEE ;
use IEEE .std_logic_1164.all ;

-- Librairie propre – package latch
library LIB_COUNTER;
use LIB_COUNTER.pack_latch.all;

entity COUNTER_MAIN_dp is
port(--les entrées de données
      Go_data : in std_ulogic ;
     --bit[3]=MR[2][3]->std_ulogic_vector((L*[Log2B]sup)-1)..0)
      CS_data : in std_ulogic_vector(2 downto 0);

      -- les sorties de données
      S_data : out std_ulogic_vector(2 downto 0);
      NS_data : out std_ulogic_vector(2 downto 0);

       -- les gardes Gi
      counter_main_G0 :  out std_ulogic;
      counter_main_G1 : out std_ulogic;

       -- une commande de latch par sortie de donnée
      Cmd_latch_S : in std_ulogic;
      Cmd_latch_NS : in std_ulogic)

end COUNTER_MAIN_dp;

architecture Arch_COUNTER_MAIN_dp of
                        COUNTER_MAIN_dp is
  -- signaux de garde et de sortie de données avant latchs
  signal s_G0, s_G1 :  std_ulogic;
  signal S_data_BFL, NS_data_BFL : std_ulogic_vector(2 downto 0);
```

```
begin
-- calcul des gardes en fonction des entrées de données
s_G_0 <= '1' when CS_data /="111" else
              '0';
s_G_1 <= '1' when CS_data = "111" else
             '0';
-- calcul des sorties de données avant latch en fonction des entrées
--de données selon la branche d'éxécution –Equation 5.5
---et 5.6 ie par l'exclusivité des gardes)
NS_data_BFL <= CS_data when (s_G_0 = '1') else
                        "000"  when (s_G_1 = '1') else
                        "---";
S_data_BFL <= CS_data when (s_G_0 = '1') else
                    "---";
-- affectation concurrente des gardes par les signaux de gardes
counter_main_G0 <= s_G_0;
counter_main_G1 <= s_G_1;

-- calcul des sorties de données à transmettre aux étages
-- suivant de pipeline par instantiation
instance_S_0 : latch_resetb port map( resetb, S_data(0),
                                      cmd_latch_S, S_data_BFL(0));
instance_S_1 : latch_resetb port map( resetb, S_data(1),
  cmd_latch_S, S_data_BFL(1));
instance_S_2 : latch_resetb port map( resetb, S_data(2),
  cmd_latch_S, S_data_BFL(2));

instance_NS_0 : latch_resetb  port map( resetb, NS_data(0),
                                cmd_latch_NS, NS_data_BFL(0) );
instance_NS_1 : latch_resetb  port map( resetb, NS_data(1),
                                cmd_latch_NS, NS_data_BFL(1) );
instance_NS_2 : latch_resetb  port map( resetb, NS_data(2),
                                cmd_latch_NS, NS_data_BFL(2) );

end Arch_COUNTER_MAIN_dp;
```

Le code VHDL synthétisable inhérent au chemin de données du processus « increment » est généré grâce à l'équation de dépendances 5.11 :

```
-- Librairie IEEE standard
library IEEE ;
use IEEE .std_logic_1164.all ;
use IEEE .std_logic_unsigned.all ;

-- Librairie propre – package latch
library LIB_COUNTER;
use LIB_COUNTER.pack_latch.all;
--  et des latch sans reset ni set

Entity COUNTER_INCREMENT_dp is
 Port (
       resetb :  in std_ulogic ;
       NS_data : in std_ulogic_vector(2 downto 0);  -- bit[3]
       CS_data : out std_ulogic_vector(2 downto 0);  -- bit[3]

       -- les gardes Gi
       -- pas de gardes

       -- une commande de latch par sortie de donnée (bloc de
base) +
       -- une commande de latch pour le bloc d'initialisation
       Cmd_latch_CS :  in std_ulogic;
       Cmd_latch_CS _init  :  in std_ulogic
      );
End COUNTER_INCREMENT_dp;

Architecture Arch_COUNTER_INCREMENT_dp of
COUNTER_INCREMENT_dp is
-- Signaux intermédiaires
-- signaux de données de sortie avant latch
 s_ CS _BFL : std_ulogic_vector(2 downto 0);
-- signaux de données de sortie entre le
-- latch de base et le latch d'initialisation
 s_ CS _AFL : std_ulogic_vector(2 downto 0);
 Begin
-- Sortie avant latchs
s_ CS _BFL  <=
     To_StdULogicVector(To_StdLogicVector (NS_data) + "001") ;

-- Sortie de données entre latchs et après latchs
Instance_s_CS_AFL_0: latch_resetb
                     port map(resetb, s_CS_AFL(0),
                     Cmd_latch_CS,  s_CS_BFL(0));
Instance_CS_0: latch  port map(CS_data(0),
                     Cmd_latch_CS_init, s_CS_AFL(0) ) ;

Instance_s_CS_AFL_1: latch_resetb
                     port map(resetb, s_CS_AFL(1),
                     Cmd_latch_CS, s_CS_BFL(1));
Instance_CS _1: latch  port map(CS_data(1),
                     Cmd_latch_CS_init, s_CS_AFL(1) ) ;

Instance_s_CS_AFL_2: latch_resetb
                     port map(resetb, s_CS_AFL(2),
                     Cmd_latch_CS,s_CS_BFL(2));
Instance_CS _2: latch  port map(CS_data(2),
                     Cmd_latch_CS_init, s_CS_AFL(2) ) ;

End Arch_COUNTER_INCREMENT_dp;
```

5.1.4 Extraction des délais de calcul dans le « chemin de données »

Nous avons vu précédemment que le modèle de circuit cible que nous proposons fait en sorte que le bloc combinatoire fournit le signal de fin de calcul en fonction des données des canaux d'entrée. Ces délais sont exploités par le bloc de contrôle, et le fait qu'ils soient adaptés au calcul effectué dans les blocs de fonctions permet d'avoir un pipeline élastique.

Ces délais sont d'abord considérés avec des valeurs a priori dans la synthèse des contrôleurs, puis lors de la rétro-annotation on introduit les valeurs réelles de ces délais, après qu'ils aient été générés par l'outil de synthèse commercial utilisé pour réaliser la synthèse du chemin de données. En effet, ces outils de synthèse commerciaux permettent moyennant l'écriture d'un script d'effectuer des « *report* » qui donnent des informations de délais pour chaque branche d'exécution (Cf. §4.4 portant éclatement des structures de sélection).

Dans les faits une fois les délais récupérés via la synthèse synchrone du chemin de donnée on crée un package de délais définissant l'ensemble des délais des sorties de données selon la garde à laquelle les sorties appartiennent. Ce package se présente comme suit :

```
On pose les sigles suivants :
CN : ComponentName
PN : ProcessName

-- Bibliothèque IEEE standard
library IEEE ;
use IEEE.std_logic_1164.all ;


PACKAGE Delay_Comp IS

-- Delai d'un latch
CONSTANT Latch_Delay : TIME := 0.1 ns ;



-- Processus P1
-- Délai max. pour le calcul des différentes gardes du processus P1
CONSTANT maxdelay_CN_P1_G0_Gn : TIME := 0.1 ns ;

-- Délai max. pour le calcul d'une même sortie dans différentes gardes
-- du processus P1
CONSTANT maxdelay_CN_P1_S_G0 : TIME := 0.1 ns ;
CONSTANT maxdelay_CN_P1_S_G1 : TIME := 0.1 ns ;
…
CONSTANT maxdelay_CN_P1_S_Gn : TIME := 0.1 ns ;
```

```
-- Processus P2
-- Délai max. pour le calcul des différentes gardes du processus P2
CONSTANT maxdelay_CN_P2_G0_Gn : TIME := 0.1 ns ;

-- Délai max. pour le calcul d'une même sortie dans différentes
gardes  du processus P2
CONSTANT maxdelay_CN_P2_S_G0 : TIME := 0.1 ns ;
CONSTANT maxdelay_CN_P2_S_G1 : TIME := 0.1 ns ;
…
CONSTANT maxdelay_CN_P2_S_Gn : TIME := 0.1 ns ;
--… idem pour les autres processus s'ils existent

COMPONENT Delay
 GENERIC( Delay_Time : TIME := 0.1 ns );
 PORT( Out_Delay : OUT STD_ULOGIC;
            In_Delay  : IN STD_ULOGIC
    );
END COMPONENT;


END Delay_Comp ;

PACKAGE BODY Delay_Comp IS
END Delay_Comp ;
```

Le composant de délai est défini génériquement comme suit :

```
-- Bibliothèque IEEE standard
library IEEE ;
use IEEE.std_logic_1164.all ;

ENTITY Delay IS
 GENERIC( Delay_Time : TIME := 0.1 ns );
 PORT( Out_Delay : OUT STD_ULOGIC;
             In_Delay  : IN STD_ULOGIC);
END Delay ;

ARCHITECTURE Arch_Delay OF Delay IS
BEGIN
          Out_Delay <= In_Delay  AFTER Delay_Time ;
END Arch_Delay ;_Latch_TMP;
END Arch_Latch ;
```

5.2 Synthèse des contrôleurs

Nous avons déjà eu l'occasion de voir que la performance d'un pipeline asynchrone se mesure par un certain nombre de paramètres (Cf. §3.1.2) parmi lesquelles l'un d'eux s'avère être très pertinent pour les circuits asynchrones du type micropipeline. Il s'agit du temps de cycle qui exprime, rappelons-le, le temps maximum qui sépare la prise en compte de deux données successives dans un étage du pipeline. En d'autres termes, il correspond au débit maximal local d'un étage du pipeline. La pertinence de ce paramètre tient au fait qu'il participe efficacement à l'amélioration de l'élasticité du pipeline laquelle constitue l'intérêt majeur du développement des circuits asynchrones de type micropipeline.

Ce temps de cycle varie notamment selon le protocole de communication utilisé. Dans le cadre de cette thèse nous ne nous intéressons qu'aux protocoles de communication quatre phases. Ces derniers se distinguent fondamentalement par l'enchaînement des transitions appliqué aux signaux de requêtes et d'acquittements. Ces différents séquencements des transitions confèrent à l'implémentation des protocoles des propriétés qui varient en termes de vitesse, consommation et complexité (surface). Dans les faits, la synthèse du contrôleur dans

les protocoles WCHB, PCHB et PCFB consiste à réaliser le circuit qui doit permettre la commande d'ouverture et de fermeture « cmd_latch_S_k » des latchs faisant office de registre mémoire.

Dans les paragraphes qui suivent nous développons pour chacun de ces trois protocoles les différentes étapes de la méthodologie d'Alain Martin [MAR 93] suivie pour générer les contrôleurs associés à ces protocoles. Pour chacun d'entre eux nous écrivons explicitement l'équation de dépendances contrainte équivalente, puis nous vérifions sa correction fonctionnelle. Cette méthode est élargie aux structures de pipeline non linéaires et aux circuits d'initialisation. Sur la base de ce travail, nous définissons les algorithmes de génération automatique des contrôleurs asynchrones sous forme d'équations de dépendances contraintes par le protocole de communication et utilisant une bibliothèque de cellules standards.

5.2.1 Méthodologie de génération d'un circuit Micropipeline en fonction d'un protocole

La méthodologie d'Alain Martin [MAR93] poursuivie pour réaliser un circuit micropipeline selon le protocole de communication commence par l'étape de décomposition des processus dans le but de faciliter les étapes suivantes de la synthèse asynchrone. Nous considérons cette étape effectuée notamment parce que l'on procède à la simplification du réseau de Petri en plusieurs sous-réseaux de Petri exempts de la structure de sélection. Ensuite on procède à l'expansion des communications qui consiste à réordonnancer les signaux de communications en respectant le protocole choisi. Lors de cette étape, il faut veiller à maintenir la correction du circuit initial. En dernier lieu, intervient l'étape de génération des règles de production qui définissent précisément les relations entre les signaux de sortie et leurs homologues d'entrée.

5.2.1.1 Génération d'un circuit Micropipeline en protocole WCHB

Dans le protocole WCHB, il y a synchronisation des phases montantes (mise à un) et descendantes (remise à zéro) entre les canaux d'entrée et de sortie. En effet, dans la phase montante, le latch n'est fermé que lorsque la sortie de donnée est disponible et que l'on détecte la présence de données valides sur le canal d'entrée. L'entrée peut alors être acquittée et le contrôleur génère le signal de requête pour l'étage suivant. Dans la phase descendante, on attend que les données de sortie soit acquittées et que le signal de requête du canal d'entrée repasse à son état initial (zéro) pour rendre le latch à nouveau transparent. De la sorte, les états du latch alternent entre ouvert et fermé. Nous traitons pour ce protocole les cas linéaire, non linéaires ainsi que la structure d'initialisation.

5.2.1.1.1 *Protocole WCHB appliqué à une structure asynchrone linéaire*

Un étage de pipeline linéaire étant un étage avec un seul canal d'entrée et un seul canal de sortie, son code CHP peut s'écrire : **[E? ; S! ; LOOP]**. L'expansion des communications avec ré-ordonnancement, c'est à dire l'écriture explicite des actions de communication avec une permutation bien choisie des actions élémentaires du protocole de communication WCHB permet d'obtenir une description implémentable. En terme de notations, « [Si] » et « [/Si] » sont des instructions de test de signaux : ils testent si le signal d'entrée « Si » est respectivement disponible ou non disponible. Les instructions « So+ » et « So- » sont des instructions d'assignation de signaux : ils signifient que le signal de sortie « So » est rendu respectivement actif ou non actif.

```
[
    [ E_req & S_ack ] ; S_req+ ; E_ack- ;
    [ /E_req & /S_ack ] ; S_req- ; E_ack+ ; LOOP
]
```

A l'exception de la directive de séquentialité « ; » toutes les instructions ainsi obtenues par l'expansion des communications sont implémentables. C'est pourquoi la prochaine étape consiste à supprimer ces « ; » et à les remplacer par un ensemble de règles dites de production :

```
-- Requête de sortie                    --Acquittement d'entrée
S_req+ <= [E_req & S_ack]               E_ack- <= [S_req]
S_req- <= [/E_req & /S_ack]             E_ack+ <= [/S_req]
```

Pour chaque signal de sortie on construit à partir de ces règles de production une cellule à une sortie et à plusieurs entrées.

```
S_req+ <= [E_req & S_ack]
S_req- <= [/E_req & /S_ack]
```

→ *Porte de Muller*

```
E_ack- <= [S_req]
E_ack+ <= [/S_req]
```

→ *Inverseur logique*

Le circuit est alors implémenté par l'interconnexion de l'ensemble de ces cellules (fig.5.4).

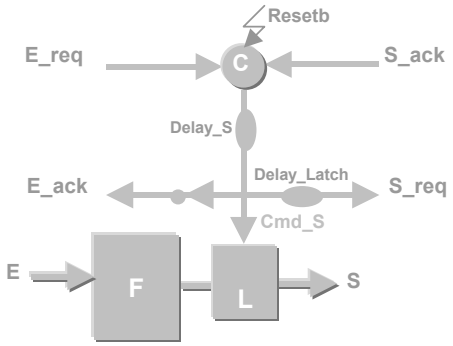


Fig. 5.4: Micropipeline WCHB d'une structure linéaire

L'élément de retard « delay_S » est défini par l'opérateur combinatoire. Il correspond au temps nécessaire pour le calcul de la sortie « S_data ». Par exemple, dans le cas du buffer, la valeur de ce délai est nulle car l'entrée du buffer est directement transférée sur sa sortie sans calcul intermédiaire. Le signal de requête à la sortie du contrôleur est différé par rapport au signal de commande de latch, d'un délai « delay_latch » relatif au temps de traversée du latch.

Les équations de dépendances (Equations 5.12) contraintes par le protocole de communication WCHB et inhérentes à ce circuit sont écrites en utilisant les éléments d'une bibliothèque de cellules standard définie au sein du groupe CIS [RIG 02]. Cette bibliothèque implémente notamment plusieurs variantes de la porte de Muller (cf. §1.2.3.4).

Commande de latch : **cmd_latch_S** = **Delay_S**(**Muller2_R**(Resetb, E_req, S_ack))

Requête de sortie du contrôleur : **S_req** = **Delay_latch**(cmd_latch_S)

Acquittement d'entrée du contrôleur : **E_ack** = **not**(cmd_latch_S)

Equations 5.12

Les signaux d'acquittement E_ack et S_ack étant actifs au niveau bas, le fonctionnement de ce circuit est tel qu'au départ après le signal de reset, les signaux de requêtes E_req et S_req sont au niveau bas et les signaux d'acquittement E_ack et S_ack sont au niveau haut. Puis le passage du signal E_req au niveau haut signale l'arrivée de la donnée. Le signal à la sortie de la porte de Muller passe alors au niveau haut puis est retardé par un délai équivalent au temps nécessaire au calcul de la sortie S. Après cela, le signal d'acquittement d'entrée E_ack descend grâce à l'inverseur, et de façon concurrente, le signal de requête de la sortie S_req monte après avoir été retardé d'un délai correspondant au passage des données au travers du latch L, et le signal de commande de latch déclenche le fonctionnement de ce dernier. Ensuite l'environnement côté sortie acquitte la donnée reçue en faisant descendre le signal S_ack, et l'environnement côté entrée réagit à la descente du signal E_ack en remettant à zéro le signal E_req. Cela provoque le retour à zéro du signal S_req (auquel l'environnement côté sortie réagit en faisant passer S_ack au niveau haut), et le retour à un du signal E_ack. Nous retrouvons ainsi l'état initial du circuit. Ce fonctionnement peut être explicité par les implications suivantes :

```
Etat initial : Resetb = 0 => (S_req=0 et E_ack=1)

E_req=0 et S_ack=1 => Aucun changement en sortie de contrôleur

E_req=1 et S_ack=1 => (S_req=1 et E_ack=0) - C'est la phase montante
                                              du protocole WCHB

E_req=0 et S_ack=0 => (S_req=0 et E_ack=1)
                                    - C'est la phase descendante
                                      du protocole WCHB
```

5.2.1.1.2 *Protocole WCHB appliqué à une structure asynchrone non-linéaire*

Un étage de pipeline non linéaire est un étage qui peut avoir plusieurs canaux d'entrées et plusieurs canaux de sorties. Le code CHP descriptif d'une telle structure de pipeline peut s'écrire **[Ej ? var_j ; Sk ! fk(var_j) ; LOOP]**. En appliquant la même démarche que pour le cas du pipeline linéaire (expansion des communication, ré-ordonnancement, puis génération des règles de production) nous aboutissons au circuit de la figure 5.5.

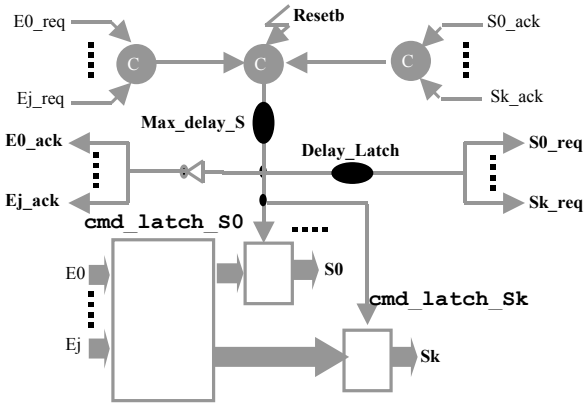


Fig. 5.5: Micropipeline WCHB d'une structure non-linéaire

L'élément de retard « max_delay_S » est une hypothèse temporelle définie par l'opérateur combinatoire et relative au temps maximum nécessaire au calcul des sorties « Sk ». Cet élément de retard dépend des fonctions « fk » et du type des sorties « Sk ».
Les équations de dépendances (5.13) contraintes par le protocole WCHB et relatives au pipeline non linéaire sont les suivantes :

Commandes de latch **:**
cmd_latch_Sk =
maxdelay_S(**Muller2_R**(Resetb,**MullerJ**($\prod_j Ej\_req$),**MullerK**($\prod_k Sk\_ack$)))

Requêtes de sortie du contrôleur **: Sk_req** = **Delay_latch**(cmd_latch_Sk)

Acquittements d'entrée du contrôleur **: Ej_ack = Not**(cmd_latch_Sk)

Equations 5.13

Le fonctionnement logique du circuit ainsi généré est présenté ci-dessous. Les notations $\prod_k Sk\_req$ et $\prod_j Ej\_ack$ signifient respectivement l'ensemble des requêtes de sorties et des acquittement d'entrées.

Etat initial : Resetb = 0 => ($\prod_k Sk\_req$ = 0 et $\prod_j Ej\_ack$ =1)

$\prod_j Ej\_req$ = 0 et $\prod_k Sk\_ack$ =1 => Aucun changement en sortie de contrôleur

$\prod_j Ej\_req$ = 1 et $\prod_k Sk\_ack$ =1 => ($\prod_k Sk\_req$ = 1 et $\prod_j Ej\_ack$ =0)

- C'est la phase montante du protocole WCHB

$\prod_j Ej\_req$ = 0 et $\prod_k Sk\_ack$ =0 => ($\prod_k Sk\_req$ = 0 et $\prod_j Ej\_ack$ =1)

-C'est la phase descendante du protocole WCHB

▪ ***Application***

Les cas de fourche à deux sorties et de convergence à deux entrées sont des cas particuliers de structures pipeline non linéaire (figure 3.21 du §3.3.2.2). Leurs équations de dépendances (5.14 et 5.15) contraintes par le protocole WCHB se présentent comme suit :

FOURCHE

cmd_latch_S1 = **cmd_latch_S2** = **max_delay_S**(**Muller2_R**(Resetb, E_req, **Muller2**(S1_ack, S2_ack)))

S1_req = **S2_req** = **Delay_latch**(cmd_latch_S1)

E_ack = **Not**(cmd_latch_S1)

Equations 5.14

CONVERGENCE

```
cmd_latch_S = max_delay_S(Muller2_R(Resetb, Muller2(E1_req, E2_req,
                                                    S_ack))

S_req = Delay_latch(cmd_latch_S)

E1_ack = E2_ack = Not(cmd_latch_S)
```

Equations 5.15

Le fonctionnement logique des circuits de fourche à deux sorties et de convergence à deux entrées dans le protocole de communication WCHB s'écrit :

FOURCHE

```
Etat initial :  Resetb = 0    => (S1_req=0 et S2_req=0 et E_ack=1)

E_req=0 et S1_ack=1 et S2_ack=1 => Aucun changement en sortie

E_req=1 et S1_ack=1 et S2_ack=1 => (S1_req=1 et S2_req=1 et E_ack=0)
                                  - C'est la phase montante du protocole

E_req=0 et S1_ack=0 et S2_ack=0 => (S1_req=0 et S2_req=0 et E_ack=1)
                                - C'est la phase descendante du protocole
```

CONVERGENCE

```
Etat initial :  Resetb = 0    => (S_req=0 et E1_ack=1 et E2_ack=1)

E1_req=0 et E2_req=0 et S_ack=1 => Aucun changement en sortie

E1_req=1 et E2_req=1 et S_ack=1 => (S_req=1 et E1_ack=0 et E2_ack=0)
                                  - C'est la phase montante du protocole

E1_req=0 et E2_req=0 et S_ack=0 => (S_req=0 et E1_ack=1 et E2_ack=1)
                                - C'est la phase descendante du protocole
```

5.2.1.1.3 *Protocole WCHB appliqué à une structure asynchrone linéaire avec initialisation*

Si l'on suppose que l'on initialise la seule sortie du pipeline linéaire à la valeur « 1 » alors le code CHP relatif au circuit linéaire avec initialisation s'écrit **[S!1 ; [E? ; S!] ; LOOP]**. En procédant à l'expansion des communications avec ré-ordonnancement et en respectant le protocole WCHB nous obtenons :

```
[                                          --Boucle
 --remise à 0                              *[
 [ /RESETb ] ; S_req- , E_ack+ ;            [E_req & S_ack]; S_req+; E_ack-;

 --Initialisation de la sortie S            [/E_req & /S_ack];S_req-;E_ack+;
 [ RESETb ]          ; S_req+ ;             ]
 [ /E_req & /S_ack]; S_req- ; E_ack+;       ]
```

La suppression des opérateurs de séquentialité « ; » et leur remplacement par un ensemble de règles de production implémentables donne :

```
-- Requête de sortie                          --Acquittement d'entrée
S_req+ <=   [ RESETb & S_req- ]… (1)   E_ack+ <=   [ /RESETb ] … (5)
          | [ E_req & S_ack ]  … (2)             | [ /S_req- ] … (6)

S_req- <=   [ /RESETb ]        … (3)   E_ack- <=   [ /S_req+ ] … (7)
          | [ /E_req & /S_ack ]… (4)
```

Sans tenir compte des délais les égalités (2), (3), (4), (6) et (7) donnent les équations de dépendances suivantes :

```
S_req = Muller2_R( RESETb, E_req, S_ack )

E_ack = not( S_req )
```

Or, l'égalité (1) montre la nécessité d'introduire une variable supplémentaire « X » telle que :

- X = '1' à la remise à 0 (/RESETb),
- X reste à '1' lorsqu'on relâche le RESET (RESETb),
- S_req = AND(X, RESETb)

L'introduction de cette variable va en quelque sorte découpler les entrées du circuits d'avec ses sorties. Ce découplage se fait par l'introduction de deux variables dénotées « sig_req » et « sig_ack ».

La ré-écriture des expansions de communications donne alors :

```
[
    --remise à 0
    [ /RESETb ] ; ( (S_req- , X+)  ; sig_ack-) , ( sig_req- ; E_ack+ ) ;

    --Initialisation de la sortie S
    [ RESETb ] ; S_req+ ;
    [ /E_req & /sig_ack ] ; sig_req- ; E_ack+ ;
    [ /sig_req & /S_ack ] ; ( X- ; S_req-) ; sig_ack+ ;

    --Boucle
    [
     [ E_req & sig_ack ] ; sig_req+ ; E_ack- ;
     [ sig_req & S_ack ] ; ( X+ ; S_req+) ; sig_ack- ;

     [ /E_req & /sig_ack ] ; sig_req- ; E_ack+ ;
     [ /sig_req & /S_ack ] ; ( X- ; S_req-) ; sig_ack+ ; LOOP
    ]
 ]
```

La phase de génération des règles de production pour rendre le circuit implémentable donne à ce moment :

```
-- Signal X intermédiaire
X+ <= [ /RESETb ]
    | [ sig_req & S_ack ]
X- <= [ /sig_req & /S_ack ]

-- Requête de sortie
S_req+ <=   [ RESETb & X]
S_req- <=   [ /RESETb ]

-- Requête de sortie intermédiaire
sig_req+ <= [ E_req & sig_ack]
sig_req- <= [ /RESETb ]

--Acquittement d'entrée
E_ack+ <=   [ /RESETb ]
         | [ E_req & /sig_ack ]
E_ack- <=   [ /E_req & /sig_ack ]

--Acquittement d'entrée intermédiaire
sig_ack+ <= /X
sig_ack- <= X
```

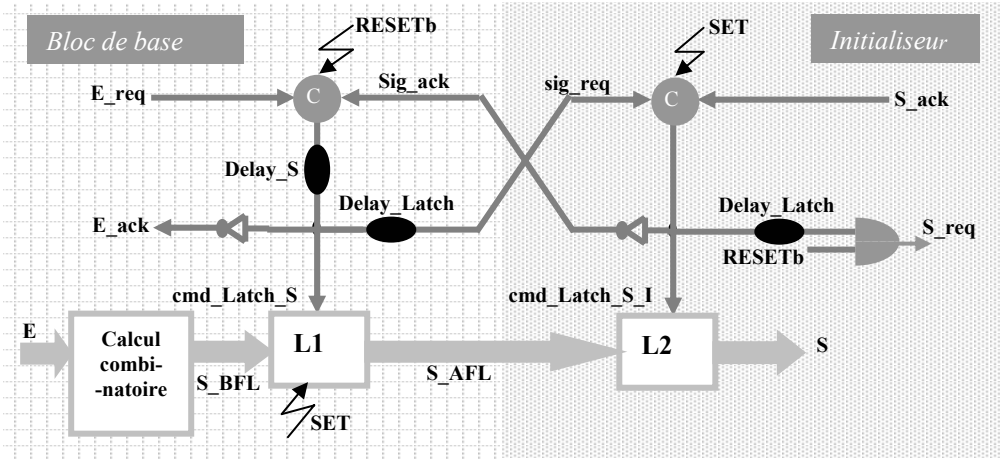


Fig. 5.6: Contrôleur WCHB d'une structure linéaire avec initialisation

Pour ce type de circuit, les équations de dépendances contraintes par le protocole WCHB sont données pour le bloc de base (5.16) et pour le bloc d'initialisation (5.17). Le bloc de base est un circuit asynchrone linéaire alors que le bloc d'initialisation est un demi-buffer et son opérateur combinatoire est la fonction unité.

```
-- Bloc de base (sans init)
cmd_latch_S = Delay_S( Muller2_R(RESETb, E_req, sig_ack ) )

E_ack  = not(cmd_latch_S)

sig_req = Delay_latch(cmd_latch_S)
```

Equations 5.16

```
-- Bloc d'initialisation
cmd_latch_S_I = Delay_S( Muller2_S( SET, sig_req, S_ack ) )

sig_ack = not( cmd_latch_S_I )

S_req = AND(RESETb, cmd_latch_S_I)
```

Equations 5.17

Le fonctionnement logique d'un circuit asynchrone linéaire avec initialisation s'exprime comme suit :

```
[RESETb=0 (≡ SET=1)]          =>    (sig_req=0 et E_ack=1) et
                                    (X=1 et S_req=0 et sig_ack=0)
                              -C'est la phase de l'initialisation

[RESETb=1 (≡ SET=0) et        => (S_req=1)
                     -C'est la phase de relâchement du Reset
                     -Une donnée est émise puis on rentre dans la boucle

[sig_req=0 et S_ack=0]         => (S_req=0 et sig_ack=1)

[E_req=1     et sig_ack=1]     => (sig_req=1 et E_ack=0)
[sig_req=1 et S_ack=1]         => (S_req=1 et sig_ack=0)
                               - C'est la phase montante du protocole

[E_req=0      et sig_ack=0]     => (sig_req=0 et E_ack=1)
[sig_req=0  et S_ack=0]         => (S_req=0 et sig_ack=1)
                              - C'est la phase descendante du protocole
```

5.2.1.1.4 *Initialisation des circuits asynchrones non linéaires en WCHB*

L'initialisation des circuits asynchrones non-linéaires telle que la fourche ou la convergence se déduit du cas linéaire puisqu'à chaque sortie de donnée initialisée on associe un buffer d'initialisation.

5.2.1.2 Génération d'un circuit Micropipeline en protocole PCHB

Le protocole WCHB permet au circuit de fonctionner correctement, toutefois le taux d'occupation du pipeline n'est que de cinquante pour cent. Cela est dû au fait que la mémorisation d'une donnée dans un étage du pipeline (fermeture du latch) n'a lieu que lorsque l'étage qui suit est vide (acquittement de sortie S_{ack} à zéro).

Le contrôleur PCHB dit semi-découplé permet d'augmenter le taux de remplissage dans le pipeline en augmentant le découplage entre la sortie et l'entrée du contrôleur. En effet, la phase montante de ce protocole est identique à celle du protocole WCHB, mais il y a désynchronisation au niveau des remises à zéro des signaux de requête et d'acquittement. Cela autorise la communication en sortie indépendamment des entrées. La donnée est ainsi mémorisée dans le latch avant que le protocole ne soit terminé côté sortie.

La méthode suivie pour définir l'algorithme de génération automatique des équations de dépendances contraintes par le protocole PCHB n'est montrée en détail que pour le cas des circuits asynchrones PCHB linéaires. Les équations de dépendances contraintes pour les circuits asynchrones PCHB non linéaires se déduisent de la même façon que pour le cas WCHB (Cf. §5.2.1.1.2).

Nous avons déjà vu que le code CHP inhérent à un circuit asynchrone linéaire s'écrivait **[E? ; S!; LOOP]**. L'expansion des communications avec ré-ordonnancement en respectant le protocole PCHB donne :

```
[
   [ E_req & S_ack ] ; S_req+ ; E_ack- ;
   (  [ not(S_ack) ] ; S_req-  );
   (  [ not(E_req) ] ; E_ack+  ) ; LOOP
]
```

Les instructions de séquentialité « ; » n'étant pas implémentable, nous générons les règles de production pour ce circuit :

```
-- Requête de sortie                -- Acquittement d'entrée
S_req+ <= E_req & S_ack             E_ack+ <= /E_req & S_req-
S_req- <= /S_ack & E_ack-           E_ack- <= S_req+
```

Nous remarquons d'emblée que la phase descendante du signal de requête « S_req » et la phase montante du signal d'acquittement « E_ack » dépendent entre autre de signaux de **sortie** du contrôleur, respectivement « E_ack » et « S_req ». Le circuit de contrôle est alors bouclé. Il est implémenté par l'interconnexion des différentes cellules générées par les règles de production (fig. 5.7).

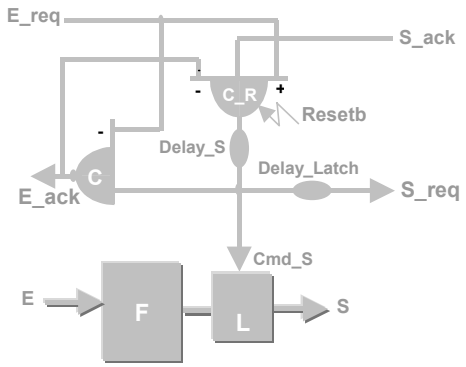


Fig. 5.7 : Micropipeline PCHB d'une structure linéaire

Les équations de dépendances (5.18) contraintes par le protocole PCHB et correspondant à la structure de pipeline linéaire sont données comme suit :

```
cmd_latch_S = Delay_S( Muller3D1P1N_R( Resetb, S_ack, E_req, E_ack ) )

S_req = Delay_latch( cmd_latch_S )

E_ack =  Muller2D1NB( S_req,  E_req )
```

Equations 5.18

Le fonctionnement logique du circuit de contrôle du pipeline linéaire PCHB s'exprime comme ci-dessous :

```
Etat initial : Resetb = 0  => (S_req=0 et E_ack=1)

E_req=0 et S_ack=1         => Aucun changement en sortie contrôle

E_req=1 et S_ack=1     => (S_req=1 et E_ack=0)
                           - C'est la phase montante du protocole

S_ack=0           => S_req=0
E_req=0           => E_ack=1
                       - C'est la phase descendante du protocole
```

5.2.1.3 Génération d'un circuit Micropipeline en protocole PCFB

Le contrôleur peut encore être amélioré par le protocole PCFB qui découple totalement les canaux d'entrée et de sortie dans la phase de remise à zéro. Ce découplage total améliore sensiblement le temps de cycle du pipeline. Le circuit obtenu est toutefois plus complexe que ceux répondant aux protocoles WCHB ou PCHB, mais il reste moins coûteux qu'une solution utilisant un protocole deux phases et basé sur la mise en œuvre de cellules toggle [RIG 02].

Comme pour le cas du protocole PCHB, la méthode suivie pour définir l'algorithme de génération automatique des équations de dépendances contraintes par le protocole PCFB n'est détaillée que pour le cas des circuits asynchrones PCFB linéaires. Les équations de dépendances contraintes pour les circuits asynchrones PCFB non linéaires se déduisent de la même façon que pour le cas WCHB (Cf. §5.2.1.1.2).

La phase d'expansion des communications avec ré-ordonnacement donne :

```
[
   [ E_req & S_ack ] ; S_req+ ; E_ack- ;
   (  [ /S_ack ] ; S_req-  ),
   (  [ /E_req ] ; E_ack+  ) ; LOOP
]
```

Il vient alors les règles de productions suivantes :

```
-- Requête de sortie              -- Acquittement d'entrée
S_req+ <= E_req & S_ack           E_ack+ <= /E_req & E_ack-
S_req- <= /S_ack & E_ack-         E_ack- <= S_req+
```

On remarque que la sortie E_ack+ dépend de la sortie E_ack-. Cela signifie que pour définir le signal d'acquittement d'entrée « E_ack » il est nécessaire de connaître son état précédent. Pour cela on introduit dans l'expansion des communications une variable supplémentaire de mémorisation notée « En ». Alors, la ré-écriture des actions de communications en respectant le protocole PCFB fait que l'on obtient :

```
[
   [ E_req & S_ack ] ; S_req+ ; E_ack- ; En- ;
   (  [ /S_ack ] ; S_req-  ),
   (  [ /E_req ] ; E_ack+  ) ; En+ ; LOOP
]
```

Les règles de productions correspondantes deviennent alors,

```
-- Requête de sortie
S_req+ <= E_req & S_ack & En+
S_req- <= /S_ack & En-

-- Acquittement d'entrée
E_ack+ <= /E_req & En-
E_ack- <= S_req+ ( & E_req )

--Règles pour la variable de
-- mémorisation En
En+ <= E_ack+ & S_req-
En- <= E_ack- & S_req+
```

L'interconnexion des différentes cellules à une sortie et à plusieurs entrées définies par ces règles de production donne le circuit asynchrone linéaire PCFB figure 5.8.

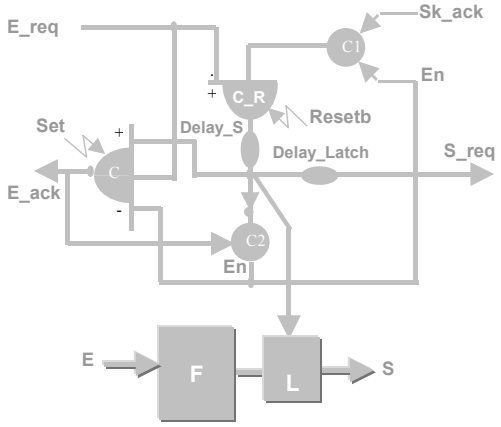


Fig. 5.8 : Micropipeline PCFB d'une structure linéaire

Les équations de dépendances (5.19) contraintes par le protocole PCFB et inhérentes à une structure pipeline linéaire se définissent comme suit :

```
cmd_latch_S = Delay_S( Muller2D1P_R( RESETb, Muller2(S_ack, En), E_req) )

S_req = Delay_latch( cmd_latch_S )

E_ack =  Muller2D1P1NB_S( SET, E_req, S_req,  En )
```

Equations 5.19

Le fonctionnement logique du circuit s'écrit :

```
Etat initial : Resetb = 0  => (S_req=0 et E_ack=1 et En=1 )

E_req=0 et S_ack=1 et En=1 => Rien

E_req=1 et S_ack=1 et En=1 => (S_req=1 et E_ack=0 et En=0)
                    - C'est la phase montante du protocole

S_ack=0  et En=0 => (S_req=0)
E_req=0 et En=0  => E_ack=1 (et En=1)
                  - C'est la phase descendante du protocole
```

5.2.2 Méthodologie pour les cas particuliers des « gardes » (choix) et « probes »

Nous expliquons la méthodologie suivie pour définir les parties de circuits asynchrones traduisant une garde ou un probe, au travers d'un exemple. Pour plus de clarté, nous réutilisons l'exemple du compteur donné en §4.6.7 et plus précisément le processus « main » du composant « counter ». On rappelle le code de ce processus :

```
…..
-------------------------
1er processus
-------------------------
process MAIN
port ( Go : in BD passive bit;
       CS : in BD active bit[3];
       NS : out BD active bit[3];
       S : out BD active bit[3])
variable css  :  bit[3];
[
  *[#Go =>
     [CS?css;
       @ [ css /= "1.1.1"[2] => NS!css , S!css
           css = "1.1.1"[2]  => Go? ; NS!"0.0.0"[2]
         ]
     ]
   ]
]

….
```

L'expansion des actions de communications du processus « main » pour la garde « G_0 » (ou « *reshuffling* ») donne :

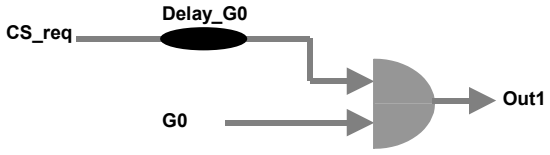
```
[Go_req ^ CS_req ^ G_0 ^ NS_ack ^ S_ack] ; NS_req+ , S_req+ , CS_ack- ;

[ /CS_req ^ /NS_ack ^ /S_ack] ; NS_req- , S_req- , CS_ack+ ;
```

Un signal de contrôle et une hypothèse temporelle doivent être associés à la garde « G0 » puisque celle-ci est calculée par le chemin de données qui est insensible au délais. On a donc :

```
CS_req ^ G0 => out1+                 CS_req  ^ G0 => out1+
/CS_req     => out1-        ⇒        /CS_req v /G0 => out1-
```

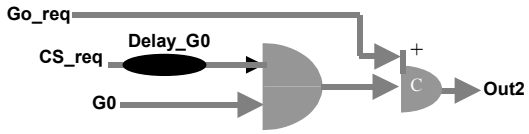
On en déduit le schéma suivant :



Dans l'étape suivante c'est le signal de requête du canal de probe « Go_req » qui est calculé. Ce dernier n'est impliqué que dans la phase montante du protocole seulement, car le canal de probe « Go » n'est pas lu dans la garde G0, et par conséquent il n'est pas acquitté au niveau de cette même garde. On introduit alors le signal « Out2 » qui se calcule comme suit :

```
{ out1 ^ Go_req => out2+
  /out1         => out2-
```

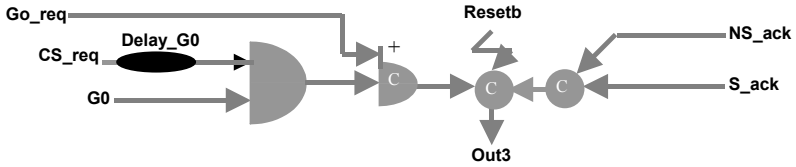
Ce qui donne une porte de Muller dissymétrique avec entrée sur la borne positive :



Finalement, les signaux d'acquittement « NS_ack » et « S_ack » sont calculés à leur tour comme suit :

```
Out2 ^ S_ack ^ NS_ack    => out3+
/Out2 ^ /S_ack ^ /NS_ack => out3-
```

On obtient alors le circuit de contrôle pour la garde G0 :



L'équation de dépendances (5.20) contrainte par le protocole WCHB et inhérente à la garde G0 d'un circuit asynchrone non linéaire (gardes) à deux sorties, avec « probe » est donnée comme suit :

Pour la garde G0

Commande de latch : **cmd_latch_NS_0** = **cmd_latch_S_0** =
max_delay_S(**Muller2_R**(Resetb,
Muller2D1P(**AND2**(G0, **delay_G0**(CS_req)), Go_req),
Muller2(NS_ack, S_ack)))

Requête de sortie du contrôleur: **NS_req_0** = **S_req_0** = **Delay_latch**(cmd_latch_S_0)

Acquittement d'entrée du contrôleur : **CS_ack_0** = **not**(cmd_latch_S_0)

Equations 5.20

Pour la seconde garde G1, la même procédure de synthèse est appliquée, à ceci près que dans cette garde le canal de probe Go est consommé ce qui rend son acquittement obligatoire. En conséquence la porte de Muller dissymétrique est remplacée par une porte de Muller symétrique (figure 5.9). Notons que seule la sortie de donnée NS est affectée dans cette garde.

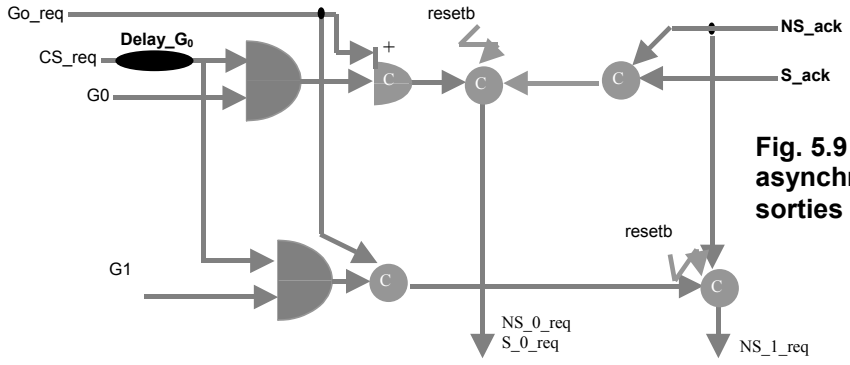


Fig. 5.9 : Contrôleur pour circuit asynchrone non linéaire à 2 sorties « gardé » et « probé »

Les équations de dépendances contraintes (5.21) par le protocole WCHB et inhérente à la garde G1 se présente alors comme suit :

Pour la garde G1
Commande de latch **:** **cmd_latch_NS_1** =
max_delay_S(**Muller2_R**(Resetb,
Muller2(**AND2**(G0, **delay_G1**(CS_req)), Go_req),
NS_ack))

Requête de sortie du contrôleur **:** **S_req_1** = **Delay_latch**(cmd_latch_S_1)

Acquittement d'entrée du contrôleur **:** **Go_ack_1 = CS_ack_1** = **not**(cmd_latch_S_1)

Equations 5.21

Une union exclusive est effectuée pour générer les équations de dépendances totales (5.22) inhérentes à chacune des deux sorties du pipeline non linéaire « gardé » et « probé » en WCHB :

Pour la sortie NS
Commande de latch **:** **cmd_latch_NS** = **OR2**[cmd_latch_NS_1, cmd_latch NS 1]

Requête de sortie du contrôleur **:** **NS req** = **OR2**[NS req 0, NS req 1]

Equations 5.22

Pour la sortie S
Commande de latch **:** **cmd_latch_S** = cmd_latch_NS_0

Requête de sortie du contrôleur **:** **S_req** = S req 0

Equations 5.23

Les équations de dépendances totales (5.24) des acquittements d'entrée sont :

CS_ack = **AND2**[CS_ack_0, CS_ack_1]
Go_ack = Go_ack_1

Equations 5.24

Finalement, une garde est réalisée grâce à une porte AND logique à deux entrées dont l'une représente la garde elle même alors que l'autre constitue l'ensemble des signaux de requêtes des canaux d'entrées dont dépend cette garde. Le probe non consommé impose l'utilisation d'une porte de Muller dissymétrique, alors que le probe consommé (ou lu) génère une porte de Muller symétrique.

D'une façon identique, on peut montrer qu'un probe comparé avec une valeur donnée (G_o#value), génère une porte AND en plus de la porte de Muller (c'est comme si l'on avait associé une garde à ce probe).

5.2.3 Génération des équations de dépendances contraintes par un protocole pour un circuit asynchrone (conforme DTL)

On reprend l'écriture d'une branche d'exécution d'un circuit asynchrone (cf. §4.6.6) :

[E_h?var_h ;
@[G_0 => E_j?var_j ;
@[G_{00} => E_m?var_m ;
@[G_{000} => E_n?var_n ; S_k? F_{Sk} (var_h , var_j , var_m , var_n) ; **BREAK**

En considérant G_t comme étant le booléen résultant du AND logique des gardes G_0, G_{00}, G_{000}, les sous-équations de dépendances non contraintes (requêtes de sorties et acquittements d'entrées) relatives au circuit non linéaire à gardes imbriquées sont données en équations 5.25 et 5.26 :

$$S_k^{Gt-req} = F_{dep}^{Gt-req}(nom\_composant, nom\_processus, (init=0), \prod_h E_h^{req}, (\prod_j E_j^{G0-req}, (G_0 = F_{garde}^{G0}(\prod_h E_h^{données}))), (\prod_m E_m^{G0-req}, (G_{00} = F_{garde}^{G00}(\prod_h E_h^{données}, \prod_j E_j^{données}))), ((\prod_n E_n^{G0-req}, G_{000} = F_{garde}^{G000}(\prod_h E_h^{données}, \prod_j E_j^{données}, \prod_m E_m^{données})) \prod_k S_k^{Gt-ack}, F_{Sk}^{Gt}(\prod_i E_j^{Gt-données}))$$

Equation 5.25

$$E_{h,j,m,n}^{Gt-acq} = F_{dep}^{Gt-acq}(nom\_composant, nom\_processus, (init=0), \prod_h E_h^{req}, (\prod_j E_j^{G0-req}, (G_0)), (\prod_m E_m^{G0-req}, G_{00}), ((\prod_n E_n^{G0-req}, G_{000})) \prod_k S_k^{Gt-ack})$$

Equation 5.26

5.2.3.1 Génération des équations de dépendances contraintes par le protocole WCHB pour un circuit asynchrone (compatible DTL)

En se basant sur les résultats de la méthodologie présentée en §5.2.1 pour la génération automatique des circuits asynchrones linéaires et non-linéaires (fourches et convergence) en WCHB, sur les résultats du §5.2.2 pour les cas particuliers des gardes et sur les sous-quations de dépendances 5.25 et 5.26 non contraintes, on obtient les sous-équations 5.27, 5.28 et 5.29 contraintes par le protocole WCHB :

Cmd_latch_Sk_000 = **Max_delay_Sk_000**(**Muller2_R**(Resetb,
MullerK($\prod_k Sk\_000\_ack$),

Muller2(**MullerN**($\prod_n En\_000\_req$),
AND2(***G_000***, **Max_delay_Gi**(

Muller2(**MullerM**($\prod_m Em\_00\_req$),
AND2(***G_00***, **Max_delay_Gi**(

Muller2(**MullerJ**($\prod_j Ej\_0\_req$),
AND2(***G_0***, **Max_delay_Gi**(
MullerH($\prod_h Eh\_req$))))))))))

Equation 5.27

Si une autre garde vient renforcer les conditions de garde sur cette branche d'exécution, on rajoute un module de « Muller2 » à deux entrées. Une des entrées de cette Muller2 est le résultat de la porte AND logique réalisant l'implémentation de la garde, alors que l'autre représente l'ensemble des signaux de requêtes d'entrée du degré d'imbrication inférieur. Dans un souci de clarté, un exemple de ce bloc de Muller2 est indiquée par la barre verticale noire.

En WCHB, le signal de requête de sortie est obtenu à partir de la commande de latch « cmd_latch_Sk_000 » moyennant un retard équivalent au temps de traversée d'un latch :

S_k_req_000 = Delay_Latch(cmd_latch_Sk_000) *Equation 5.28*

Les signaux d'acquittement des entrées E_h, E_j, E_m, E_n sont équivalents et sont également obtenus à partir de la commande de latch « cmd_latch_Sk_000 » moyennant un inverseur logique :

E_h_ack_000 = not(cmd_latch_Sk_000) *Equation 5.29*

Le total des requêtes de sortie en supposant que l'on a deux gardes au troisième niveau d'imbrication (G_{000} et G_{001}) est obtenu par union exclusive de la façon suivante :
On distingue 2 cas :

- Pour toutes les valeurs de k telles que les sorties Sk existent dans les 2 combinaisons de gardes (G_0 et G_{00} et G_{000}) et (G_0 et G_{00} et G_{001}) alors :

Sk_req = **OR**(Sk_000_req , Sk_0001_req) *Equation 5.30*

- Pour toutes les valeurs de k telles que les sorties Sk existent seulement dans l'une des 2 combinaisons de gardes (G_0 et $G_{00\ et}$ G_{000}) et (G_0 et $G_{00\ et}$ G_{001}) alors :

```
Soit : Sk_req = Sk_000_req
Soit : Sk_req = Sk_001_req
```

Equations 5.31

Le total des acquittements d'entrées avec la même supposition que pour le total des requêtes de sorties est déduit de la façon suivante :

```
Eh_ack = AND(Eh_000_ack , Eh_001_ack)
Ej_ack = AND(Ej_000_ack , Ej_001_ack)
Em_ack = AND(Em_000_ack , Em_001_ack)
```

Equations 5.32

Pour les En_ack, on distingue 2 cas :

- Pour toutes les valeurs de « n » telles que les entrées En existent dans les 2 combinaisons de gardes (G_0 et $G_{00\ et}$ G_{000}) et (G_0 et $G_{00\ et}$ G_{001}) alors :

```
En_ack = AND(En_000_ack , En_001_ack)
```

Equation 5.33

- Pour toutes les valeurs de « n » telles que les entrées En existent seulement dans l'une des 2 combinaisons de gardes (G_0 et $G_{00\ et}$ G_{000}) et (G_0 et $G_{00\ et}$ G_{001}) alors :

```
Soit : En_ack = En_01_ack
Soit : En_ack = En_02_ack
```

Equations 5.34

5.2.3.2 Génération des équations de dépendances contraintes par le protocole PCHB pour un circuit asynchrone (compatible DTL)

On procède de la même façon que pour le cas WCHB. On a alors :

```
Cmd_latch_Sk_000 = Max_delay_Sk_000(Muller3D1P1N_R(Resetb,
                   MullerK( ∏_h Sk_000_ack ),

                   Muller2( MullerN( ∏∏_n En_000_req ),
                         AND2( G_000, Max_delay_Gi (

                   Muller2 ( MullerM ( ∏_m Em_00_req ),
                         AND2( G_00, Max_delay_Gi (

                   Muller2( MullerJ( ∏_j Ej_0_req ),
                         AND2( G_0, Max_delay_Gi (
                   MullerH( ∏_h Eh_req ))))))))))
```

Equation 5.35

Les requêtes de sorties pour cette branche d'exécution sont obtenues de la même manière qu'en WCHB :

$$\mathbf{S_k\_req\_000} = \mathbf{Delay\_Latch}(\text{cmd\_latch\_Sk\_000}) \qquad \textit{Equation 5.36}$$

Les acquittements d'entrées pour cette branche d'exécution sont obtenus de la manière suivante :

$$\begin{aligned}\mathbf{Eh,j,m,n\_ack\_000} = \mathbf{Muller2D1NB}(&\text{cmd\_latch\_Sk\_000},\\ &\mathbf{Muller2}(\mathbf{MullerN}(\textstyle\prod_n En\_000\_req),\\ &\quad\mathbf{AND2}(\mathbf{G\_000}, \mathbf{Max\_delay\_Gi}(\\ &\mathbf{Muller2}(\mathbf{MullerM}(\textstyle\prod_m Em\_00\_req),\\ &\quad\mathbf{AND2}(\mathbf{G\_00}, \mathbf{Max\_delay\_Gi}(\\ &\mathbf{Muller2}(\mathbf{MullerJ}(\textstyle\prod_j Ej\_0\_req),\\ &\quad\mathbf{AND2}(\mathbf{G\_0}, \mathbf{Max\_delay\_Gi}(\\ &\mathbf{MullerH}(\textstyle\prod_h Eh\_req)))))))))\end{aligned} \qquad \textit{Equation 5.37}$$

Si une autre garde vient renforcer les conditions de garde sur cette branche d'exécution, on rajoute un module de « Muller2 » tel que l'indique la barre verticale noire. Pour les totaux des requêtes de sorties et des acquittements d'entrées on procède exactement de la même manière que pour le protocole WCHB.

5.2.3.3 Génération des équations de dépendances contraintes par le protocole PCFB pour un circuit asynchrone quelconque (compatible DTL)

On procède de la même façon que pour le cas WCHB. Ainsi :

$$\begin{aligned}\mathbf{Cmd\_latch\_Sk\_000} = \mathbf{Max\_delay\_Sk\_000}(&\mathbf{Muller2D1P\_R}(\text{Resetb},\\ &\mathbf{Muller2}(\text{En}_{\text{Sk\_000}}, \mathbf{MullerK}(\textstyle\prod_k Sk\_000\_ack)),\\ &\mathbf{Muller2}(\mathbf{MullerN}(\textstyle\prod_n En\_000\_req),\\ &\quad\mathbf{AND2}(\mathbf{G\_000}, \mathbf{Max\_delay\_Gi}(\\ &\mathbf{Muller2}(\mathbf{MullerM}(\textstyle\prod_m Em\_00\_req),\\ &\quad\mathbf{AND2}(\mathbf{G\_00}, \mathbf{Max\_delay\_Gi}(\\ &\mathbf{Muller2}(\mathbf{MullerJ}(\textstyle\prod_j Ej\_0\_req),\\ &\quad\mathbf{AND2}(\mathbf{G\_0}, \mathbf{Max\_delay\_Gi}(\\ &\mathbf{MullerH}(\textstyle\prod_h Eh\_req)))))))))\end{aligned} \qquad \textit{Equation 5.38}$$

Les requêtes de sorties pour cette branche d'exécution sont obtenues de la même manière qu'en WCHB :

$$\mathbf{S_k\_req\_000 = Delay\_Latch(cmd\_latch\_Sk\_000)} \qquad \textit{Equation 5.39}$$

Les acquittements d'entrées pour cette branche d'exécution sont obtenus de la manière suivante :

$$\begin{aligned}
&\mathbf{Eh,j,m,n_ack_000} = \mathbf{Muller2D1P1NB_S}(\mathrm{Set},\\
&\quad \mathbf{Muller2}(\mathbf{MullerN}(\textstyle\prod_n En_000_req),\\
&\qquad \mathbf{AND2}(\mathbf{G_000}, \mathbf{Max_delay_Gi}(\\
&\quad \mathbf{Muller2}(\mathbf{MullerM}(\textstyle\prod_m Em_00_req),\\
&\qquad \mathbf{AND2}(\mathbf{G_00}, \mathbf{Max_delay_Gi}(\\
&\quad \mathbf{Muller2}(\mathbf{MullerJ}(\textstyle\prod_j Ej_0_req),\\
&\qquad \mathbf{AND2}(\mathbf{G_0}, \mathbf{Max_delay_Gi}(\\
&\quad \mathbf{MullerH}(\textstyle\prod_h Eh_req))))))\\
&\quad \mathrm{Cmd_latch_Sk_000},\\
&\quad \mathrm{En_{Sk_000}}\\
&)
\end{aligned} \qquad \textit{Equation 5.40}$$

Et la variable « $\mathrm{En_{Sk\_000}}$ » qui permet le découplage totale entre les entrées de données et les sorties de données lors de la phase de remise à zéro, vaut :

$$\mathbf{En_{Sk\_000}} = \mathbf{Muller2}(\ \mathbf{not}(\mathrm{cmd\_latch\_Sk\_000})\ ,\ \mathrm{Eh,j,m,n\_ack\_000}\ ) \qquad \textit{Equation 5.41}$$

Pour les totaux des requêtes de sorties et des acquittements d'entrées procéder exactement de la même manière que pour le protocole WCHB.

5.3 Exemple du compteur en WCHB

On reprend l'exemple du compteur :

```
-------------------------
component counter
-------------------------
component counter
port ( Go : in BD passive bit;
       S : out BD active bit[3])
channel CS : BD bit[3];
channel NS : BD bit[3];
begin
-------------------------
1er processus
-------------------------
process MAIN
port ( Go : in BD passive bit;
       CS : in BD active bit[3];
       NS : out BD active bit[3];
       S : out BD active bit[3])
variable css  :  bit[3];
[
  *[#Go =>
     [CS?css;
       @ [ css /= "1.1.1"[2] => NS!css , S!css
           css = "1.1.1"[2]  => Go? ; NS!"0.0.0"[2
         ]
     ]
   ]
```

```
-------------------------
2nd processus
-------------------------
process INCREMENT
port (  CS : out BD passive bit[3];
        NS : in BD passive bit[3])
variable x  :  bit[3];

[CS!"0.0.1"[2];
  *[ NS?x ; CS!(x + 1)]
]
-------------------------
end counter;
-------------------------
```

Les équations de dépendances du contrôleur relatif au processus «main» explicitent la relation de causalité entre les ports de sorties NS et S et les ports d'entrées Go et CS. Dans la branche d'exécution correspondant au choix (CS /= 7[10]) on retrouve les sorties NS et S affectées concurremment. Par conséquent, l'application du modèle de l'équation de dépendances contrainte par le protocole WCHB, avec présence d'un canal de sonde (probe) non consommé donne:

```
NS_0_req = S_0_req = maxdelay_counter_main_S_G0( MULLER2_R( ResetB,
                     MULLER2(NS_ack,S_ack),
                     MULLER2DP(AND2(counter_main_G0,
                      max_delay_counter_main_G0_G1(CS_req)),
                     Go_req ) ) )
```

Dans la branche d'exécution correspondant au choix (CS = 7[10]), l'équation de dépendances donnant la requête du port de sortie NS se base sur l'application du modèle de l'équation de dépendances contrainte par le protocole WCHB. Le canal de sonde (probe) « Go » est dans ce cas consommé, ce qui transforme la porte de muller dissymétrique en porte de muller symétrique puisque dans ce cas le canal de probe « Go » doit être acquitté.

```
NS_1_req = maxdelay_counter_main_S_G0( MULLER2_R( ResetbB,
            NS_ack,
            MULLER2(AND2( G_1,
            maxdelay_counter_main_G0_G1 ( CS_req ) ),
            G0_req ) ) )
```

La sortie NS étant présente dans les deux gardes G0 et G1 on déduit la requête de sortie NS_req par l'union exclusive deux équations de dépendances ainsi générées :

```
NS_req = OR(NS_0_req, NS_1_req)
```

La sortie S n'est quand à elle affectée que dans le choix G0. De ce fait la requête de sortie S_req s'assimile à celle de la sortie S dans la garde G0.

```
S_req = S_0_req
```

L'acquittement de l'entrée CS est l'union exclusive des deux acquittements de cette même entrée dans les gardes G0 et G1. L'acquittement dans une garde étant obtenue en WCHB par une simple inversion logique du signal de requête de sortie relatif à cette garde, on a alors :

```
CS_0_ack = NOT[ NS_0_req ]
CS_1_ack = NOT[ NS_1_req ]
```

Et comme les signaux d'acquittement sont actifs au niveau bas alors l'union exclusive se traduit par une porte "et" logique:

```
CS_ack = AND2[ CS_0_ack,  CS_1_ack ]
```

Finalement le circuit obtenu est donné à la figure fig.5.10

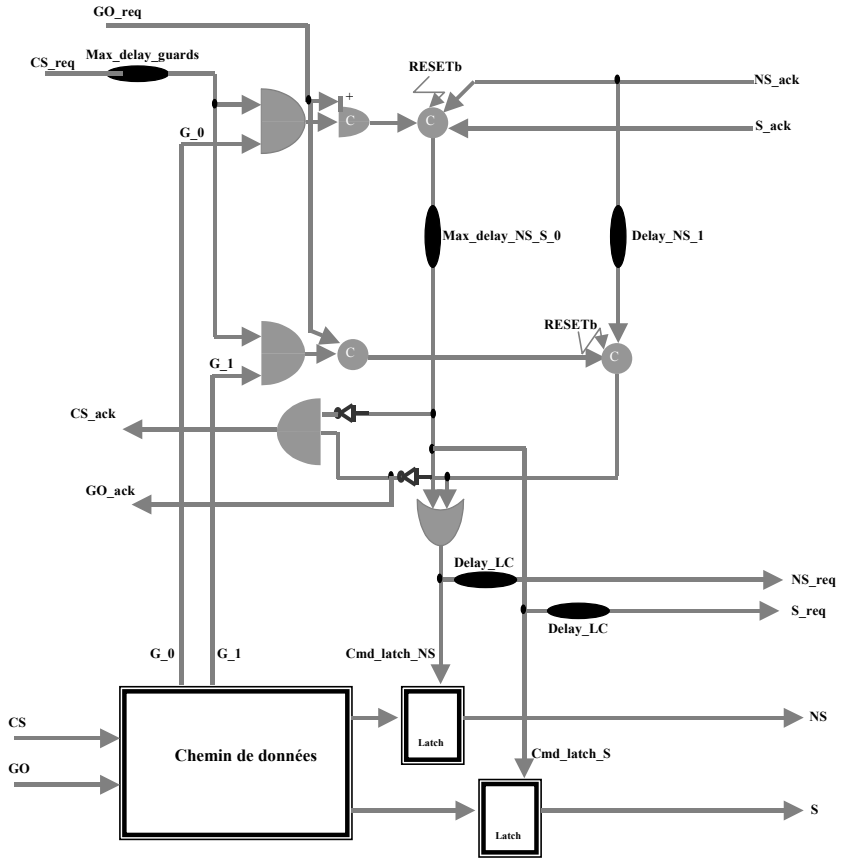


Fig. 5.10 : Contrôleur niveau portes relatif au processus « main » du composant « counter ».

Les équations de dépendances du contrôleur correspondant au processus « increment » sont ceux du buffer WCHB « initialisant ». L'initialisation de la sortie avant de déclencher la

boucle du processus nécessite l'ajout d'un bloc d'initialisation qui permet d'émettre la valeur d'initialisation une et une seule fois.

Les équations de dépendances pour le bloc de base définissent la commande du latch d'initialisation « cmd_latch_CS », le signal intermédiaire de requête « sig_CS_req » ainsi que le signal d'acquittement d'entrée « NS_ack ».

Les équations de dépendances pour le bloc d'initialisation définissent la commande du latch « cmd_latch_CS_I », le signal de requête « CS_req » du port de sortie, ainsi que le signal intermédiaire d'acquittement « sig_CS_ack ».

```
cmd_latch_CS_I = MULLER2_S( SET, sig_CS_req, CS_ack )

CS_req = delay_LC( and2(Resetb, cmd_latch_CS_I) )

sig_CS_ack = not(cmd latch CS I)
```

Le circuit obtenu est celui de la figure fig. 5.11

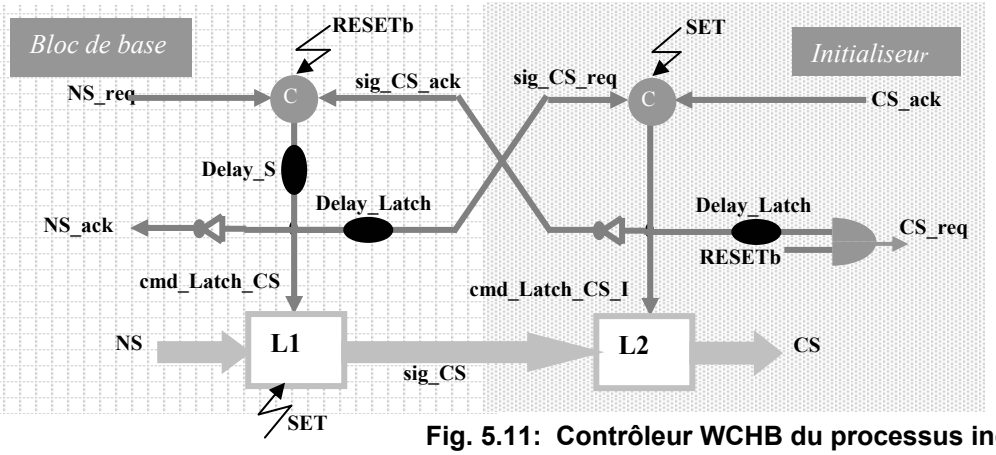


Fig. 5.11: Contrôleur WCHB du processus increment

La description VHDL du circuit micropipeline du compteur est donnée en annexe. Elle comprend un composant global « counter_mp » qui contient les deux composants «counter_main » et « counter_increment » correspondant aux deux processus « main » et « increment ». Chacun de ces deux composants contient à son tour deux composants, l'un décrivant le chemin de donnée, l'autre représentant le contrôleur.

5.4 Optimisation et projection technologique

L'optimisation logique est une étape cruciale de la synthèse en ce sens qu'elle a pour objectif de proposer une implémentation fonctionnellement équivalente mais optimisée selon des

critères imposés par le concepteur en vue d'obtenir un gain en surface, en rapidité ou en consommation.

La projection technologique se propose de projeter un circuit logique composé de portes de complexité variable en un circuit fonctionnellement équivalent composé de portes prédéfinies en cellules standard dont le dessin (*layout*) a été réalisé et optimisé pour constituer le niveau de granularité le plus fin au moment du routage. Il s'agit pour cela de décomposer le schéma initial de sorte qu'il ne soit plus constitué que de portes disponibles dans la librairie, ou au contraire de procéder à des factorisations si la librairie de cellules contient des sur-éléments de portes à implémenter.

Il semble au travers de ces deux définitions, qu'il soit plus simple de développer un ensemble de règles considérant simultanément l'optimisation et la projection technologique. Toutefois, l'application de ces règles aux circuits asynchrones micropipeline dont le contrôle est insensible aux délais, doit se faire tout en s'assurant que le circuit reste exempt d'aléa et que les fourches isochrones soient respectées.

5.4.1 Optimisation logique

Pendant longtemps les tableaux de Karnaugh et les lois de De Morgan ont été les principaux outils permettant d'obtenir une implémentation optimale de circuit. Aujourd'hui les arbres de décisions binaires (BDD) qui permettent de modéliser efficacement les fonctions logiques sont largement utilisés dans les logiciels commerciaux.

On peut également procéder à des tests d'équivalence fonctionnelle. Toutefois, tout test d'égalité fonctionnelle représente un problème NP-Complet qui ne permet pas de prévoir à l'avance le temps de résolution. Il apparaît donc que la phase d'optimisation nécessite des ressources de calculs importantes, c'est pourquoi les logiciels d'optimisation émettent des suppositions permettant de réduire les calculs nécessaires. Il reste cependant difficile de se prononcer sur le caractère optimal de la solution adoptée.

Le contrôle micropipeline étant insensibles aux délais nous adoptons les algorithmes d'optimisation utilisés pour les circuits synchrones [MIS 87] renforcés des contraintes d'insensibilité aux délais et du respect du caractère isochrone des fourches [DIN 03]. Il s'agit de parcourir l'ensemble des factorisations possibles pour le circuit en exécutant que celles qui réalisent un certain optimum selon des critères prédéfinis et pour lesquelles les contraintes d'aléas et de fourches isochrones sont respectées.

5.4.2 Décomposition

La méthode de synthèse adoptée fait que nous retrouvons souvent dans le circuit synthétisé des portes avec un nombre d'entrées (*fan-in*) élevé. Comme les portes de bibliothèque ne peuvent assurer une telle fonction, il est nécessaire de scinder la porte pour factoriser une partie des entrées ou modéliser la porte complexe avec des portes plus simples.

Contrairement au groupage de portes pour lequel il y a élimination des signaux, la scission d'une porte complexe en portes plus simples risque dans certains cas de rendre le circuit non-insensible aux délais.

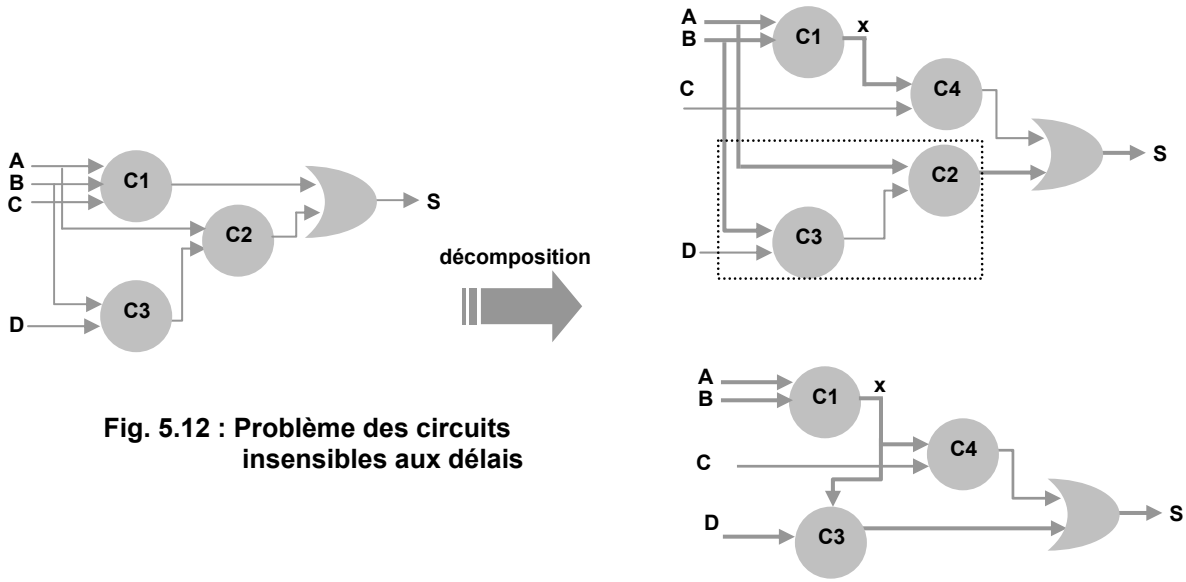


Fig. 5.12 : Problème des circuits insensibles aux délais

Dans la figure 5.12 le signal intermédiaire « x » issu de la décomposition peut être activé sans que la porte en aval le soit. Alors, si le chemin encadré en pointillé est également activé, la redescente du signal intermédiaire n'est pas vérifié. De ce fait, cette décomposition n'est pas insensible aux délais. En effet, dans la première décomposition proposée les chemins en gras sont ceux qui sont activés. Alors on voit bien que la porte de Muller C1 commute mais elle n'est pas acquittée. Dès lors nous ne maitrisons plus cette partie du circuit et ce dernier n'est plus insensible aux délais.

Dans la seconde décomposition proposée par contre, toutes les portes qui commutent sont bel et bien acquittées. Cette décomposition préserve donc l'insensibilité aux délais du circuit.

Burns [BURN 96] propose des règles de décomposition portant sur le signal intermédiaire qui permettent d'affirmer le caractère valide ou non d'une telle décomposition.

Dans notre méthodologie de synthèse nous effectuons la décomposition de portes de *fan-in* élevé dans le cas où la propriété d'insensibilité aux délais est garantie. En l'occurrence lorsque la décompostion ne créée pas de portes qui commutent sans participer de façon effective dans le changement de la sortie comme cela est le cas pour la porte C1 dans la première décomposition.

5.4.3 Projection technologique

Plusieurs techniques ([COR 97b][CHO 99][BEE 96][GIO 94]) ont été utilisées pour réaliser la projection technologique de structures matérielles très diverses qui constitue l'ultime étape avant le routage (placement définitif des éléments de circuits sur le masque). Comme l'objectif de cette étape de la synthèse est de décomposer les fonctions logiques complexes en portes élémentaires de la bibliothèque, nous sommes contraints de considérer de nouveau le problème d'optimisation car une décomposition n'est jamais unique. C'est pourquoi les logiciels de synthèse traitent souvent la projection technologique conjointement avec l'optimisation.

Le principe de la projection sur une bibliothèque de cellules standard (« semi-custom design ») est adopté car elle permet la réduction du temps de développement global. En effet, le niveau de granularité des cellules standard est plus gros que celui du transistor et le routage

s'en trouve simplifié. Cette projection est pour l'instant naïve et simple puisqu'elle relie un pour un les portes du circuits aux portes de la bibliothèque. Notons que mapper sur une bibliothèque de cellules standard présente l'inconvénient de l'impossibilité de réaliser l'optimisation au niveau transistor.

5.5 Conclusion

Nous avons exposé dans ce chapitre la synthèse de circuits asynchrones de type micropipeline en portes standard. A partir des équations de dépendances nous montrons la séparation entre le chemin de données et le contrôle. Le chemin de données est traduit en code VHDL synthétisable qui est soumis à une synthèse synchrone avec les outils du commerce. Cette synthèse du chemin de donnée permet également d'extraire les éléments de délais nécessaires à l'annotation du bloc de contrôle.

Nous présentons la méthodologie de synthèse des contrôleurs en considérant le protocole de communication ainsi que le modèle de circuit cible. Ainsi nous avons explicité l'implémentation des contrôleurs avec les protocoles WCHB, PCHB et PCFB, puis celle des opérateurs d'initialisation, de garde (sélection) et de sonde de canal (probe). Nous avons déduit alors les algorithmes de génération d'un circuit asynchrone micropipeline pour les trois protocoles de communication considérés.

L'optimisation et la projection technologique constituent deux étapes majeures dans le processus de synthèse. Ils doivent être menés de façon à garantir l'équivalence du circuit en s'assurant que le circuit reste insensible aux délais et en respectant le caractère isochrone des fourches.

CONCLUSION

L'objectif de cette thèse est de proposer une méthodologie et un outil de synthèse dédiés à la conception de circuits intégrés asynchrones micropipeline. Les méthodologies de conception asynchrone se distinguent d'une part par la méthode de spécification et d'autre part par la méthode de synthèse. Nous avons présenté dans la partie consacrée à l'état de l'art, une vue d'ensemble sur ces méthodes de spécification et de synthèse, permettant entre autres de faire ressortir leurs limitations et leurs points forts respectifs.

Le modèle de circuit que nous ciblons est donné après avoir présenté les techniques du pipeline asynchrone. Il s'agit du pipeline données groupées ou micropipeline dont les avantages, outre qu'il soit moins gros en surface, et qu'il consomme moins que les autres modèles, est de tirer partie de la synthèse synchrone. Ce modèle est appliqué en utilisant divers protocoles (séquentiel, WCHB, PCHB, PCFB) autant pour les pipelines linéaires que pour les pipelines non linéaires.

Le flot de conception « *top-down* » proposé, part d'une spécification basée sur un langage de haut niveau (CHP) décrivant le fonctionnement du circuit asynchrone à générer. Ce langage proposé dans sa forme originale par l'université Caltech, est étendu pour répondre à nos besoins en termes de simulation et de synthèse. Cette extension concerne principalement le typage des données et la modélisation structurelle et hiérarchique du circuit. Cette spécification est compilée pour obtenir une représentation intermédiaire basée sur une combinaison de réseaux de Petri et de graphes de flot de données. Outre le fait que cette représentation constitue une description formelle du circuit asynchrone, elle permet également de découpler le *front end* du *back end* de la procédure de synthèse.

La méthode de synthèse proposée est basée sur la spécification DTL qui permet d'identifier les éléments de mémorisation et de fixer un ensemble de règles qui doivent régir la description d'un circuit asynchrone. Ces règles ont pour objectif de limiter les éléments de mémorisation de sorte que toute spécification conforme aux règles DTL est synthétisable en circuits asynchrones. Toutefois, pour les spécifications non conformes, des transformations sont proposées et servent à extraire les éléments de mémorisation par décomposition. La génération de circuits micropipeline est basée sur les équations de dépendances dont la génération se fait à partir de la représentation intermédiaire selon des algorithmes explicités. Ces équations décrivent les sorties du circuit en fonction exclusive de ses entrées indépendamment du protocole de communication et de l'architecture cible.

L'implémentation micropipeline s'effectue en séparant le chemin de données du contrôle en distinguant les équations de dépendances des données de celles décrivant le contrôle. Les équations relatives au chemin de données, permettent d'exprimer les sorties en fonction des entrées. Ces équations sont traduites en un code VHDL RTL qui est synthétisé à l'aide des outils de synthèse commerciaux (Leonardo par exemple). Pour le contrôle, les équations de dépendances explicitent les signaux de requête et d'acquittement des circuits asynchrones en considérant un protocole de communication quatre phase donné. Une implémentation micropipeline des opérateurs d'initialisation, de garde et de probe est proposée. L'optimisation micropipeline concerne le contrôle et impose de garantir une construction optimisée du contrôleur sans aléa et de respecter les fourches isochrones. Le bloc de contrôle optimisé du circuit micropipeline est décrit en VHDL structurel et mappé sur une bibliothèque de cellules standard. En effet, les portes de Muller sont conçues avec ces dernières. Toutefois cela n'exclut pas que la projection puisse également se faire sur des technologies spécifiques dédiées à l'asynchrone.

L'objectif initial de cette thèse est atteint en ce sens que nous proposons une méthodologie considérant trois protocoles (WCHB, PCHB, et PCFB) pour la synthèse automatique de circuits asynchrones micropipeline. Un outil prototype limité au protocole WCHB a été développé. De ce fait, ce travail représente une contribution sur les méthodologies et l'automatisation de la conception des circuits asynchrones.

Perspectives

Une première perspective serait d'étendre l'outil prototype au traitement des protocoles PCHB et PCFB. Cela favorisera les études comparatives entre ces trois protocoles quatre phases appliqués à différents types de circuits. Sur le plan de la méthodologie, l'exploration du protocole deux phases pourrait s'avérer très fructueuse pour une amélioration des performances en terme de vitesse.

Par ailleurs, le flot de conception tel qu'il est proposé est extensible aussi bien en amont qu'en aval. En effet, il semble aisé d'ajouter du côté *front-end* un compilateur spécifique à un autre langage de spécification, alors que du côté *back-end* d'autres styles d'implémentation de circuits asynchrones peuvent être ciblés.

Enfin, ce type de circuit asynchrone est très performant (proche du synchrone) en terme de surface mais offre également des perspectives dans le domaine de la faible consommation et du faible rayonnement électromagnétique. Notamment, il est possible d'agir sur les retards inclus dans les contrôleurs afin de modifier le spectre rayonné des circuits micropipeline. Cette propriété intéressante pourrait être exploitée à terme par la mise en place d'un outil d'évaluation du rayonnement électromagnétique complémentaire à l'outil de synthèse et en relation avec les travaux menés par D.Panyasak [PAN 04].

REFERENCES

[ABR 01] A. Abrial, J. Bouvier, M. Renaudin, P. Senn and P. Vivet, "A New Contactless Smart Card IC using On-Chip Antenna and Asynchronous Microcontroller", Journal of Solid-State Circuits, Vol. 36, 2001, pp. 1101-1107.

[ACK 82] Ackerman W. B., « Data flow languages », IEEE Computer, vol. 15, no. 2, pp. 15-25, Feb. 1982.

[BAR 00] A. Bardsley and D. A. Edwards. The Balsa asynchronous circuit synthesis system. In Forum on Design Languages, September 2000.

[BAR 00b] A. Bardsley and D. A. Edwards. Synthetising an asynchronous DMA controller with Balsa. Journal of Systems Architecture, 46 : 1309-1319, 2000.

[BAR 97] BARDSLEY A., EDWARDS D., « Compiling the language Balsa to delay-nsensitive hardware », *Hardware description languages and their applications (CHDL),* C.D. Kloos and E. Cerny, editors, , pp. 89-91, April, 1997.

[BEE 92] P. Beerel and T.H.-Y. Meng. *Automatic gate-level synthesis of speed-independent circuits*. In Proc. International Conf. Computer-Aided Design (ICCAD), pages 58187.1EEE Computer Society Press, November 1992.

[BER 88] C. H. (Kees) van Berkel, Martin Rem, and Ronald W. J. J. Saeijs. VLSI Programming. *In Proc. of the 1988 IEEE Int. Conf. on Computer Design (ICCD)* : VLSI in Computers & Processors, 1988, 152--156.

[BER 91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, Frits Schalij Formal verification: The VLSI-programming language tangram and its translation into handshake circuits, in Proceedings of the conference on European design automation (EDAC), p.384-389, February 1991

[BER 92] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103-128, June 1992.

[BER 93] Kees Van Berkel, Handshake circuits: an asynchronous architecture for VLSI programming, *vol. 5 of Cambridge International Series On Parallel Computation*, Cambridge University Press, 1993.

[BER 94] Kees van Berkel, Ronan Burgess, Joep Kessels, Marly Roncken, Frits Schalij, Ad Peeters. Asynchronous Circuits for Low Power: A DCC Error Corrector, IEEE Design & Test, Volume 11 Issue 2, April 1994.

[BER 95] Kees van Berkel, Ronan Burgess, Joep Kessels, Marly Roncken, Frits Schalij, Ad Peeters and Rick van de Wiel. A single-rail re-implementation of a DCC error detector using a generic standard-cell library. I*n Asynchronous Design Methodologies*, pages 72-79. IEEE Computer Society Press, May 1995.

[BER 96] Kees van Berkel, Arjan Bink; Single-track handshaking signaling with application to micropipelines and handshake circuits; *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 122-133; 1996-03-01

[BER 98] Van Berkel C., Saeijs R., Compilation of communicating processes into delay-insensitive circuits, *in International Conference on Computer Design (ICCD)*, IEEE Computer Society Press, 1998

[BLU 00] BLUNNO I., LAVAGNO L., « Automated synthesis of micro-pipelilnes from behavioral Verilog HDL », *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Eilat, Israel, April 1-6, 2000.

[BRE 72] J.G. Bredeson and P.T. Hulina. Elimination of static and dynamic hazards for multiple input changes in combinational switching circuits. Information and Control, 20:114-224, 1972.

[BRE 75] J.G. Bredeson Synthesis of multiple input-change hazard-free combinationalswitching circuits without feedback. Int. J. Electronics, 39(6):615-624, 1975.

[BRU 91] Erik Brunvand, "Translating Concurrent Communicating Programs into Asynchronous Circuits," PhD. Thesis, Carnegie Mellon University, 1991

[BRZ 89] J. A. Brzozowski and J. C. Ebergen. *Recent developments in the design of asynchronous circuits*. Technical Report CS-89-18, University of Waterloo, Computer Science Department, 1989.

[BUR 88] Steven M. Burns. Automated compilation of concurrent programs into self-timed circuits. *Master's thesis*, California Institute of Technology, 1988.

[BUR 88c] Steven M. Burns and Alain J. Martin. Synthesis of Self-Timed circuits by program transformation. In G. J. Milne, editor, *The Fusion of Hardware Designa and Verification*, pages 99-116. Elsevier Science Publishers, 1988.

[BUR 96] Steven M. Burns, General condition of the decomposition of state holding elements. *In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, IEEE Computer Society Press, March 1996.

[CHU 93] CHU T.A., « CLASS: a CAD system for automatic synthesis and verification of asynchronous finite state machines », *Integration, The VLSI Journal*, Vol 15, N° 3, October 1993, pp. 263-289.

[CHU 87] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987

[COA 01] William S. Coates , Jon K. Lexau , Ian W. Jones , Scott M. Fairbanks , Ivan E. Sutherland, FLEETzero: An Asynchronous Switching Experiment, *Proceedings of the Seventh International Symposium on Asynchronous Circuits and Systems (ASYNC)*, p.173, March 11-14, 2001

[C0R 97] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A Yakovlev. Petrify : a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. IEICE Transactions on information and Systems, E80-D(3):315-325, March 1997.

[C0R 02] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A Yakovlev. Logic Synthesis of Asynchronous Controllers and Interfaces. *Springer-Verlag,* 2002

[DALL98] W. Dally and J. Poulton, *Digital Systems Engineering*. Cambridge University Press, 1998.

[DAV 78] DAVIS A.L., « The architecture and system method of DDM-1 : A recusively-structured data driven machine », *Proc. Fifth Annual Symposium on Computer Architecture*, 1978.

[DAV 93] DAVIS A., COATES B., STEVENS K., « The post office experiment : Designing a large asynchronous chip », *Proc 26th annu. Hawaii Int. Conf. on Systems Sciences*, vol. I, pp. 409-418, 1993.

[DAV 93b] DAVIS A., COATES B., STEVENS K., Automatic Synthesis of fast compact asynchronous control circuits. *In S. Furber and M. Edwards, editors Asynchronous Design Methodologies*, volume A-28 of IFIP Transactions, pages 193-207. Elsevier Science Publishers, 1993.

[DAV 97] DAVID R., ALLA H., Du Grafcet aux réseaux de Pétri, 2ème édition Hermès Sciences 504 pages - 5/31/97 – ISBN. 2866013255.

[DEA 92] Mark E. Dean. STRiP : A self Timed RISC Processor Architecture. *PhD thesis, Stanford University*, 1992.

[DIN 03] A. Dinh-duc Synthèse QDI des circuits asynchrones, *PhD Thesis* (in French), INP of Grenoble, 2003

[DJI76] E. W. Djikstra, A Discipline of Programming, Prentice Hall, Englewood Cliffs, N.J. 1976

[EBE 91] EBERGEN J., « A formal approach to designing delay-insensitive circuits », *Distributed Computing*, Vol. 5, N°. 3, pp. 107-119, July, 1991.

[EIC 65] EB Eichelberger. Hazard detection in combinational and sequential switching circuits. IBM Journal of Research and Devlopment, 9:90-99, March 1965

[ELH 95] EL HASSAN B., GUYOT A., RENAUDIN M. ET LEVERING V., « New self timed ring and their application to division and square root extraction », *ESSCIRC'95*, Lille, France, Sept. 1995,

[ELH 95a] EL HASSAN B., « Architecture VLSI Asynchrone utilisant la logique différentielle à précharge : Application aux opérateurs arithmétiques », Thèse de l'Institut National Polytechnique de Grenoble (INPG), spécialité Microélectronique, soutenue le 26 Septembre 1995 à Grenoble.

[END 94] Modelling and Simulation of Asynchronous Systems Using the LARD Hardware Description Language. In Proceedings of the 12th European Simulation Multiconference on Simulation – Manchester, Society for Computer Simulation International, pages 39-43, June 1994.

[FAN 96] Karl M. Fant and Scott A Brandt. NULL Conventionnal Logic : A complete and consistent logic for asynchronous digital circuit synthesis. *In International Conference on Application-specific Systems, Architectures, and Processors*, pages 261-273, 1996.

[FER 02] Marcos Ferretti and Peter A. Beerel. Single-track asynchronous pipeline templates using 1-of-N encoding. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1008-1015, March 2002.

[FUH 99] R.M. Fuhrer, S.M. Nowick, M. Theobald, N.K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines*. Technical Report CUCS-020-99*, Columbia University, July 1999.

[FUR 94] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. AMULET1: A micropipelined ARM. In *Proceedings IEEE Computer Conference (COMPCON)*, pages 476-485, March 1994.

[FUR 96] S. B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[FUR 96b] Stephen B. Furber and Paul Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247-253, June 1996.

[FUR 99] FURBER S., GARSIDE J., RIOCREUX P., TEMPLE S., DAY P., LIU J., PAVER N., « AMULET2e: an asynchronous embedded controller », *Proceedings of the IEEE*, vol. 87, n° 2, February 1999, p. 243-256.

[GAG 98] VAN GAGELDONK H., VAN BERKEL K., PEETERS A., BAUMANN D., GLOOR D., STEGMANN G., « An asynchronous low-power 80C51 microcontroller », *Proc. of the Int. Symp. on Advanced Research in Async. Circuits and Systems*, 1998, IEEE, p. 96-107.

[HAU 95] HAUCK S., « Asynchronous Design Methodologies : An Overview », *Proceeding of the IEEE*, Vol. 83, N° 1, pp. 69-93, January, 1995.

[HOA 78] Hoare C.A.R., Communicating Sequential Processes, Communications of the ACM 21, vol. 8, (April 1978) pp.666-677

[HUF 54] HUFFMAN, D. A., The Synthesis of Sequential Switching Circuits, Journal of the Franklin Institute, 257, nos. 3 and 4, March and Apr. 1954, 294--295.

[IEE 91] IEEE, *Proceedings of the IEEE, special section on another look at real-time programming*, Vol. 79, N° 9, pp. 1268-1336, September, 1991.

[JOS 93] MB Josephs, JT Udding. An overview of DI algebra. In TN Mudge, V Milutinovic, L Hunter, eds. Proc. 26th Annual Hawaii Int. Conf. on System Sciences I, pp. 329-338, IEEE CS Press, 1993.

[KEN 81] Ken Kennedy. A Survey of Data Flow Analysis Techniques. Program Flow Analysis: Theory and Applications, Steven S. Muchnick and Neil D. Jones, Prentice-Hall, 1981.

[KES 00] Joep Kessels, Torsten Kramer. Gerrit den Besten, Ad Peeters, Volker Timm : Applying Asynchronous Circuits in Contactless Smart Cards. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), pages 36-44. IEEE Computer Society Press, April 2000.

[KES 00b] Joep Kessels, Torten Kramer, Ad Peeters, and Volker Timm. DESCALE : a design experiment for a smart card application consuming low energy. In Rene van Leuken, Reinder Nouta, and Alexander de Graaf, editors, *European Low Power Initiative for Electronic System Design*, pages 247-262. Delft Institute of MicroElectronics and Submicron Technology, July 2000.

[KES 99] Joep Kessels and Paul Marston. Designing asynchronous standby circuits for a low-power pager. *Proceedings of the IEEE*, 87(2):257-267, February 1999.

[KIS 94] KISHINEVSKY M. A., KONDRATYEV A. K., TAUBIN A. R., VARSHAVSKY V. I., « Concurrent Hardware, The Theory and Practice of Self-Timed Design », *Wiley Series in Parallel Computing*, Chichester, 1994.

[KLE 87] KLEEMAN L., CANTONI A., « Metastable behavior in digital systems », *IEEE Design & Test of Computers*, December 1987, pp. 4-19.

[KON 02] Alex Kondratyev, Lawrence Neukolm, Oriol Roig, Alexander Taubin, and Karl Fant. Checkingdelay-insensitivity : 104 gates and beyond. *In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC),* pages 149-157, April 2002.

[KUD 96] KUDVA P., GOPALAKRISHNAN G., JACOBSON H., NOWICK S.M., « Synthesis of hazard-free customized CMOS complex-gate networks under multiple-input changes », *Proc. Of the 33rd ACM/IEEE Design Automation Conference*, pp.77-82, June, 1996.

[KUN 92] KUNG D.S., « Hazard-non-increasing gate-level optimization algorithms », *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 631-634, November 1992.

[LEE 91] LEE E. A., « Consistency in Dataflow Graphs », *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 2, April 1991.

[LIG 00] Michel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, and Alex Kondratyev. Asynchronous design using commercial HDL synthesis tools. *In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC),* pages 114-125. IEEE Computer Society Press, April 2000.

[LIN 94] LIN B., DEVADAS S., « Synthesis of hazard-free multi-level logic under multiple-input changes from binary decision diagrams », *Proc. Of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 542-549, November 1994.

[LIN 98] LINDEMANN C., *Performance Modelling with Deterministic and Stochastic Petri Nets*, John Wiley & Sons 1998.

[LINE 98] A. M. Lines. Pipelined Asynchronous circuits. *M. Sc. Thesis, California Institute of technology*, June 1995, revised 1998.

[MAR 03] Alain J. Martin, Mika Nyström, Karl Papadantonakis, Paul I. Penzes, Piyush Prakash, Catherine G. Wong, Jonathan Chang, Kevin S. Ko, Benjamin Lee, Elaine Ou, James Pugh, Eino-Ville Talvala, James T. Tong, Ahmet Tura. *The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller.* Accepted for

publication, 9th IEEE International Symposium on Asynchronous Systems & Circuits, 2003

[MAR 97] MARTIN A., LINES A., MANOHAR R., NYSTRÖM M., PENZES P., SOUTHWORTH R., CUMMINGS U., LEE T., « The design of an asynchronous MIPS R3000 microprocessor », *Proceedings of the 17th Conference on Advanced Research in VLSI*, 1997, p. 164-181.

[MAR 94] MARSHALL A., COATES B., SIEGEL P., « Designing an Asynchronous Communications Chip », *IEEE Design and Test of Computers*, Volume 11, Number 2, Summer 1994, pp. 8-21.

[MAR 93] MARTIN A., « Synthesis of Asynchronous VLSI Circuits », Caltech-CS-TR-93-28.

[MAR 90] Alain J. Martin, The limitations to delay-insensitivity in asynchronous circuits, Proceedings of the sixth MIT conference on Advanced research in VLSI, p.263-278, March 1990

[MAR 90b] Alain J. Martin Programming in VLSI : From communicating processes to delay insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1-64. Addison-Wesley, 1990

[MAR 89] MARTIN A. J., BURNS S. M., LEE T. K., BORKOVIC D., AND HAZEWINDUS P. J., « The Design of an Asynchronous Microprocessor », *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference in VLSI*, Charles L. Seitz editor, , pp. 351-373, MIT Press, 1989.

[MAR 86] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226-234, 1986.

[MAY 90] May, D., "Compiling occam into silicon", in Developments in Concurrency and Commu-nication, CAR Hoare, Ed., Addison-Wesley, pp. 87-129, 1990.

[MCA 92] MCAULLEY A.J., « Four state asynchronous architectures », *IEEE transactions on computers*, Volume 41, N° 2, pp 129 -142, Feb. 1992.

[MEN 89] T.H.-Y. Meng, R.W. Brodersen, and D.G. Messerschmitt, "Automatic synthesis of asynchronous circuits from highlevel specifications", IEEE Transactions on Computer- Aided Design, Vol. 8, No. 11, pp. 1185-1205, Nov. 1989.

[MEN 91] Teresa H.-Y. Meng, Synchronization Design for Digital Systems. *Kluwer Academic Publishers, 1991*

[MIL 65] R.E. Miller. Sequential Circuits and Machines: Volume 2 of Switching Theory . Wiley, 1965.

[MIS 87] Robert K. Brayton, Richard Rudell, A. Sangiovanni-Vincentelli, A.R. Wang , MIS : a multiple-level Logic Optimization System, proc. of IEEE Trans. On Computer-Aided Design, Vol CAD-6, N°6, November 1987.

[MOL 85] MOLNAR C. E., FANG T. P., ROSENBERG F. U., « Synthesis of delay-insensitive modules », *Chapel Hill conference on VLSI*, computer science press , pp 67 - 85, 1985.

[MUL 59] D.E. Muller and W.S. Bartky. A theory of asynchronous circuits. Annals of the Computation Laboratory of Harvard University. *Volume XXIX: Proceedings of an International Symposium on the Theory of Switching*, Part I, pages 204--243, 1959.

[MYE 92] Chris Myers and Teresa H.-Y. Meng. Synthesis of Timed Asynchronous Circuits. *In Proc. International Conf. Computer Design (ICCD),* pages 279-282. IEEE Computer Society Press, October 1992

[MYE 01] Chris Myers. Asynchronous Circuit Design. *John Wiley and Sons*, 2001

[NIE 94a] NIELSEN C.D., « Evaluation of function blocks for Asynchronous design », *EURODAC'94*, Grenoble, France, pp 454 - 459, Sept.1994.

[NIEL 99] Lars S. Nielsen and Jens Sparsø. Designing asynchronous circuits for low-power: An IFIR filter bank for a digital hearing aid. *Proceedings of the IEEE*, 87(2):268-281, February 1999.

[NOW 93] S. M. Nowick, "Automatic Synthesis of Burst-Mode Asynchronous Controllers", *PhD Thesis, Stanford University*, Department of Computer Science, March 1993.

[NOW 95] NOWICK S.M., DILL D.L., « Exact two level minimization of hazard-free logic with multiple input changes », *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14(8), pp. 986-997, August 1995.

[NYS 01] Mika Nyström Asynchronous Pulse Logic. PhD Thesis, California Institute of Technology, May 2001.

[OZDA02] Recep O. Ozdag , Peter A. Beerel, High-Speed QDI Asynchronous Pipelines, *Proceedings of the Eighth International Symposium on Asynchronus Circuits and Systems*, p.13, April 08-11, 2002

[OZDA02b] Recep O. Ozdag, Montek Singh, Peter A. Beerel, and Steven M. Nowick. "High-Speed Non-Linear Asynchronous Pipelines." *Proceedings of Design, Automation and Test in Europe (DATE-02),* Paris, France, March 2002.

[PAN 04] D. Panyasak, Conception de circuits asynchrones pour la faible émission électromagnétique, *PhD Thesis* (in French), INP of Grenoble, 2004

[PET 81] PETERSON, J. L. *Petri Net Theory and the Modeling of Systems.* Prentice- Hall, Englewood Cliffs, N.J (1981).

[PET 62] C. A. Petri : Kommunikation mit Automaten. *Phd Thesis, Bonn, Institut für Instrumetelle Mathematik*, 1962

[REN 00a] RENAUDIN M., VIVET P., GEOFFROY PH., « ASPRO : a toy demo », *4th AciD Workshop*, Grenoble, France, 31st – 1st February, 2000.

[REN 99] RENAUDIN M., VIVET P., ROBIN F., « A Design Frame Work for Asynchronous/ Synchronous Circuit Based on CHP to HDL Transaction », *International Symposium on Advanced Research in Asynchronous Circuits and Systems-ASYNC'98*, Barcelona, Spain, April 19-21, pp 135-144, 1999.

[REN 98] RENAUDIN M., VIVET P., ROBIN F., « ASPRO-216 : a standard-cell Q.D.I. 16-bit RISC asynchronous microprocessor », *Proc. of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998, IEEE, p. 22-31.

[REN 96] RENAUDIN M., ELHASSAN B., GUYOT A., « A New Asynchronous Pipeline Scheme : Application to the Design of a Self-Timed Ring Divider », *IEEE Journal of Solid-State Circuits*, Vol. 31, N° 7, pp. 1001-1013, July, 1996.

[REN 94] RENAUDIN M., ELHASSAN B., « The Design of Fast Asynchronous Adder Structures and Their Implementation Using D.C.V.S. Logic », *Proceedings ISCAS*, London, May, 1994.

[RIG 02] J.B. Rigaud, Spécification de bibliothèques pour la synthèse de circuits asynchrones, *PhD Thesis* (in French), INP of Grenoble, 2002

[ROB 97a] ROBIN F., « Etude d'architectures VLSI numériques parallèles et asynchrones pour la mise en œuvre de nouveaux algorithmes d'analyse et rendu d'images », Thèse de doctorat de l'ENST Paris, spécialité Electronique et Communications, soutenue à Grenoble le 27 octobre 1997.

[ROB 97b] ROBIN F., RENAUDIN M., PRIVAT G., VAN DEN BOSSCHE N., « Un réseau cellulaire VLSI fonctionnellement asynchrone pour le filtrage morphologique d'images », *Traitement du Signal*, numéro spécial, vol. 14, n° 6, Adéquation Algorithme Architecture, pp. 655-664, 1997.

[ROT 99] ROTEM S. ET AL, « RAPPID : An asynchronous instruction length decoder », *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Barcelona, April 18-22, pp. 60-70, 1999.

[ROS 85] L. Y. Rosenblum and A.V. Yakovlev. Signal graphs: from self-timed to timed ones, Proc. of the Int. Workshop on Timed Petri Nets, Torino, Italy, July 1985, IEEE Computer Society Press, NY, 1985, pp. 199-207.

[SCH 00] Schuster, S., Reohr, W., Cook, P., Heidel, D., Immediato, M., and Jenkins, K. Asynchronous interlocked pipelined CMOS circuits operating at 3.3-4.5 GHz, *in ISSCC 2000*, Page(s): 292-293

[SEN 92] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton and A.L. Sangiovanni-Vincentelli, SIS: A System for Sequential Circuit Synthesis, *Technical Report*, U. C., Berkeley, May 1992.

[SEI 70] Charles L. Seitz Asynchronous machines exhibiting concurrency, 1970. *Record of the project MAC Concurent Parallel Computation*

[SIE 93] SIEGEL P., DE MICHELLI G., DILL D., « Technology mapping for generalized fundamental-mode asynchronous designs », *Proc. Of the 30th ACM/IEEE Design Automation Conference*, pp. 61-67, June, 1993.

[SIN 01] Montek Singh, Steven M. Nowick: MOUSETRAP: Ultra-High-Speed Transition-Signaling Asynchronous Pipelines. *In Proc. International Conf. Computer Design (ICCD,* pages 9-17, November 2001

[SIN 00] Montek Singh , Steven M. Nowick, High-Throughput Asynchronous Pipelines for Fine-Grain Dynamic Datapaths, *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, p.198, April 02-06, 2000

[SIN 00b] Montek Singh and Steven M. Nowick, "Fine-Grain Pipelined Asynchronous Adders for High-Speed DSP Applications." In *Proceedings of IEEE Computer Society Annual Workshop on VLSI,* Orlando, FL, April 2000 (IEEE Computer Society Press, Los Alamitos, CA).

[SPA] SPARSO JENS, FURBER STEEVE, « Principles of Asynchronous Circuit Design : A Systems Perspective », *European Low-Power Initiative for Electronic System Design*, pp. 18-19, 2001.

[STE 99] STEVENS K., GINOSAR R., ROTEM S., « Relative Timings », *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Barcelona, April 18-22, pp. 208-218, 1999.

[SUTH01] I.E. Sutherland, S. Fairbanks. GasP - A Minimal FIFO Control. In Proc. *International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 46-53, IEEE Computer Society Press, 2001.

[SUT 89] SUTHERLAND I. E., « Micropipelines », *Communication of the ACM*, Volume 32, N°6, June 1989.

[TER 99] Hiroaki Terada, Souichi Miyata, and Makoto Iwata. DDMP's: Self-timed super-pipelined data-driven multimedia processors. *Proceedings of the IEEE*, 87(2):282-296, February 1999.

[THE 96] THEOBALD M., NOWICK S.M., WU T., « Espresso-HF : a heuristic hazard-free minimizer for two-level logic », *33rd ACM/IEEE Design Automation Conference*, pp. 71-76, June 1996.

[TIE 94a] TIERNO J. A., MARTIN A.J., BORKOVIC D. ET LEE T., « A 100 MIPS GaAs asynchronous microprocessor », *IEEE Design & Test of Computers*, Vol. 11, N°. 2, pp. 43-49, 1994.

[TIE 94c] TIERNO J., MARTIN A., BORKOVIC A., LEE T., « A 100-MIPS GaAs asynchronous microprocessor », *IEEE Design and Test of Computers*, vol. 11, n° 2, 1994, p. 43-49.

[TUGS02] Sunan Tugsinavisut, Peter A. Beerel: Control Circuit Templates for Asynchronous Bundled-Data Pipelines. *In Proc. Design, Automation and test in Europe (DATE)*, page 1098, March 2002.

[UDD 86] Jan Tijmen Udding: A Formal Model for Defining and Classifying Delay-Insensitive Circuits and Systems. Distributed Computing 1(4): 197-204 (1986)

[UNG 69] UNGER S.H., « Asynchronous sequential switching circuits », *Wiley interscience*, New York, NY, 1969.

[UNG 71] Stephen H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions on Computers*, 20(12):1437-1444, December 1971.

[VAN 92] VAN BERKEL K., « Beware the isochronic fork », *Integration, the VLSI journal*, N° 13, pp. 103-128, 1992

[VAN 93] VAN BERKEL K., « Handshake Circuits - An Asynchronous Architecture for VLSI Programming », *Cambridge University Press*, 1993.

[VAR 90] Victor I. Varshavsky, editor. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[WAK 94] John F. Wakerly, Digital Design: Principales and Practices, *Second Edition, Prentice,* 1994

[WES 67] Wesley A. Clark. Macromodular computer systems. *In AFIPS Conference Proceedings; 1967 Spring Joint Computer Conference*, volume 30, pages 335-336, Atlantic City, NJ, 1967, Academic Press.

[YAK 00] A. Yakovlev, L. Gomes and L. Lavagno. Hardware Design and Petri Nets. Kluwers Academic Publishers, 2000.

[YKM 94] Chantal Ykman-Couvreur, Bill Lin, and Hugo de Man. Assassin : A synthesis system for asynchronous control circuits. *Technical report*, IMEC, September 1994. User and Tutorial manual.

[YUN 92] Kenneth Y. Yun and David L. Dill. Automatic synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 576-580. IEEE Computer Society Press, November 1992.

[YUN 96] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996

[YUN98] Kenneth Y. Yun, Ayoob E. Dooply, Julio Arceo, Peter A. Beerel, and Vida Vakilotojar. The Design and Verification of a High-Performance Low-Control-Overhead Asynchronous Differential Equation Solve. *IEEE Transactions on VLSI Systems,* 6(4):643-655, December 1998

[WIL 91] WILLIAMS T.E. ET HOROWITZ M.A., « A Zero Overhead Self-Timed 160-ns 54-b CMOS Divider », *IEEE Journal of Solid State Circuits*, Vol. 26, N°. 11, pp. 1651 - 1661, Nov. 1991.

[ZHE 98] Zheng H., Specification and compilation of timed systems, *Master thesis*, University of Utah, 1998.

ANNEXES

A1.Eléments de syntaxe

A1.1 Le fichier source

Le fichier source d'un programme écrit en langage CHP est un simple fichier texte dont l'extension est par convention « .chp ».
Aspect d'un programme en CHP
Un programme écrit en langage CHP est une déclaration d'un ou de plusieurs composants. Chaque composant est constitué d'un ensemble de processus parallèles ou encore de composants dits « locaux » reliés les uns aux autres par des canaux. Les composants locaux sont destinés à être instanciés. Les processus renferment les instructions qui doivent être exécutées. Un programme CHP ressemblera donc à ceci :

```
COMPONENT buffer
PORT (e : IN DR ;
      s : OUT DR)
BEGIN
 PROCESS buffer_proc
 PORT (e : IN DR ;
       s : OUT DR)
 VARIABLE x : DR;
 BEGIN
  [ e?x ; s!x ; LOOP];
 END;
END;
```

Ce programme décrit un circuit tampon ou « buffer » qui lit en entrée une donnée sur le port « e » puis restitue cette donnée en sortie sur le port « s ». Il effectue cette action de façon bouclée.

A1.2 Conventions lexicales

Typologie

Le CHP est un langage insensible à la casse (case insensitive). Cela signifie qu'il n'y a pas de différence entre les caractères majuscules et minuscules. De ce fait, écrire BEGIN, BeGiN ou begin est strictement équivalent.

Commentaires

L'ajout d'un commentaire dans un programme CHP s'effectue après le signe « -- » et doit finir avant la nouvelle ligne. L'ajout de lignes de commentaires se fait en utilisant les balises « /* » et « */ ».

Caractères permis

[\t\r\n ;:\[\]\.\ »\'\(\)\-\+\ !\ ?*@,/#<>=_a-zA-Z0-9]

Identificateurs

Les identificateurs commencent par une lettre et sont suivis par une liste de lettres et/ou de chiffres. Comme le CHP étendu peut être compilé en VHDL, en C, ou autre, les mots clé du langage cible sont interdits. Ce point de contrôle est laissé au bon soin du concepteur pour le moment. Tous les objets CHP (composants, variables, canaux, …) sont repérés par un unique identificateur.

Mots clés CHP

ABS	ACTIVE	AND	BD	BEGIN	BIT
BOOLEAN	CHANNEL	COMPONENT	CONSTANT	DI	DR

END	FALSE	IN	INTEGER	MAP	MOD
MR	NAND	NATURAL	NOR	NOT	OR
OTHERS	OUT	PASSIVE	PORT	PROCESS	ROL
ROR	SDR	SIGNED	SKIP	SLA	SLL
VARIABLE	XNOR	XOR			

A1.3 Type de données

Un entier N est représenté en précision arbitraire par ″d_{L-1} , d_{L-2} , .., d_0″[B][L]. Cette écriture signifie vecteur de « L » digits représentés dans la base « B ». La longueur L et la base B sont des nombres naturels. Le type de données peut être signé ou non signé.
Type de données non signé
La valeur d'un entier non signé selon l'écriture définie précédemment est donnée par

$$N= \sum_{i=0}^{L-1} d_i B_i$$

Les types sont présentés selon l'ordre de complexité (valeur simple, vecteur, ou tableau). Deux types de base existent en CHP étendu, un type simple et un type vecteur.

Valeur simple
La valeur simple est équivalente à un tableau dont la longueur et la dimension valent 1.
MR[B] : type de Multi-Rails en base B représentant un nombre entre 0 et (B-1), codé en « 1 parmi n ».

Exemple :

```
VARIABLE  b : MR[B] ;     -- déclaration d'une variable b avec le type MR
          b := "x"[B];    -- affectation de b avec 0 <= x < B
```

Vecteur
Le type vecteur est équivalent à un tableau dont la dimension vaut 1. Sa syntaxe se traduit comme suit :
a[i] -- le ième élément de a
--« a » est un vecteur (contient plus d'un élément).
Pour accéder à un élément du vecteur « a », il faut préciser l'index de l'élément entre crochets après l'identificateur de la variable. « i » est une expression qui peut être évaluée statiquement et dont la valeur doit être comprise dans l'intervalle [0 , L-1], si L est la longueur du vecteur « a ». La variable a[i] est du même type que a, excepté que sa longueur est « 1 ».

MR[B][L] : Vecteur de L éléments de type Multi-Rails en base B. Une variable de ce type correspond à un nombre entre 0 et (B_L - 1).

Exemple :

```
VARIABLE  b : MR[B][3] ;     -- déclaration d'un vecteur de 3 éléments en base
B
          b := "x.y.z"[B];   -- affectation du vecteur b avec  0 <= x , y, z< B
```

D'autres types de données basés sur les types non-signés précédents ont été définis. Ils introduisent plus de facilité pour les concepteurs. Ils se déclinent comme suit :

Valeurs simples

DR	Type de double-rails - équivalent au type MR[2]
BIT	Nombre binaire (0 ou 1) - équivalent au type MR[2]
BOOLEAN	Valeur booléenne (vraie ou fausse) - équivalent au type MR[2]
SR	Type de mono-rail - équivalent à MR[1] Sert uniquement à synchroniser des processus communicants (donc il

n'est pas utilisé pour des variables)

Vecteurs
DR[L] Vecteur de L éléments de type double-rails - équivalent au type MR[2][L]
BIT[L] Vecteur de nombres binaires - équivalent au type MR[2][L]
BOOLEAN[L] Vecteur de booléens - équivalent au type MR[2][L]
NATURAL[Max] Nombres naturels appartenant à l'intervalle [0, Max] – équivalent au type MR[2][L] , L étant la partie entière supérieure de $Log_{2}[Max+1]$. Max est un entier naturel qui doit être spécifié lors de la déclaration de l'élément du type NATURAL.

Tableaux
Le type "tableau" peut être utilisé uniquement dans la déclaration de variables et de constantes. Les ports et les canaux ne peuvent être déclarés en type tableau pour des raisons technologiques (le protocole de communication implique 1 seul acquittement par canal de communication, or si un canal est déclaré en tableau et que ses différents éléments sont connectés à différents ports, quel port devra alors arbitrer l'acquittement et que feront les autres ports en parallèle ?). Tous les types de CHP sont représentables en tableau, excepté SR.

MR[B][L][D]
Dans tous les cas D est la longueur du tableau. C'est à dire que le tableau est constitué de D éléments de type MR[B][L] ou SMR[B][L] (tous les autres types étant des dérivés de ces deux types).
Un tableau ne peut pas être utilisé en tant que tel dans un programme CHP. Seuls les éléments d'un tableau peuvent être utilisés dans les expressions.

Pour accéder à un élément d'une variable tableau, la syntaxe est la même que celle utilisée pour accéder a un digit d'un type vecteur.

```
a[n]  -- le nième élément du tableau « a »
      -- « n » est une expression
```

Si n est une expression évaluable statiquement, n doit être compris dans l'intervalle [0, D-1].
Le type de la variable a[n] est soit MR[B][L] ou SMR[B][L] selon la déclaration de « a ».
Si les éléments du tableau sont des vecteurs alors, les digits des ces éléments peuvent être accédés comme suit :

```
a[i][n]      -- le ième élément de a[n]
a[i..j][n]   -- tous les éléments de a[i][n] à a[j][n]
```

Les autres types non signés de tableau dérivent du type de base MR[B][L][D] :
DR[L][D] Tableau de D vecteurs de L éléments de type double-rails - équivalent au type MR[2][L][D]
BIT[L][D] Tableau de D vecteurs de nombres binaires - équivalent au type MR[2][L][D]
BOOLEAN[L][D] Tableau de D vecteurs de booléens - équivalent au type MR[2][L]
NATURAL[Max][D] Tableau de D nombres naturels appartenant à l'intervalle [0, Max] – équivalent au type MR[2][L] , L étant la partie entière supérieure de $Log_{2}[Max+1]$. Max est un entier naturel qui doit être spécifié lors de la déclaration de l'élément du type NATURAL

Type de données signé
La valeur d'un entier signé selon l'écriture définie précédemment est donnée par

$$N = \begin{cases} \sum_{i=0}^{L-1} d_i B^i & \text{si } d_{L-1} < B/2 \\ (d_{L-1} - B)\ B^{L-1} \sum_{i=0}^{L-2} d_i B^i & \text{si } d_{L-1} >= B/2 \end{cases}$$

Cette écriture signifie que seul le digit de poids fort (le plus à gauche) porte le signe. Les autres digits se comportent comme des digits non signés.

En outre tous les types de données non signés présentés plus haut (sauf le type SR) possèdent leurs équivalents en type signé. Les deux types de base (1 type simple et 1 vecteur) sont donc :

Valeur simple
SMR[B] Type de Multi-Rails signé en base B. Si B est pair, ce type représente un nombre compris dans l'intervalle [–(B/2), (B/2)-1]. Si B est impair, il représente un nombre compris dans l'intervalle [-(B-1)/2 , (B-1)/2].

Exemple:
VARIABLE b : SMR[4]; -- déclaration d'une variable
b := "2"[4]; --affectation de b : b = 2 – 4 = -2

Vecteur
SMR[B][L] Vecteur de L éléments « Multi-Rail » signés en base B.

Exemple :
VARIABLE b : SMR[3][3] ; -- déclaration de b
b := "2.2.2"[3]; -- affectation : b=(-1)*3² + 2* 3^1 + 2* 3^0 = -1

Les autres types de données signés définis sont basés sur les types signés de base précédents. Ils introduisent plus de facilité pour les concepteurs. Ces types sont les suivants :

Valeur simple
SDR Type double-rails signé - équivalent à SMR[2]
Vecteur
SDR[L] Vecteur de L éléments double-rails signés - équivalent à SMR[2][L]
INTEGER[Max] Nombre de type SMR[2][L] compris dans l'intervalle [–Max+1),Max] - équivalent à SMR[2][L]
Tableau
SMR[B][L][D] Tableau de D vecteurs de L éléments en base B
SDR[L][D] Tableau de D vecteurs de L éléments de type double-rails - équivalent au type MR[2][L][D]
INTEGER[Max][D] Tableau de D nombres de type SMR[2][L] compris dans l'intervalle [–Max+1),Max] - équivalent à SMR[2][L]

A1.4 Conversion de types de données
Les données manipulées en CHP sont typées. Cela signifie que pour chaque donnée que l'on utilise il faut préciser le type de donnée, ce qui permet de connaître l'occupation mémoire de la donnée ainsi que sa représentation. Les deux opérandes d'une opération

logique, arithmétique ou encore d'une action de communication doivent être du même type. L'affectation est la seule opération qui permet de convertir les types de données et les rendre compatibles. Le type de données de la partie droite de l'affectation est toujours converti au type de données de la partie gauche. A titre d'illustration si une variable « x » est déclarée avec le type DR et une variable « y » avec le type MR[3] alors l'affectation x :=y appelle automatiquement la procédure de conversion de type et « y » est converti en DR.

A1.5 Opérateurs

Toutes les opérations impliquent une vérification de type, mais tous les types CHP s'expriment en fonction des 2 types de base MR[B][L] (type non signé) et SMR[B][L] (type signé. Une conversion de type est effectuée seulement lorsque les types des opérandes sont structurellement différents. A titre d'exemple les opérations impliquant les types BOOLEAN, BIT et DR ne nécessitent pas de conversion puisque ils sont tous équivalent au type MR[2][1].

On note les opérateurs par ordre de priorité décroissante de haut en bas et de droite à gauche.

()
- (unaire) not abs
*** / mod + - and nand or nor xor xnor sll sla srl sra rotl rotr**
= /= < <= > >=
 # ! ?
:=

Opérateur de conversion de signe

Une conversion de signe ne se fait pas automatiquement mais explicitement. Le passage d'un type non signé à signé s'effectue par l'opérateur SIGNED(< expression non signée>). A l'inverse le passage d'un type signé à non signé s'obtient par l'opérateur UNSIGNED(<expression non signée>). Il est noter que l'opération de conversion de signe ne change pas la structure du type (base et longueur) mais uniquement le signe. Par ailleurs, la valeur arithmétique n'est conservée que si elle appartient à l'intervalle du type de destination (< $(B^L+1)/2$ si la conversion est vers signé, >0 si la conversion est vers non signée). A titre d'exemple un MR[3][3] qui vaut ″2.2.2″[3]= ″26″[10] s'il est converti en signé vaudra ″2.2.2″[3]= ″-1″[10].

Opérateur d'affectation / conversion

« := ». L'opérateur d'affectation / conversion peut être utilisé avec tous les types du CHP, le type « SR » mis à part. Dans l'écriture « Var := Expression » l'opérande Var doit obligatoirement être une variable (pas de constante, canal, port ou expression). L'affectation est utilisée entre différents types à condition qu'ils soient de signe identique. Dans ce cas on doit obligatoirement recourir à l'opérateur de conversion de signe. Cet opérateur est le seul qui permet la conversion de type.

Opérateurs de comparaison

« =, /=, <, >, <=, >= ». Si les deux opérandes comparés sont des variables, alors ils doivent être du même type (même signe, même base, même longueur).Si l'un des deux opérandes comparés est une expression constante et l'autre une variable, alors le type de l'expression est converti pour être adapté au type de la variable.

Opérateurs logiques

« not, nand, and, nor, or, xnor, xor ». Ils sont identiques aux opérateurs logiques classiques définis pour les valeurs booléennes à ceci près qu'en CHP ils sont étendus aux autres types.

Dans une base donnée, le résultat d'une opération logique est le même que celui d'une opération booléenne effectuée bit par bit avec la représentation binaire de chaque digit de la valeur. Les deux opérandes doivent cependant être du même type (base, longueur, signe). Pour effectuer une opération logique entre opérandes de différents types une affectation doit être explicitée.

Opérateurs arithmétiques
« +, -, *, mod, abs ». La restriction pour les opérateurs arithmétiques est que les deux opérandes doivent être du même type (base, longueur, signe). Dans le cas contraire, une affectation / conversion est nécessaire. Les deux opérateurs de négation unaire « - » et abs s'appliquent uniquement pour le type signé.

Opérateurs de décalage
« sll, sla, srl, sra, rotl, rotr ». L'opérande qui exprime le nombre digits à décaler doit être non signé. L'opérande sur lequel s'effectue le décalage est de type quelconque, SR exclu.

Opérateur de séquentialité
« ; ». Sémantiquement, l'écriture « inst1 ; inst2 » signifie que l'instruction 2 n'est exécutée que lorsque l'instruction « inst1 » a été elle même exécutée. Dès la fin de l'exécution de l'instruction « inst2 » on passe à la suite du programme.

Opérateur de parallélisme
« , ». Sémantiquement, l'écriture « inst1 , inst2 » signifie que l'instruction « inst1 » s'exécute parallèlement que l'instruction « inst2 ». On ne passe à la suite du programme que lorsque l'exécution des 2 instructions est terminée.

Opérateur d'actions de communication
Les canaux et les ports sont utilisés pour communiquer entre processus et composants. Pour qu'un canal ou un port de type « td » réceptionne ou émette une donnée à travers une variable, il faut que cette variable soit également de type « td » excepté pour le signe. Aucune conversion n'est effectuée à ce niveau vu que seul l'opérateur d'affectation le permet. Les opérateurs assurant les actions de communications sont au nombre de 3 : le probe « # », l'opérateur de réception de données « ? », et l'opérateur d'émission de données « ! ».
Le probe « # » : Le probe permet de vérifier si une donnée est prête sur un port. Cet opérateur retourne un booléen et ne peut être utilisé que pour les ports passifs (un port d'entrée est passif par défaut, un port de sortie doit par contre être explicité passif avant d'être probé). En fait, il n'est utilisé que dans les expressions de gardes car seules les gardes retournent un booléen. L'écriture « **#**port_name » teste si une donnée est présente dans le canal relié au port port_name . Si le port « probé » est un port d'entrée, on peut également comparer la valeur de Cette donnée à une valeur explicite grâce à l'écriture « **#**port_name **=** expression ».
Réception de données « ? » :
La syntaxe « <nom_de_port> ? ; » permet de recevoir une donnée sans la mémoriser et la syntaxe « <nom_de_port> ? <nom_de_variable> ; » permet de lire une donnée en la stockant dans une variable. Seuls les ports d'entrée peuvent être utilisés à gauche de l'opérateur « ? » et seules les variables sont utilisées pour recevoir les données émises par ces ports.
Emission de données « ! » :
La syntaxe « <nom_de_port> ! ; » permet d'émettre un acquittement et n'est permise qu'avec le type SR. La syntaxe « <nom_de_port> ! <expression> ; » permet d'émettre une donnée contenue dans une expression à travers un port. Seuls les ports de sortie peuvent être utilisés à gauche de l'opérateur « ! » et toutes les expressions peuvent être émises les données émises à travers ces ports. Les types des opérandes doivent être strictement identiques.

A.2 Correspondance CHP→ VHDL-µPipeline jusqu'au niveau des instructions

A.2.1 Composant

Tout composant CHP devient le composant « top » qui décrit le circuit micropipeline. Ce composant « top » est un couple entité/architecture où :

- L'entité VHDL porte le même nom que le composant CHP suivi du suffixe « _MP ».
- L'architecture VHDL porte le même nom que le composant CHP précédé du préfixe « Arch_ » et suivi du suffixe « _MP ».

A.2.2 Port

Tout port de composant CHP de type MR[B][L], à l'exception du type SR, est traduit par trois (03) ports associés à l'entité VHDL composant « top » décrivant le circuit µpipeline :

- Le premier port représente la donnée,
  - il possède le même nom que le port du composant CHP, avec le suffixe « _data »,
  - il est de type std_ulogic_vector(((L*[Log2(B)]$_{sup}$ – 1) ... 0).
- Le second port représente la requête,
  - il possède le même nom que le port du composant CHP, avec le suffixe « _req »,
  - il est de type std_ulogic
- Le troisième port représente l'acquittement,
  - il possède le même nom que le port du composant CHP, avec le suffixe « _ack »,
  - il est de type std_ulogic

On ajoute à la liste des ports un port de remise à zéro ou d'initialisation :
- de nom « resetb », qui signifie remise à zéro à niveau bas,
- de type std_ulogic

Au type SR qui ne transporte pas de données, on fait juste correspondre un port de requête et un port d'acquittement (pas de port de données).

A.2.3 Constante

Toute constante CHP de type MR[B][L] est déclarée dans la partie déclarative de l'architecture VHDL.

- avec le même nom,
- avec le type std_ulogic_vector(((L*[Log2(B)]$_{sup}$ – 1) ... 0).
- éventuellement avec une valeur initiale sous la forme d'un littéral VHDL ou d'une expression manipulant des globaux et/ou des littéraux.

A.2.4 Canal

Tout canal CHP de type MR[B][L] déclaré dans le composant se traduit par trois (03) signaux de l'architecture VHDL:

- Le premier signal représente la donnée du canal,

- il possède le même nom que le canal du composant CHP, avec le suffixe « _data »,
- il est de type std_ulogic_vector(((L*[Log2(B)]$_{sup}$ – 1) ... 0).

- Le second signal représente la requête associée au canal,
  - il possède le même nom que le canal du composant CHP, avec le suffixe « _req »,
  - il est de type std_ulogic

- Le troisième signal représente l'acquittement associé au canal,
  - il possède le même nom que le canal du composant CHP, avec le suffixe « _ack »,
  - il est de type std_ulogic

A.2.5 Composant local

Tout composant local déclaré en CHP étendu devient un composant local dans l'architecture VHDL. Le composant devra avoir été compilé au préalable dans la bibliothèque de travail.

A.2.6 Instance

Toute déclaration d'instance CHP est convertie en déclaration d'instance VHDL aux signaux de requête/acquittement près, la syntaxe étant identique en CHP et en VHDL.

- La correspondance <port de l'entité instanciée / port de l'entité ou signal de l'architecture> doit être systématiquement explicitée.

- Il est clair que l'on doit avoir accès à l'architecture de ce composant local instancié pour la génération des deux sous-composants :

  - « contrôle » dans lequel on explicitera les requêtes de sortie et les acquittements d'entrée.

  - et « data-path » dans lequel on explicitera le calcul combinatoire.

A.2.7 Processus

Tout processus CHP devient un sous-composant du composant « top » qui décrit le circuit VHDL. Ce sous-composant est un couple entité/architecture où :

- le nom de l'entité est le même que le nom du composant CHP concaténé du nom du processus CHP : « nomducomposant_nomduprocessus »,
- le nom de l'architecture est le même que celui de l'entité auquel on ajoute le préfixe « Arch_ »

A.2.8 Port du processus

Tout port de processus CHP est traduit par trois (03) ports associés à l'entité VHDL du sous-composant du composant « top » décrivant le circuit micropipeline :

- Le premier port représente la donnée,
  - il possède le même nom que le port du processus CHP, avec le suffixe « _data »,
  - il est de type std_ulogic_vector(((L*[Log2(B)]$_{sup}$ – 1) ... 0).

- Le second port représente la requête,

  - il possède le même nom que le port du processus CHP, avec le suffixe « _req »,
  - il est de type std_ulogic

- Le troisième port représente l'acquittement,
  - il possède le même nom que le port du processus CHP, avec le suffixe « _ack »,
  - il est de type std_ulogic

On ajoute à la liste des ports un port de remise à zéro ou d'initialisation :
- de nom « resetb », qui signifie remise à zéro à niveau bas,
- de type std_ulogic

Par ailleurs, le port de donnée est également associé au sous-composant de « données » du sous-composant représantant le processus CHP. Ce sous-composant de donnée est un couple entité/architecture dont :
- L'entité porte le même nom que celui du composant CHP, concaténé du nom du processus CHP auquel il est associé, concaténé du suffixe « _DP »
- L'architecture porte le même nom que celui du composant CHP avec le préfixe « Arch_ », concaténé du nom du processus CHP auquel il est associé, concaténé du suffixe « _DP »

Egalement, les ports de requête et d'acquittement sont également associés au sous-composant de « contrôle » du sous-composant représantant le processus CHP. Ce sous-composant de contrôle est un couple entité/architecture dont :
- L'entité porte le même nom que celui du composant CHP, concaténé du nom du processus CHP auquel il est associé, concaténé du suffixe « _CONTROL »
- L'architecture porte le même nom que celui du composant CHP avec le préfixe « Arch_ », concaténé du nom du processus CHP auquel il est associé, concaténé du suffixe « _CONTROL »

A.2.9 Variable

Les variables CHP sont toujours locales aux processus CHP étendu. Ces variables servent en CHP étendu à des calculs intermédiaires ou à l'adaptation de types (cast). En VHDL :

- elles n'apparaissent plus en ce sens où les calculs se font directement sur les ports,
- les éventuels adaptations de types (cast) se font par la concaténation de bits nuls en tant que préfixe,
- le délai induit par les calculs intermédiaires que font ces variables est pris en compte dans la partie contrôle.

A.3 Modèle général de code VHDL synthétisable généré pour le chemin de données

```
-- Bibliothèque IEEE standard
library IEEE ;
use IEEE.std_logic_1164.all ;

-- Bibliothèque propre au composant à synthétiser
-- nom du composant à partir de « équation 4.35 »
library LIB_ComponentName ;
use LIB_ComponentName.LATCH.all;

-- le nom du processus en cours de traitement à partir de « équation 5.3 »
ENTITY COMP_P1_DP IS – Processus P1 du composant COMP
PORT(
   RESETb : IN STD_ULOGIC ;

  -- Entrées de données à partir de « équation 5.3 »
  P1_E1_data, P1_E2_data, …, P1_En_data : IN STD_ULOGIC_VECTOR(
                                          (L*log2(B))-1 downto 0);

  -- Sorties de données à partir de « équation 5.3 »
  P1_S1_data, P1_S2_data, …, P1_Sn_data : OUT STD_ULOGIC_VECTOR(
                                          (L*log2(B))-1 downto 0);

  -- gardes à partir de « équation 5.3 »
  COMP_P1_G0, COMP_P1_G1, …, COMP_P1_Gn : OUT STD_ULOGIC ;

 -- commandes des latchs de sorties de données à partir de « équation 5.3 »
 CMD_LATCH_COMP_P1_S1, CMD_LATCH_COMP_P1_S2, …,
                          CMD_LATCH_COMP_P1_Sn: IN STD_ULOGIC
  );
END COMP_P1_DP ;

ARCHITECTURE ARCH_COMP_P1_DP  OF COMP_P1_DP  IS

-- Déclaration des Signaux intermédiaires des gardes (utilisés en entrées)
SIGNAL s_COMP_P1_G0, s_COMP_P1_G1, …, s_COMP_P1_Gn :
STD_ULOGIC ;

-- signaux de sortie avant latch
SIGNAL s_P1_S1_BFL, s_P1_S2_BFL, …, s_P1_Sn_BFL :
            STD_ULOGIC_VECTOR((L*log2(B))-1 downto 0);

-- En supposant que seul la sortie « Sn » est initialisée on déclare en
-- plus le signal après latch (AFL).
-- sortie initialisés à partir de « équation 5.3 »
SIGNAL s_P1_Sn_AFL : STD_ULOGIC_VECTOR((L*log2(B))-1 downto 0);
BEGIN

-- Calcul des valeurs des gardes
          s_COMP_P1_G0 <= '1' WHEN ( CONDITION_0 ) ELSE
                          '0' ;
          s_COMP_P1_G1 <= '1' WHEN ( CONDITION_1 ) ELSE
                          '0' ;
          …
          s_COMP_P1_Gn <= '1' WHEN ( CONDITION_n ) ELSE
                          '0' ;

-- Calcul des sorties avant latchs à partir de « équation 5.3 »
-- Si P1_S1_data est de type « STD_ULOGIC_VECTOR(1 downto 0)»
          s_P1_S1_BFL <= "01" WHEN (s_COMP_P1_G0 = '1') ELSE
                         "10" WHEN (s_COMP_P1_G1 = '1') ELSE
                         …
                         "00" WHEN (s_COMP_P1_Gn = '1') ELSE
                         '-' ;

-- Si P1_S2_data est de type « STD_ULOGIC»
s_P1_S2_BFL <= '1' WHEN (s_COMP_P1_G0 = '1') ELSE
               '0' WHEN (s_COMP_P1_G1 = '1') ELSE
               …
               '0' WHEN (s_COMP_P1_Gn = '1') ELSE
               '-' ;
…
-- Si P1_Sn_data est de type « STD_ULOGIC_VECTOR(2 downto 0)»
s_P1_Sn_BFL <= "001" WHEN (s_COMP_P1_G0 = '1') ELSE
               "010" WHEN (s_COMP_P1_G1 = '1') ELSE
               …
               "000" WHEN (s_COMP_P1_Gn = '1') ELSE
               '-' ;

-- Affectation des gardes
COMP_P1_G0 <= s_COMP_P1_G0 ;
COMP_P1_G1 <= s_COMP_P1_G1 ;
…
COMP_P1_Gn <= s_COMP_P1_Gn ;

-- Calcul des sorties après latchs
-- En supposant que P1_S1_data est de type
« STD_ULOGIC_VECTOR(1 -- downto 0)»

Instance_P1_S1_0 : LATCH_RESETb PORT MAP( RESETb,
P1_S1_data(0), cmd_latch_COMP_P1_S1, s_P1_S1_BFL(0) );

Instance_P1_S1_1 : LATCH_RESETb PORT MAP( RESETb,
       P1_S1_data(1), cmd_latch_COMP_P1_S1, s_P1_S1_BFL(1) );


-- En supposant que P1_S2_data est de type « STD_ULOGIC»
Instance_P1_S2 : LATCH_RESETb PORT MAP( RESETb, P1_S2_data,
                 cmd_latch_COMP_P1_S2, s_P1_S2_BFL );
            …

-- En supposant P1_Sn_data est de type « STD_ULOGIC_VECTOR(2
downto 0)»
-- Et que P1_Sn_data est initialisée dans le processus P1 à la  valeur
« 001 »
Instance_P1_Sn_AFL_0 : LATCH_SET PORT MAP( SET, P1_Sn_AFL(0),
                cmd_latch_COMP_P1_Sn, s_P1_Sn_BFL(0) );
Instance_P1_Sn_0 : LATCH PORT MAP( P1_Sn_data(0),
               cmd_latch_COMP_P1_Sn_I, s_P1_Sn_AFL(0) );

Instance_P1_Sn_AFL_1 : LATCH_RESETb PORT MAP( RESETb,
          P1_Sn_AFL(1),  cmd_latch_COMP_P1_Sn, s_P1_Sn_BFL(1) );
Instance_P1_Sn_1 : LATCH PORT MAP( P1_Sn_data(1),
           cmd_latch_COMP_P1_Sn_I, s_P1_Sn_AFL(1) );

Instance_P1_Sn_AFL_2 : LATCH_RESETb PORT MAP( RESETb,
             P1_Sn_AFL(2),  cmd_latch_COMP_P1_Sn, s_P1_Sn_BFL(2) );
Instance_P1_Sn_2 : LATCH PORT MAP( P1_Sn_data(2),
            cmd_latch_COMP_P1_Sn_I, s_P1_Sn_AFL(2) );
END ;
```

A4. Code VHDL du circuit micropipeline compteur

A4.1--Composant global “counter” contenant 2 composants “main” et “increment”

```
-- Librairie IEEE standard
library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_unsigned.all;

-- Librairie propre
library LIB_COUNTER ;
use LIB_COUNTER.package_delay.all ;
use LIB_COUNTER.package_latch.all ;

-- Librairie portes standards + Mullers
library tast_component ;
use tast_component.tast_component.all;

Entity COUNTER_MP is
 Port (
        resetb :  in std_ulogic ;

        Go: in std_ulogic;  -- bit
        Go_req  : in std_ulogic ;
        Go_ack  : out std_ulogic ;

        S  : out std_ulogic_vector(2 downto 0);  -- bit[2..0]
        S_req  : out std_ulogic ;
        S_ack  : in std_ulogic
       ) ;
End COUNTER_MP;

Architecture ARCH_ COUNTER_MP of COUNTER_MP is
--Process MAIN
Component COUNTER_MAIN
  Port (
        resetb :  in std_ulogic ;

         -- données
         Go: in std_ulogic;  -- bit
         CS  : in std_ulogic_vector(2 downto 0);  -- bit[2..0]

         NS  : out std_ulogic_vector(2 downto 0);  -- bit[2..0]
         S  : out std_ulogic_vector(2 downto 0);  -- bit[2..0]

         -- contrôle
        Go_req : in std_ulogic ;
        Go_ack : out std_ulogic ;
        CS_req : in std_ulogic ;
        CS_ack : out std_ulogic ;

        NS_req  : out std_ulogic ;
        NS_ack  : in std_ulogic ;
        S_req  : out std_ulogic ;
        S_ack  : in std_ulogic )
end component;

--Process INCREMENT
Component COUNTER_INCREMENT
  Port (
        resetb :  in std_ulogic ;

         -- données
         NS_data  : in std_ulogic_vector(2 downto 0);  -- bit[2..0]
         CS_data  : out std_ulogic_vector(2 downto 0);  -- bit[2..0]

         -- contrôle
        NS_req : in std_ulogic ;
        NS_ack : out std_ulogic ;

        CS_req  : out std_ulogic ;
        CS_ack  : in std_ulogic )

end component;

Signal s_CS, s_NS : std_ulogic_vector(2..0);
Signal s_CS_req, s_NS_req : std_ulogic;
Signal s_CS_ack, s_NS_ack : std_ulogic;

Begin

I_COUNTER_MAIN  : COUNTER_MAIN port map(resetb, Go, s_CS, s_NS, S, Go_req, s_CS_req, Go_ack, s_CS_ack,  s_NS_req, S_req, s_NS_ack, S_ack);

I_COUNTER_INCREMENT : COUNTER_INCREMENT port map(resetb,
        s_NS, s_CS, s_NS_req, s_NS_ack,  s_CS_req, s_CS_ack);

End ARCH_COUNTER_MP;
```

A4.2--Composant “main” contenant 2 composants “main_control” pour le contrôle et “main_dp” pour le chemin de données

```
--Librairie IEEE standard
library IEEE ;
use IEEE .std_logic_1164.all ;

-- Librairie portes standards
library tast_component ;
use tast_component.tast_component.all;

Entity COUNTER_MAIN is
  Port (
        resetb :  in std_ulogic ;

         -- données
         Go_data: in std_ulogic;  -- bit
         CS_data  : in std_ulogic_vector(2 downto 0);  -- bit[2..0]

         NS_data  : out std_ulogic_vector(2 downto 0);  -- bit[2..0]
         S_data  : out std_ulogic_vector(2 downto 0);  -- bit[2..0]

         -- contrôle
        Go_req : in std_ulogic ;
        Go_ack : out std_ulogic ;
        CS_req : in std_ulogic ;
        CS_ack : out std_ulogic ;

        NS_req  : out std_ulogic ;
        NS_ack  : in std_ulogic ;
        S_req  : out std_ulogic ;
        S_ack  : in std_ulogic )
End COUNTER_MAIN,

Architecture Arch_COUNTER_MAIN of COUNTER_MAIN is
Component COUNTER_MAIN_CONTROL
  Port (
        resetb :  in std_ulogic ;

        Go_req : in std_ulogic ;
        Go_ack : out std_ulogic ;
        CS_req : in std_ulogic ;
        CS_ack : out std_ulogic ;

        NS_req  : out std_ulogic ;
        NS_ack  : in std_ulogic ;
        S_req  : out std_ulogic ;
        S_ack  : in std_ulogic ;

        -- les gardes Gi
        -- COUNTER_MAIN_G0,  COUNTER_MAIN_G1 : in std_ulogic ;

        --Une commande de latch pour chaque port de sortie
        Cmd_latch_NS,  Cmd_latch_S : out std_ulogic
        )
end component;

Component COUNTER_MAIN_DP
  Port (
        resetb :  in std_ulogic ;

        Go_data : in std_ulogic;  -- bit
        CS_data : in std_ulogic_vector(2 downto 0);  -- bit[2..0]

        NS_data : out std_ulogic_vector(2 downto 0);  -- bit[2..0]
        S_data : out std_ulogic_vector(2 downto 0);  -- bit[2..0]

        -- les gardes Gi
        COUNTER_MAIN_G0,  COUNTER_MAIN_G1 : out std_ulogic ;

        -- Une commande de latch pour chaque port de sortie
        Cmd_latch_NS,  Cmd_latch_S:  in std_ulogic
        )
end component;

signal s_COUNTER_MAIN_G0, s_COUNTER_MAIN_G1 : std_ulogic;
signal s_Cmd_latch_NS, s_Cmd_latch_S : std_ulogic;

BEGIN

I_COUNTER_MAIN_DP  :  COUNTER_MAIN_DP  port  map(resetb, Go_data,  CS_data,  s_COUNTER_MAIN_G0,  s_COUNTER_MAIN_G1, s_Cmd_latch_NS, s_Cmd_latch_S);

I_COUNTER_MAIN_CONTROL  :  COUNTER_MAIN_CONTROL  port map(resetb,  Go_req,  CS_req,  Go_ack,  CS_ack,  NS_req,  S_req, NS_ack,  S_ack,  s_COUNTER_MAIN_G0,  s_COUNTER_MAIN_G1, s_Cmd_latch_NS, s_Cmd_latch_S);

End ARCH_COUNTER_MAIN;
```

A4.3--Composant “increment” contenant 2 composants “increment_control” pour le contrôle et “increment_dp” pour le chemin de données

```
--Librairie IEEE standard
library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_unsigned.all;

-- Librairie portes standards
library tast_component ;
use tast_component.tast_component.all;


Entity COUNTER_INCREMENT is
  Port (
        resetb :  in std_ulogic ;

         -- données
         NS_data  : in std_ulogic_vector(2 downto 0);  -- bit[2..0]
         CS_data  : out std_ulogic_vector(2 downto 0);  -- bit[2..0]

         -- contrôle
        NS_req : in std_ulogic ;
        NS_ack : out std_ulogic ;

        CS_req  : out std_ulogic ;
        CS_ack  : in std_ulogic )

End COUNTER_INCREMENT,

Architecture Arch_COUNTER_INCREMENT of COUNTER_INCREMENT is
Component COUNTER_INCREMENT_CONTROL
  Port (
        resetb :  in std_ulogic ;

        NS_req : in std_ulogic ;
        NS_ack : out std_ulogic ;

        CS_req  : out std_ulogic ;
        CS_ack  : in std_ulogic ;

        -- les gardes Gi
        -- pas de gardes dans ce processus

        --Une commande de latch pour chaque port de sortie + une commande pour chaque port de sortie initialisé
        Cmd_latch_CS : out std_ulogic ;
        Cmd_latch_CS_init : out std_ulogic
        )
end component;

Component COUNTER_MAIN_DP
  Port (
        resetb :  in std_ulogic ;

        NS_data : in std_ulogic_vector(2 downto 0);  -- bit[2..0]
        CS_data : out std_ulogic_vector(2 downto 0);  -- bit[2..0]

        -- les gardes Gi
        -- pas de gardes dans ce processus

        -- Une commande de latch pour chaque port de sortie + une commande pour chaque port de sortie initialisé
        Cmd_latch_CS  :  in std_ulogic;
        Cmd_latch_CS_init:  in std_ulogic
        )
end component;

signal s_Cmd_latch_CS, s_Cmd_latch_CS_init : std_ulogic;

BEGIN


I_COUNTER_INCREMENT_DP : COUNTER_INCREMENT_DP port map(resetb, NS_data, CS_data, s_Cmd_latch_CS, s_Cmd_latch_CS_init);

I_COUNTER_INCREMENT_CONTROL  :  COUNTER_MAIN_CONTROL  port  map(resetb,  NS_req,  NS_ack,  CS_req,  CS_ack,  s_Cmd_latch_CS,
s_Cmd_latch_CS_init);

End ARCH_COUNTER_INCREMENT;
```

A4.4 Contrôleur du processus «main »

```
-- Librairie IEEE standard
library IEEE ; ….

Entity COUNTER_MAIN_CONTROL is
  Port (
        resetb :  in std_ulogic ;

        Go_req : in std_ulogic ;
        Go_ack : out std_ulogic ;
        CS_req : in std_ulogic ;
        CS_ack : out std_ulogic ;

        NS_req  : out std_ulogic ;
        NS_ack  : in std_ulogic ;
        S_req  : out std_ulogic ;
        S_ack  : in std_ulogic ;

        -- les gardes Gi
        -- COUNTER_MAIN_G0,  COUNTER_MAIN_G1 : in std_ulogic ;

        --Une commande de latch pour chaque port de sortie
        Cmd_latch_NS,  Cmd_latch_S : out std_ulogic  )
End COUNTER_MAIN_CONTROL;

Architecture ARCH_COUNTER_MAIN_CONTROL of COUNTER_MAIN_CONTROL is
   -- signaux AutoName (pour la Netlist)
    signal AutoName_0,  AutoName_1,  AutoName_2,  AutoName_3, AutoName_4 : std_ulogic;
    signal AutoName_5,  AutoName_6,  AutoName_7,  AutoName_8, AutoName_9 : std_ulogic;
    signal AutoName_10,  AutoName_11,  AutoName_12,  AutoName_13, AutoName_14 : std_ulogic;
    signal AutoName_15,  AutoName_16,  AutoName_17: std_ulogic;

Begin
--Equations – Requêtes de Sorties – WCHB
--Branch 0
--NS_0_req= S_0_req = maxdelay_counter_main_S_G0( MULLER2_R( ResetB,
                          MULLER2(NS_ack,S_ack),MULLER2DP(AND2(counter_main_G0,
                          maxdelay_counter_main_G0_G1(CS_req )), Go_req ) ) )

I_delay_0 : Delay generic map(maxdelay_counter_main_G0_G1) port map ( AutoName0, CS_req ) ;
AutoAnd_1 :  tast_and2 port map( AutoName1, counter_main_G0,  AutoName0) ;
AutoMuller_2 : tast_muller2DP port map( AutoName2,  AutoName1, Go_req) ;
AutoMuller_3 : tast_muller2 port map( AutoName3,  NS_ack, S_ack) ;
AutoMuller_4 : tast_muller2_R port map( resetb, AutoName4,  AutoName3, AutoName2) ;
I_delay_5 : Delay generic map(maxdelay_counter_main_S_G0) port map ( AutoName5, AutoName4 ) ;
NS_0_req <= AutoName5 ;
S_0_req <= AutoName5 ;

--Branch 1
-- NS_1_req = max_delay_NS_1( MULLER2_R( ResetbB, NS_ack,
            MULLER2(AND2( G_1, maxdelay_counter_main_G0_G1 ( CS_req )
                ),G0_req ) ) )

I_delay_6 : Delay generic map(maxdelay_counter_main_G0_G1) port map ( AutoName6, CS_req ) ;
AutoAnd_7 :  tast_and2 port map( AutoName7, counter_main_G1,  AutoName6) ;
AutoMuller_8 : tast_muller2 port map( AutoName8,  AutoName7, Go_req) ;
AutoMuller_9 : tast_muller2_R port map( resetb, AutoName9,  AutoName8, NS_ack) ;
I_delay_10 : Delay generic map(maxdelay_counter_main_S_G1) port map ( AutoName10, AutoName9 ) ;
NS_1_req <= AutoName10 ;

-- Commandes de latchs
-- cmd_latch_NS = OR[ NS_0_req, NS_1_req]
AutoOr_11 : tast_or2 port map( AutoName11,  AutoName5,  AutoName10) ;
cmd_latch_NS <= AutoName11 ;
---- cmd_latch_S =  S_0_req
cmd_latch_NS <= AutoName5 ;

--Total des requêtes de sorties
--NS_req = DELAY_LC[ cmd_latch_NS ]
I_delay_12 : Delay generic map(DELAY_LC) port map ( AutoName12, AutoName11 ) ;
NS_req <= AutoName12;

-- S_req = DELAY_LC[ cmd_latch_S ]
I_delay_13 : Delay generic map(DELAY_LC) port map ( AutoName13, AutoName5 ) ;
S_req <= AutoName13;

-- Equations Brutes – Acquittements d'entrées – WCHB
--Branch 0
-- CS_0_ack = NOT[ NS_0_req ]
 AutoNot_14 : tast_inv port map( AutoName14, AutoName5) ;
 CS_0_ack <= AutoName14 ;

  --Branch 1
  -- CS_1_ack = NOT[ NS_1_req ]
AutoNot_15: tast_inv port map( AutoName15, AutoName10) ;
CS_1_ack <= AutoName15 ;
```

```
-- Go_1_ack = NOT[ NS_1_req ]
AutoNot_16: tast_inv port map( AutoName16, AutoName10) ;
Go_1_ack <= AutoName16 ;

  -- Total
 --CS_ack = AND[ CS_0_ack,  CS_1_ack ]
 AutoAnd_17: tast_and2 port map(AutoName17, AutoName14,  AutoName15) ;
CS_ack <= AutoName17 ;

--Go_ack = Go_1_ack
Go_ack <= AutoName16;

End ARCH_COUNTER_MAIN_CONTROL;
```

A4.5 Chemin de donnée du processus «main »

```
-- Librairie IEEE standard
library IEEE ;
use IEEE .std_logic_1164.all ;

-- Librairie propre – package latch
library LIB_COUNTER;
use LIB_COUNTER.pack_latch.all;

entity COUNTER_MAIN_dp is
port(--les entrées de données
      Go_data : in std_ulogic ;
     --bit[3]=MR[2][3]->std_ulogic_vector((L*[Log2B]sup)-1)..0)
      CS_data : in std_ulogic_vector(2 downto 0);

      -- les sorties de données
      S_data : out std_ulogic_vector(2 downto 0);
      NS_data : out std_ulogic_vector(2 downto 0);

       -- les gardes Gi
      counter_main_G0 :  out std_ulogic;
      counter_main_G1 : out std_ulogic;

       -- une commande de latch par sortie de donnée
      Cmd_latch_S : in std_ulogic;
      Cmd_latch_NS : in std_ulogic)

end COUNTER_MAIN_dp;

architecture Arch_COUNTER_MAIN_dp of
                          COUNTER_MAIN_dp is
   -- signaux de garde et de sortie de données avant latchs
  signal s_G0, s_G1 :  std_ulogic;
  signal S_data_BFL, NS_data_BFL : std_ulogic_vector(2 downto 0);
```

```
begin
-- calcul des gardes en fonction des entrées de données
s_G_0 <= '1' when CS_data /="111" else
                '0';
s_G_1 <= '1' when CS_data = "111" else
               '0';
-- calcul des sorties de données avant latch en fonction des entrées
--de données selon la branche d'éxécution –Equations 4.37
---et 4.38 et union exclusive (garantie par l'exclusivité des gardes)
NS_data_BFL <= CS_data when (s_G_0 = '1') else
                        "000"  when (s_G_1 = '1') else
                        "---";
S_data_BFL <= CS_data when (s_G_0 = '1') else
                      "---";
-- affectation concurrente des gardes par les signaux de gardes
counter_main_G0 <= s_G_0;
counter_main_G1 <= s_G_1;

-- calcul des sorties de données à transmettre aux étages
-- suivant de pipeline par instantiation
instance_S_0 : latch_resetb port map( resetb, S_data(0),
                                      cmd_latch_S, S_data_BFL(0));
instance_S_1 : latch_resetb port map( resetb, S_data(1),
  cmd_latch_S, S_data_BFL(1));
instance_S_2 : latch_resetb port map( resetb, S_data(2),
  cmd_latch_S, S_data_BFL(2));

instance_NS_0 : latch_resetb  port map( resetb, NS_data(0),
                                cmd_latch_NS, NS_data_BFL(0) );
instance_NS_1 : latch_resetb  port map( resetb, NS_data(1),
                                cmd_latch_NS, NS_data_BFL(1) );
instance_NS_2 : latch_resetb  port map( resetb, NS_data(2),
                                cmd_latch_NS, NS_data_BFL(2) );

end Arch_COUNTER_MAIN_dp;
```

A4.6 Contrôleur du processus «increment»

```
-- Librairie IEEE standard
library IEEE ;
use IEEE .std_logic_1164.all ;

-- Librairie propre
library LIB_COUNTER ;
use LIB_COUNTER.package_delay.all ;

-- Librairie portes standards + Mullers
library tast_component ;
use tast_component.tast_component.all;

Entity COUNTER_INCREMENT_CONTROL is
  Port (
        resetb :  in std_ulogic ;
        NS_req : in std_ulogic ;
        NS_ack : out std_ulogic ;

        CS_req  : out std_ulogic ;
        CS_ack  : in std_ulogic ;

        -- les gardes Gi
        -- pas de garde dans le process increment

        -- une commande de latch pour chaque port de sortie
        Cmd_latch_CS  : out std_ulogic;
        Cmd_latch_CS_init  : out std_ulogic)
End COUNTER_MAIN_CONTROL;

Architecture ARCH_COUNTER_INCREMENT_CONTROL of COUNTER_INCREMENT_CONTROL is

   -- signaux AutoName (pour la Netlist)
    signal AutoName_0,  AutoName_1,  AutoName_2,  AutoName_3, AutoName_4 : std_ulogic;
    signal AutoName_5,  AutoName_6,  AutoName_7,  AutoName_8, AutoName_9 : std_ulogic;
    signal AutoName_10,  AutoName_11,  AutoName_12,  AutoName_13, AutoName_14 : std_ulogic;
    signal AutoName_15,  AutoName_16,  AutoName_17: std_ulogic;

-- Signaux intermédiaires nécessaires pour rajouter le bloc d'initialisation
 sig_CS_req : std_ulogic ;
 sig_CS_ack : std_ulogic ;


BEGIN

-- Bloc de contrôle de base
-- cmd_latch_CS = maxdelay_counter_increment_S[ MULLER2_R( resetb, NS_req,  sig_CS_ack )  ]
AutoMuller_0 : Muller2_R port map( resetb, AutoName0, NS_req,  sig_CS_ack ) ;
I_delay_1 : Delay generic map(maxdelay_counter_increment_S) port map ( AutoName1, AutoName0 ) ;
cmd_latch_CS  <= AutoName1 ;

-- sig_CS_req =DELAY_LC[ cmd_latch_CS]
I_delay_2 : Delay generic map(DELAY_LC) port map ( AutoName2, AutoName1 ) ;
sig_CS_req <= AutoName2;


-- NS_ack = NOT[cmd_latch_CS]
AutoNot_3 : tast_inv port map( AutoName3,  AutoName1) ;
NS_ack <= AutoName3 ;

-- Bloc de contrôle pour l'initialisation
-- cmd_latch_CS_init =  MULLER2_S(  set,  sig_CS_req, CS_ack  )
AutoMuller_4 : Muller2_S port map( set, AutoName4,  sig_CS_req, CS_ack  ) ;
cmd_latch_CS_init  <=  AutoName4 ;

-- CS_req = AND(   DELAY_LC[ cmd_latch_CS_init]   ,    resetb      )
I_delay_5 : Delay generic map(DELAY_LC) port map ( AutoName5, AutoName4 ) ;
AutoAnd_6 :  AND port map(AutoName6,  resetb,  AutoName5 );
CS_req <= AutoName6;

--sig_CS_ack = NOT[cmd_latch_CS_init ]
AutoNot_7 : tast_inv port map( AutoName7,  AutoName4) ;
sig_CS_ack <= AutoName7 ;
End ARCH_COUNTER_INCREMENT_CONTROL;
```

A4.7 Chemin de donnée du processus «increment »

```
-- Librairie IEEE standard
library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_unsigned.all;

-- Librairie propre – package latch
library LIB_COUNTER;
use LIB_COUNTER.pack_latch.all;
--  et des latch sans reset ni set

Entity COUNTER_INCREMENT_dp is
 Port (
       resetb :  in std_ulogic ;
       NS_data : in std_ulogic_vector(2 downto 0);  -- bit[3]
       CS_data : out std_ulogic_vector(2 downto 0);  -- bit[3]

       -- les gardes Gi
       -- pas de gardes

       -- une commande de latch par sortie de donnée (bloc de
base) +
       -- une commande de latch pour le bloc d'initialisation
       Cmd_latch_CS :  in std_ulogic;
       Cmd_latch_CS _init  :  in std_ulogic
      );
End COUNTER_INCREMENT_dp;

Architecture Arch_COUNTER_INCREMENT_dp of
COUNTER_INCREMENT_dp is
-- Signaux intermédiaires
-- signaux de données de sortie avant latch
 s_ CS _BFL : std_ulogic_vector(2 downto 0);
-- signaux de données de sortie entre le
-- latch de base et le latch d'initialisation
 s_ CS _AFL : std_ulogic_vector(2 downto 0);

 Begin
-- Sortie avant latchs
s_ CS _BFL  <=
  To_StdULogicVector(To_StdLogicVector (NS_data)+"001") ;

-- Sortie de données entre latchs et après latchs
Instance_s_CS_AFL_0: latch_set port map(set, s_CS_AFL(0),
                     Cmd_latch_CS,  s_CS_BFL(0));
Instance_CS_0: latch  port map(CS_data(0),
                     Cmd_latch_CS_init, s_CS_AFL(0) ) ;

Instance_s_CS_AFL_1: latch_resetb
                     port map(resetb, s_CS_AFL(1),
                     Cmd_latch_CS, s_CS_BFL(1));
Instance_CS _1: latch  port map(CS_data(1),
                     Cmd_latch_CS_init, s_CS_AFL(1) ) ;

Instance_s_CS_AFL_2: latch_resetb
                     port map(resetb, s_CS_AFL(2),
                     Cmd_latch_CS,s_CS_BFL(2));
Instance_CS _2: latch  port map(CS_data(2),
                     Cmd_latch_CS_init, s_CS_AFL(2) ) ;

End Arch_COUNTER_INCREMENT_dp;
```

RESUME

Les circuits asynchrones ont des caractéristiques qui les démarquent nettement des circuits synchrones : modularité quasi-parfaite, absence d'horloge, contrôle local. Ils tendent à constituer une sérieuse alternative pour pallier aux problèmes posés par l'intégration en silicium d'applications de plus en plus complexes. Le goulot d'étranglement principal pour l'adoption de la conception des circuits asynchrones se situe au niveau du manque de méthodologies et d'outils puissants pour ce type de conception. Ce travail de thèse porte sur la définition d'une méthodologie de conception de circuits intégrés asynchrones micropipeline. La synthèse micropipeline est une approche qui exploite à la fois les outils commerciaux de synthèse pour le chemin de données, et la synthèse de contrôleurs asynchrones pour le contrôle (« STG » avec Petrify , « BURST MODE » avec Minimalist).

La méthodologie générale pour la modélisation et la synthèse de circuits asynchrones est basée sur la spécification dite DTL (Data Transfer Level) qui définit une façon d'écrire les codes sources garantissant une synthèse rapide et systématique pouvant cibler plusieurs styles de circuits asynchrones. Cette méthode de conception part d'une spécification basée sur un langage de haut niveau (CHP ou Concurrent Hardware Processes). Elle permet en sortie de générer des circuits en portes logiques élémentaires et en portes de Muller. Il a été procédé à un prototypage de cette méthode de synthèse. Ce prototype est conçu pour être intégré dans l'outil de conception automatique de circuits asynchrones TAST (Tima Asynchronous Synthesis Tool) dont le synthétiseur génère des circuits asynchrones QDI, pour l'étendre à la génération de circuit micropipelines. Par ailleurs, la méthodologie de synthèse a été étendue à l'utilisation de différents types de contrôleurs asynchrones susceptibles d'en améliorer les performances en termes de vitesse et de consommation.

MOTS-CLES

Circuits asynchrones, méthodologie de conception, synthèse de circuits asynchrones, circuits micropipeline, synthèse de contrôleurs asynchrones, processus concurrents communicants, réseau de Pétri, protocoles de communications, langage CHP, graphes de flot de données, équations de dépendances, outils de synthèse.

TITLE

Logical synthesis of micropipeline asynchronous circuits

ABSTRACT

The inherent asynchronous circuit features (modularity, clockless system, local control) brings a serious alternative to face the problems encountered by the silicon integration of more and more complex applications. The main bottleneck to adopt the asynchronous logic is due to the lack of methodologies and efficient tools for this kind of design. The thesis works aim to define a micropipeline asynchronous design methodology. The micropipeline synthesis approach use both commercial tools for data path synthesis and specific tools for asynchronous control synthesis (« STG » using Petrify, « BURST MODE » using Minimalist).

The overall methodology for the modelling and the synthesis of asynchronous circuits is based on the DTL specification (Data Transfer Level) which assumes a restriction of source code allowing a rapid and systematic synthesis and targeting several kinds of asynchronous circuits. This design methodology starts from a high level programming language named CHP (Concurrent Hardware Processes) and generates a gate netlist composed of elementary logic and Muller gates. This synthesis methodology has been prototyped. This prototype has been designed for its integration in the TAST automatic asynchronous design flow (Tima Asynchronous Synthesis Tool) which generate QDI circuits, to spread it in the generation of micropipelines circuits. Furthermore, the synthesis methodology has been extended for different kinds of asynchronous controller to improve performances such as speed and energy consumption.

KEYWORDS

Asynchronous circuits, Design Methodology, Asynchronous circuits synthesis, Micropipeline circuits, Asynchronous Controlers synthesis, Concurrent communicating processes, Pétri nets, Communication protocols, CHP langage, Data flow graphs, Dépendancies equations, Synthesis tools.

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France.

ISBN : 2-84813-046-6 (version brochée)
ISBNE : 2-84813-047-4 (version électronique)